EXTENDING DYNAMIC SCRIPTING

by

JEREMY R. LUDWIG

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2008

“Extending Dynamic Scripting,” a dissertation prepared by Jeremy R. Ludwig in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:

Arthur Farley, Chair of the Examining Committee

Date

Committee in Charge: Arthur Farley, Chair
Li-Shan Chou
Dejing Dou
Steve Fickas
Prasad Tadepalli

Accepted by:

Dean of the Graduate School

© 2008 Jeremy R. Ludwig

An Abstract of the Dissertation of

Jeremy R. Ludwig for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken December 2008

Title: EXTENDING DYNAMIC SCRIPTING

Approved: ____
Arthur Farley

The dynamic scripting reinforcement learning algorithm can be extended to improve the speed, effectiveness, and accessibility of learning in modern computer games without sacrificing computational efficiency. This dissertation describes three specific enhancements to the dynamic scripting algorithm that improve learning behavior and flexibility while imposing a minimal computational cost: (1) a flexible, stand alone version of dynamic scripting that allows for hierarchical dynamic scripting, (2) a method of using automatic state abstraction to increase the context sensitivity of the algorithm, and (3) an integration of this algorithm with an existing hierarchical behavior modeling architecture. The extended dynamic scripting algorithm is then examined in the three different contexts. The first results reflect a preliminary investigation based on two abstract real-time strategy games. The second set of results comes from a number of abstract tactical decision games, designed to demonstrate the strengths and weaknesses of extended dynamic scripting. The third set of results is generated by a series of experiments in the context of the commercial computer role-playing game Neverwinter Nights demonstrating the capabilities of the algorithm in an actual game. To conclude, a number of future research directions for investigating the effectiveness of extended dynamic scripting are described.

CURRICULUM VITAE

NAME OF AUTHOR: Jeremy R. Ludwig

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon; Eugene, Oregon
University of Pittsburgh; Pittsburgh, Pennsylvania
Iowa State University; Ames, Iowa
University of Northern Iowa; Waterloo, Iowa

DEGREES AWARDED:

Doctor of Philosophy, Computer Science, 2008, University of Oregon
Master of Science, Computer Science, 2000, University of Pittsburgh
Bachelor of Science, Computer Science with Minors in Psychology and Philosophy, 1997, Iowa State University

AREAS OF SPECIAL INTEREST:

Machine Learning
Behavior Modeling
Artificial Intelligence

PROFESSIONAL EXPERIENCE:

Artificial Intelligence Researcher, Stottler Henke Associates, 2000-Present
Graduate Student Researcher, Intelligent Systems Lab at the University of Pittsburgh, 1999-2000
Software Engineer, Engineering Animation Inc., 1997-1998
Undergraduate Research Assistant, Artificial Intelligence Research Group at Iowa State University, 1996-1997
Undergraduate Research Assistant, Center for Non-Destructive Evaluation Ames Lab, 1996

GRANTS, AWARDS AND HONORS:

$100,000, Second language retention intelligent training system, ONR, 2008
$70,000, Rapid simulation development processes and tools for job performance assessment, ARI, 2007
$100,000, Asymmetric adversaries for synthetic training environments, OSD, 2007
$100,000, Effectively communicating medical risks, OSD, 2002

PUBLICATIONS:

Ludwig, J., Jensen, R., Proctor, M., Patrick, J., & Wong, W. (2008). Generating adaptive behaviors for simulation-based training. In *Adaptive Agents in Cultural Contexts: Papers from the AAAI Fall Symposium*. Menlo Park, CA: AAAI Press.

Ludwig, J., & Farley, A. (2007). A learning infrastructure for improving agent performance and game balance. In *Optimizing Player Satisfaction: Papers from the 2007 AIIDE Workshop*. Stanford, CA: AAAI Press.

Ludwig, J., Livingston, G., Vozalis, E. & Buchanan, B. (2000). What's new? Using prior models as a measure of novelty in knowledge discovery. In *Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence*. Washington, DC: IEEE Computer Society.

Ludwig J, Fine MJ, Livingston G, Vozalis E, & Buchanan B (2000). Using prior models as a measure of novelty in knowledge discovery. In J. Overhage (Ed.) *Proceedings of the 2000 AMIA Annual Symposium*.

ACKNOWLEDGMENTS

I would like to first acknowledge all of the people who contributed scientifically to the work presented in this dissertation. My advisor Arthur Farley, along with my committee members Li-Shan Chou, Dejing Dou, Steve Fickas, and Prasad Tadepalli, have all been indispensable resources in completing this research. I especially appreciate all of the insight and guidance that Art provided throughout this research and all of the time he spent helping me become a better scientist. My friend and colleague Dan Fu has contributed much in the way of suggestions and encouragement.

I am also grateful for the support given by my family and friends in this endeavor. Most importantly, my wife Jennifer has given unconditional support and encouragement and my son Oscar has helped me to keep everything in perspective. My parents (Angie, Myron, Bob, Lynn) and in-laws (Steve, Jean) have given both love and encouragement. Andrew Christianson has been a good friend and sounding board throughout this process. The late James Welton helped me understand that I could achieve more than I thought possible. During my time at the University of Oregon and at the University of Pittsburgh I have met a number of students and faculty that have had a significant impact on me. Of particular importance are Peter Boothe, Julian Catchen, Anthony Hornof, Kevin Huck, Sailesh Ramakrishnan, and Will Bridewell. There are also a number of colleagues from Stottler Henke who supported me in this undertaking. Of particular importance are Ryan Houlette, Jim Ong, Sowmya Ramachandran, Rob Richards, and Dick Stottler.

Finally, I owe a deep debt of gratitude to my previous research advisors Bruce Buchanan and Vasant Honavar. Without their friendship and advice, I would never have attempted this undertaking.

To Jennifer,

for everything.

TABLE OF CONTENTS

Chapter Page

Chapter Page

Chapter Page

APPENDICES

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

Figure Page

LIST OF TABLES

CHAPTER I
INTRODUCTION

The desire to create computer programs that can play games has been a driving force in artificial intelligence research, starting with Samuel's groundbreaking work in developing a computer program to play checkers in the early 1950's (Russell & Norvig, 1995). A recent special issue of *Machine Learning* demonstrates the continuing significance of game-based artificial intelligence (AI) research (Bowling, Fürnkranz, Graepel, & Musick, 2006).

While there has been a significant amount of research on classic two-player turn-based games like checkers, chess, and backgammon (Russell & Norvig, 1995) and on multi-player card games such as poker (e.g., see poker.cs.ualberta.ca), this dissertation focuses on the modern interactive computer games advocated by Laird and van Lent (2001) for the study of human-level AI. These include common commercial computer games such as first person shooters (FPS), real-time strategy (RTS), and role-playing (RPG) games. Examples of these game types used in computer science research are Unreal Tournament, Warcraft, and NeverWinter Nights.

The main difference between creating agents (computer programs) that play classical games such as checkers or poker and modern computer games is that the latter requires a much broader behavior model (J.E. Laird & van Lent, 2001). For example, an agent that plays chess is expected to examine the current board positions and recommend a move given the current difficulty setting. In a role-playing game a computer controlled character is expected to act more like a human – carrying on limited conversations with the human player and advancing the story arc in addition to intelligently assisting in combat. In classical games, the problem being solved by the agent can often be framed as a search problem, where there exist some standard, reasonable, solutions (e.g., expectimax (Russell & Norvig, 1995)). In modern computer games, the challenge is to create a broad, complex, behavior model that is often difficult to represent directly as a search problem.

The increasing complexity of desired agent behavior has resulted in a number of behavior modeling architectures that are designed to support the creation of agent behavior (behavior models) for modern computer games. While in many cases architectures are written specifically for a single game, there are a number of general purpose behavior modeling architectures. These are referred to as AI-middleware in the game industry (Flemming, 2008) and share much in common with the integrated development environments common in software engineering. The goal of a behavior modeling architecture is to make it easier to specify complex agent behavior.

After agent behavior has been specified in a behavior modeling architecture, machine learning techniques can be used to adapt agent behavior *online* – i.e. while the human player is playing the game on their own computer. Reinforcement learning (RL) (Sutton & Barto, 1998) is a broad class of machine learning algorithms commonly used to perform online adaptation in games. RL algorithms learn to take actions that result in the greatest reward based on feedback from the game. For example, an agent in a first person shooter could use a RL algorithm to learn whether to act *aggressively* or *defensively* based on how well the selected behavior performs against a particular human player. Dynamic scripting is an instance of a RL algorithm designed specifically for modern computer games (Spronck, Ponsen, Sprinkhuizen-Kuyper, & Postma, 2006).

While the dynamic scripting algorithm has shown significant promise in controlling agent behavior in modern computer games, there are a number of deficiencies that need to be addressed. First, there is currently no dynamic scripting implementation that has been integrated with a behavior modeling environment. This means that in order to use dynamic scripting, a modeler must implement the dynamic scripting algorithm from scratch. Additionally, most behavior modeling environments are hierarchal in nature, so the modeler would need to extend the basic dynamic scripting algorithm so that modular techniques such as task decomposition could be used. Second, there is a need to improve the ability of dynamic scripting to take advantage of state information. While algorithms such as Q-learning can take into account all of the state information available in a game and learn what actions to perform in specific game states, dynamic scripting has previously been used in cases where either only a single game state, or a limited set of game states, are considered. What is needed is to address these shortcomings is a flexible, dynamic scripting-based behavior modeling tool that is able to support task decomposition and to take into account game state information.

Our thesis is that dynamic scripting can be extended to improve the learning performance, applicability, and flexibility of dynamic scripting-based learning in modern computer games with reasonable computational cost. This dissertation proposes three specific extensions to the dynamic scripting algorithm that improve learning behavior and flexibility while imposing a minimal cost: (1) developing a flexible, stand alone, version of dynamic scripting that allows for **hierarchical dynamic scripting**; (2) extending the **context sensitivity** of hierarchical dynamic scripting through automatic abstract state construction; and (3) performing an **architecture integration** where this algorithm is integrated with an existing hierarchical behavior modeling architecture. The result of this research will be to create a useful framework for game developers, where they can easily define adaptive behaviors that utilize dynamic scripting-based learning techniques.

In order to verify the claims of performance, applicability, flexibility, and computational cost, results generated by the extended algorithm are compared to results from the standard dynamic scripting algorithm. Learning performance will be measured in terms of the relative score achieved by the agent and how fast the agent was able to achieve this score, given a set number of learning opportunities. Applicability is evaluated by examining whether or not the learning algorithm can be used to play a particular game. Flexibility is demonstrated by the ability of the author to express different types of domain knowledge in the authoring of agent behaviors. Computational cost is the amount of CPU time it takes the algorithm to select an action in the game and to learn from its experience.

Chapter II is devoted to providing the background knowledge necessary to understand this thesis statement. First, it gives an introduction on the use of artificial intelligence in modern computer games. Following this is a narrower, but deeper, examination of the use of online machine learning in games focusing on reinforcement learning. Finally, the last section in this chapter describes the dynamic scripting algorithm in detail, contrasting this algorithm with more commonly used reinforcement learning algorithms.

Chapter III is an in-depth examination of the extensions to the dynamic scripting algorithm (EDS). This research builds on extensions that have worked well in the context of standard reinforcement learning algorithms and recasts them to work in the context of dynamic scripting and complex computer games. Related work is described in this chapter as well and includes summaries of related research on the dynamic scripting algorithm, on tree-based state abstraction and on the use of reinforcement learning algorithms in behavior modeling architectures. The specific contribution of this dissertation with respect to previous research is clearly outlined in this section. This chapter concludes with an illustrative example of EDS in action. This illustration is performed in the context of two simple real-time strategy sub-games and illustrates the main features of EDS.

Chapter IV presents an abstract game description language that supports the construction of abstract tactical games. Four distinct games are created in this framework, ranging from entire simple prediction games to elements of more complex real-time strategy and role playing games. These four games are used experimentally to evaluate the performance of EDS in a variety of different gaming environments.

In Chapter V, EDS is demonstrated in a modern computer game. Using the game Neverwinter Nights, the performance of the extended algorithm is directly compared to the performance of the dynamic scripting algorithm and to Q-learning variants. This comparison is a series of experiments that demonstrate the impact of each extension separately.

Chapter VI concludes this dissertation, beginning with a summary of the presented research, results, and contributions. This chapter goes on to discuss the types of learning problems where the extended dynamic scripting algorithm is expected to excel. The final section of the chapter is devoted to discussing future research based on extended dynamic scripting.

CHAPTER II
BACKGROUND

The goal of this chapter is to provide the relevant background information for EDS. First, it gives a broad introduction on the use of artificial intelligence in modern computer games. Following this is a narrower, but deeper, examination of online learning in games. The next section covers a particular type of online learning, reinforcement learning, in the context of modern computer games. Finally, the last section in this chapter describes the dynamic scripting algorithm in detail.

Artificial Intelligence in Modern Computer Games

A primary application of artificial intelligence (AI) algorithms in modern games is control of the behavior of simulated entities (agents) within a game. For example, a computer controlled opponent in a first person shooter is expected to aim their weapon and fire upon the human player when the agent perceives the human player. Agent behaviors such as this are developed using a diverse set of behavior modeling architectures, where a behavior modeling architecture is a specialized programming environment designed to make it easier to author, modify, debug, and run agent behavior. A behavior author programs behavior for an agent in a specific architecture, resulting in a behavior model that can be used to control the agent at runtime.

A behavior modeling architecture is defined in a large part by the type of model representation it depends upon. Different modeling representations can define behavior in C++/Java code, scripts (Spronck et al., 2006) or finite state machines (Fu & Houlette, 2003b); on the fly by a planning system (Orkin, 2004); from hybrid architectures that plan within or over hierarchal task networks (Hoang, Lee-Urban, & Muñoz-Avila, 2005); or with production rule based systems such as ACT-R (Best, Lebiere, & Scarpinatto, 2002) and Soar (Wray, Laird, Nuxoll, Stokes, & Kerfoot, 2005). Although there have been efforts to automate this process (e.g. Ponsen, Spronck, Muñoz-Avila, & Aha, 2006), agent behaviors are generally created by a computer programmer (behavior author) in an integrated development environment (Ludwig & Houlette, 2006).

Below are two behavior model examples where each model makes use of a different behavior architecture. The first is an example of a hierarchical state machine for the first-person-shooter game Counterstrike. The second example uses a cognitive architecture to control an agent in a simple tank game.

Hierarchical Task Network Example

This example uses a graphical canvas to describe reactive agent behavior in the first-person-shooter game Counterstrike. A screen shot from the game, where the player is wearing night vision goggles, is shown in Figure 1.

(RADIO) 大野狼: Fire in the hole!
Rakarth has joined the game
(RADIO) 大野狼: Fire in the hole!

Figure 1 Image from the game CounterStrike ("CS: Source beta," 2004).

A task network that moves the agent to a destination is shown in Figure 2. In this diagram, rectangles represent actions that the character takes in the game and ovals represent observations the agent makes about the game state. Bold rectangles, such as *Attack*, indicate that the action is another task network, while non-bold rectangles indicate a primitive action. Ordered and directed arrows show the flow of control from one action to another. Behavior starts at the initial node (top left) and transfers control to the *FollowPath* action. If the observation *AtDestination* is true, then control transfers to the final node (bottom left) and the behavior exits. This type of diagram, from the SimBionic modeling tool, appears throughout this document (see APPENDIX A SIMBIONIC GLOSSARY for a glossary).

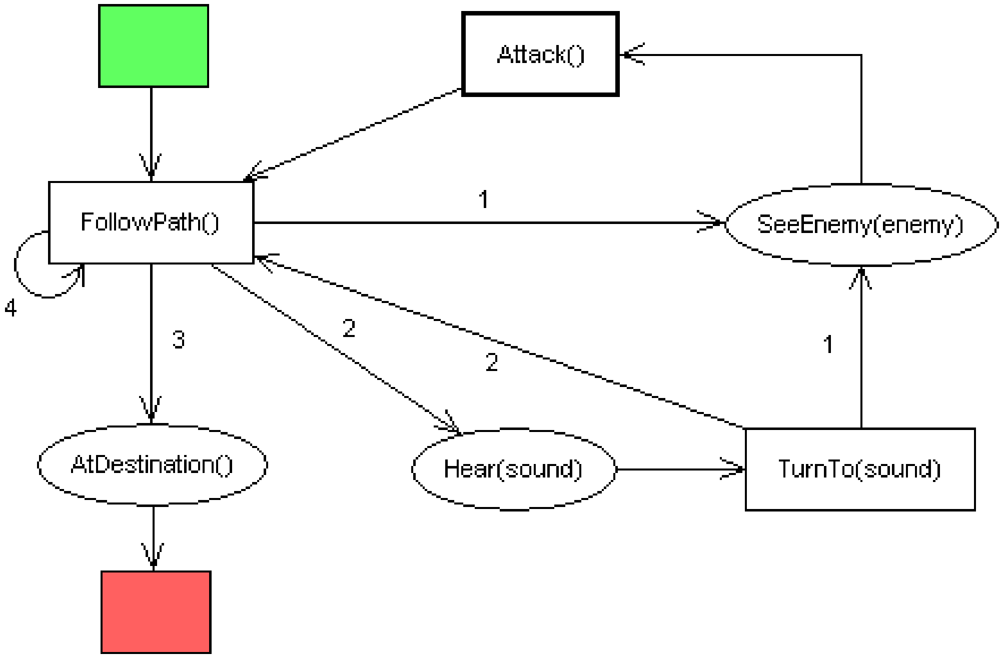


Figure 2 Move to destination behavior for Counterstrike (Ludwig & Houlette, 2006).

There are two other transitions out of the *FollowPath* action. The first checks to see if an enemy is seen while following the given path, and the second checks for hearing an enemy. If one of these observations is true the Attack sub-task is started as shown in Figure 3.

The *Attack* behavior attempts to shoot at the enemy and then takes one of four transitions. The first transition checks to see if the enemy is dead, the second if the agent is out of ammo, and the third checks to make sure that the agent can still see the enemy. If none of these are true, the fourth transition will be taken and the behavior will exit. This will return control to the parent behavior, which resumes the *FollowPath* task.

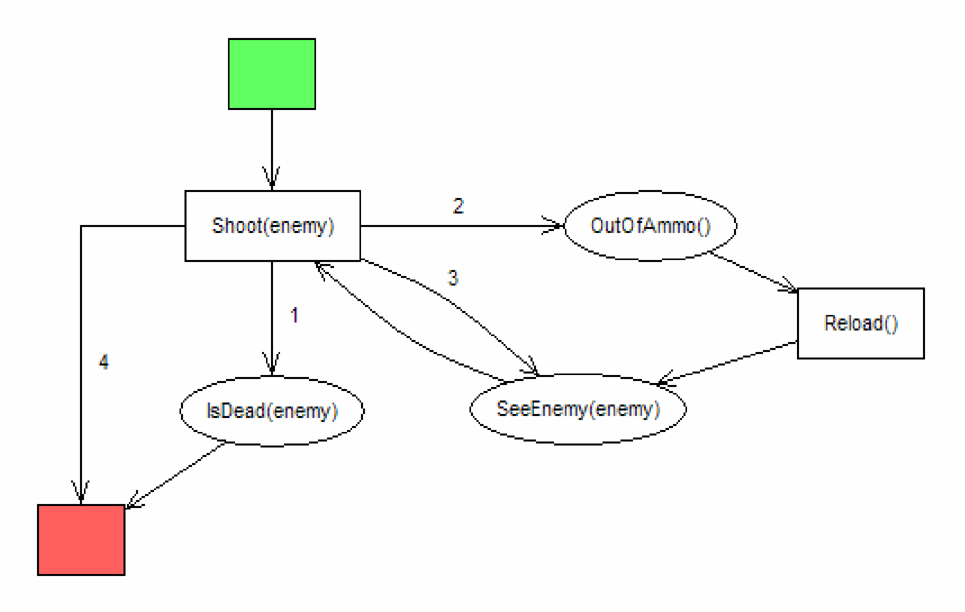


Figure 3 Attack sub-behavior for Counter Strike (Ludwig & Houlette, 2006).

Behavior modeling techniques based on state-machines are very popular in the gaming industry because they are easy to implement, computationally efficient, an intuitive representation of behavior, accessible to subject matter experts in addition to programmers, relatively easy to maintain, and can be developed in a number of commercial integrated development environments (Fu & Houlette, 2003b). The downside to state-machine approaches is that by focusing on simplicity of representation and computational efficiency they often do not include advanced capabilities such as learning and planning / problem solving. The lack of an ability to perform problem solving or learn from experience can result in a brittle system where the agent behavior can't cope with unforeseen situations or learn from its mistakes.

As an example of a problem related to lack of planning capabilities, Orkin (2006) discusses an agent that is in a room when it sees its human opponent outside. One way of writing the adversary in a state-machine would cause it to open the door and move into the same room as its opponent. However, the human opponent could decide to stand in front of the door to block it from being opened. If the FSM does not include any contingency behaviors, the agent is stuck in the room. Orkin contrasts this to a planning-based system capable of re-planning to accomplish the agent's goals (moving to the same room as the opponent) with other actions. These new plans could include trying to kick the door down or exiting through a window. While the same set of actions could be performed in either architecture, the FSM requires that all plans be constructed ahead of time by the behavior author rather than in real-time by the planning system.

The following section provides a detailed example of a plan-based approach using the Soar cognitive architecture.

Cognitive Architecture Example

This example, from the Soar tutorial (J.E. Laird & Congden, 2005), illustrates a number of the features of the Soar cognitive architecture. It focuses on the ability of the architecture to create plans to control agent behavior in the TanksSoar game, as opposed to running through a pre-created state machine. These plans are represented as a series of IF-THEN rules, known as production rules, that link together in a very specific way.

Each tank in the game is controlled by a separate instance of a Soar behavior model. In controlling a tank the model is responsible for moving, turning, firing missiles, listening for sounds, using the radar, etc., while conserving limited resources such as energy, missiles, and health. Figure 4 shows an example game with four tanks, each represented by a diamond. Health and energy for each tank are shown on the left. The tutorial includes a default model for controlling the tanks, which is described in detail below.

The Soar production rules, combined with game state information, control the actions taken by each tank at each decision point. There are a number of different production rule types that are considered by Soar in order, each time a decision is made for a tank: *elaborations*, *propose*, *prefer*, and *apply*. First *elaborations* augment the current game by creating new attribute-value pairs that become a temporary part of the game state information. Second the *propose* rules enumerate the possible actions (known in Soar as operators) that the tank can take in the current state. Third the *prefer* rules order the possible actions. Finally, *apply* rules are used to inform the tank of the single action that it should perform. The behavior author is responsible for creating the production rules that are used in these fours steps. Each of these steps is described below, including example production rules in Soar syntax.

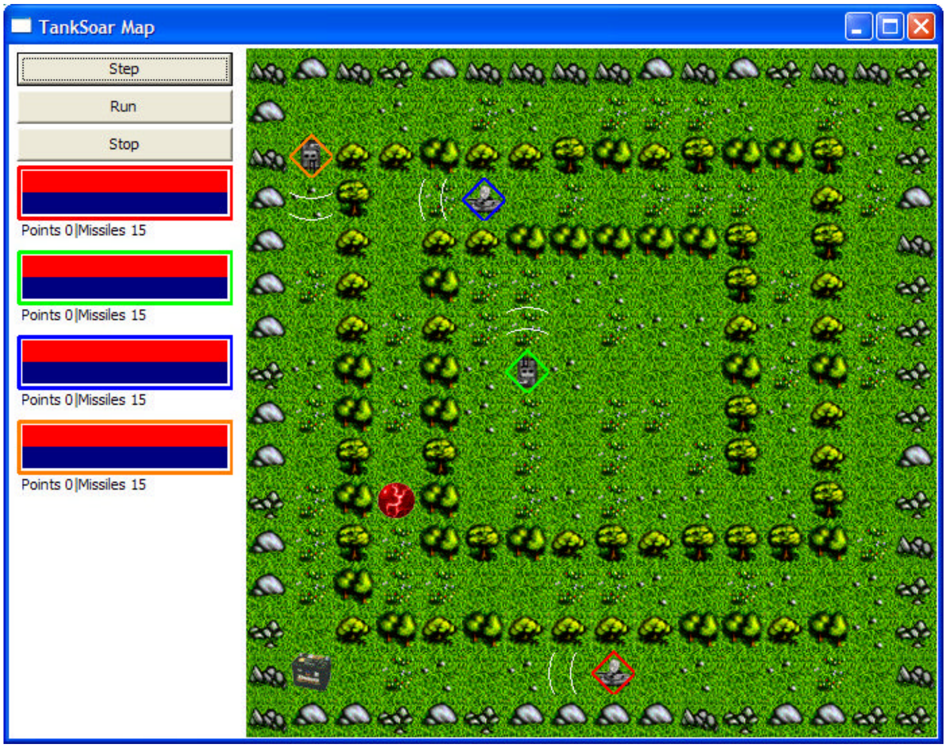


Figure 4 Screen image from the TankSoar game.

Elaborations

Elaborations are used to augment the observed game state with additional information. The following two rules are both used to elaborate the observed state with the value *missiles-energy low*. The first rule does this if the number of missiles is low and the second if the energy is low.

```
sp {elaborate*state*missiles*low
   (state <s> ^name tanksoar
           ^io.input-link.missiles 0)
-->
   (<s> ^missiles-energy low)
}

sp {elaborate*state*energy*low
   (state <s> ^name tanksoar
              ^io.input-link.energy <= 200)
```

```
      -->
         (<s> ^missiles-energy low)
      }
```

These elaborations become part of the game state and can be used to activate other production rules.

Propose

Soar models are based on the idea of proposing and selecting actions using all available knowledge. The following three rules propose to move forward, turn to the left or right, or to turn around. In some instances, these three rules will propose a number of operators simultaneously. All three of these rules are only applicable when the agent's current goal is *wander*. When a different goal is active then different action proposal rules would come into play.

The first rule proposes a forward move, if the way forward is not blocked.

```
      sp {propose*move
         (state <s> ^name wander
                    ^io.input-link.blocked.forward no)
      -->
         (<s> ^operator <o> +)
         (<o> ^name move
              ^actions.move.direction forward)}
```

The second rule states that if the way forward is blocked, and left or right is not blocked, then propose to turn left or right and turn on the radar. Note that this production rule will fire twice if both the left and right directions are not blocked.

```
      sp {propose*turn
         (state <s> ^name wander
                    ^io.input-link.blocked <b>)
         (<b> ^forward yes
              ^ { << left right >> <direction> } no)
      -->
         (<s> ^operator <o> + =)
```

```
   (<o> ^name turn
        ^actions <a>)
   (<a> ^rotate.direction <direction>
        ^radar.switch on
        ^radar-power.setting 13)
```

The third rule states that if the forward, left, and right directions are all blocked then the agent should turn to the left. This initiates a complete turn based on the actions of later agent decisions.

```
sp {propose*turn*backward
   (state <s> ^name wander
              ^io.input-link.blocked <b>)
   (<b> ^forward yes ^left yes ^right yes)
-->
   (<s> ^operator <o> +)
   (<o> ^name turn
        ^actions.rotate.direction left)
}
```

Prefer

When more than one operator is proposed for a single decision, a selection process makes use of preference rules to decide which operator to choose. For example, the following rule prefers the *radar-off* operator to a *turn* or *move* operator.

```
sp {select*radar-off*move
      (state <s> ^name wander
      ^operator <o1> +
      ^operator <o2> +)
      (<o1> ^name radar-off)
      (<o2> ^name << turn move >>)
-->
      (<s> ^operator <o1> > <o2>)
}
```

The Soar architecture has a built in mechanism for selecting a single action from all of the proposed actions. This mechanism takes into account the user-defined preference rules. Once the decision is made, *apply* rules tell the agent what action to take.

Apply

Finally, separate rules apply the selected operator by telling the agent to perform the specified action. In this particular rule, the agent is told to move in the given direction.

```
sp {apply*move
   (state <s> ^operator <o>
              ^io.output-link <out>)
   (<o> ^direction <direction>
        ^name move)
-->
   (<out> ^move.direction <direction>)
}
```

Summary

The advantage of production systems such as Soar is that they support aspects of intelligent behavior, such as problem solving and learning, not supported by other types of architectures. For example, if the current goal is *wander* and the way forward is clear the agent will continue to select the production rule for going forward. When the agent runs into a wall with the same goal, it will re-plan by proposing two different actions and then use preference operators to decide between the two possibilities. If the agent can not decide between the two, an *impasse* is reached and a new subgoal is created. Soar attempts to resolve the subgoal by searching for the best solution. If resolved, Soar's learning mechanism (called *chunking)* creates a new production rule to remember what to do if the same problem arises again. When the goal changes from *wander* to *rechargeEnergy* then a different set of production rules will become active which help the agent find an energy source. The drawback to production systems, as seen in the simple model above, is that developing agent behavior that is less brittle comes with a price - it can be difficult to author, modify, and debug complicated sets of production rules.

Online Learning

Once created, agent behavior can be altered by applying machine learning techniques. This section discusses two different ways of applying machine learning to games, off-line and online learning, and then focuses on the online learning issues relevant to EDS.

There are two broad categories of techniques: offline and online learning. Offline learning techniques are those performed before the game AI is distributed. For example, in the car-racing game ReVolt a genetic algorithm was used to determine the best racing paths for the computer controlled drivers (J. E. Laird & van Lent, 2005). As players use the game, the pre-determined driving paths do not change. Another example of offline learning is found in the game Battlecuriser: 3000 AD. In this game, neural networks were trained in advance to perform route finding, threat identification, and decision making tasks for repairs and combat (Smart).Offline learning is used to create or adapt agent behavior that will behave appropriately right out of the box, and is the form of machine learning most commonly found in games (Millington, 2006).

In contrast, online learning creates or adapts agent behavior while the user is playing the game (Yannakakis & Hallam, 2005). In games equipped with online learning, the behavior of game controlled characters will change as the human player becomes more proficient or learns new tactics. There are two related goals for which online learning is used. The first is to alter behaviors (known as “performance optimizing”) such that the computer can outplay the human, creating the highest-performing behavior (Aha, Molineaux, & Ponsen, 2005). The second is performance balancing, to adapt the behavior of an agent or team of agents to the level of play of the human. Performance balancing provides a level of play that is a challenge to players but still gives them a chance to win (Andrade, Ramalho, Santana, & Corruble, 2005).

Given that the set of actions and perceptions available to an agent are fixed by the game itself, online adaptation of agent behavior is performed by controlling the state or order in which actions are selected. Laird and van Lent (2005) describe three specific types of online adaptation: learning by observation, learning by instruction, and learning from experience. In learning by observation, an agent learns to mimic the movements or tactics of the human player. The LiveCombat software (http://www.ailive.net/) provides an excellent illustration of this concept, where computer controlled agents learn to mimic complex combat maneuvers demonstrated by the human player in a matter of minutes.

This is done by developing a statistical model of what the human does in any given context and using this model to select actions. While this example builds the entire agent model from scratch, a more common approach (Manslow, 2002) is to create behavior that includes parameters and then to learn to adjust these parameters while the game is played. The goal is to create behavior where learning a small number of parameters can have a relatively large effect on the produced behavior. An example of learning a parameter by instruction is found in the game Black & White (Lionhead Studios). In this game, the behavior of an in-game avatar depends on a decision tree, where the decision tree itself is a complex parameter that is re-learned based on feedback from the human player (Fu & Houlette, 2003a). Finally, an example of learning a parameter from experience is provided by Andrade et al. (2005). The authors use feedback on how well the agent is performing in a real-time boxing game, relative to the human player, to adjust the type of actions used by the agent to make the game more or less difficult for the human player. Dynamic scripting, discussed in detail later in this dissertation, is another example of learning parameters from experience, where the parameter being learned is the relative value of each of the actions available to the agent (Spronck et al., 2006).

With the idea of creating or adapting behavior, online learning has elements of risk not present in offline learning since the game is often beyond the control of the developer when learning occurs (Yannakakis & Hallam, 2005). Specifically, when using online learning the behavior of entities will change on each individual “game machine” (e.g., computer) as the game is played. This is quite different from using offline learning to create or tweak behavior and then sending out the same static behavior to all of the instances of the game. Taking this into account, Spronck et al. (2006) list a number of computational requirements that should be present in online learning algorithms to help ensure quality game play in the face of these risks (speed, effectiveness, robustness to the randomness inherent in games, and efficiency of learning) as well as four functional requirements (clarity, variety, consistency, and scaling to player ability level). The definitions given by the author are below:

> “**Speed**: Adaptive game AI must be computationally fast, since learning takes place during game-play.

> **Effectiveness**: Adaptive game AI must be effective during the whole learning process, to avoid it becoming inferior to manually-designed game AI.
>
> **Robustness**: Adaptive game AI has to be robust with respect to the randomness in inherent in most games.
>
> **Efficiency**: Adaptive game AI must be efficient with respect to the number of learning opportunities needed to be successful, since in a single game, a player experiences only a limited number of encounters with similar situations.
>
> **Clarity**: Adaptive game AI must produce easily interpretable results, because game developers distrust learning techniques of which the results are hard to understand.
>
> **Variety**: Adaptive game AI must produce a variety of different behaviors, because agents that exhibit predictable behavior are less entertaining than agents that exhibit unpredictable behavior.
>
> **Consistency**: The average number of learning opportunities needed for adaptive game AI to produce successful results should have a high consistency, i.e., a low variance, to ensure that their achievement is independent both from the behavior of the human player, and from random fluctuations in the learning process.
>
> **Scalability**: Adaptive game AI must be able to scale the difficulty level of its results to the skill level of the human player."

There are numerous current research efforts aimed at providing or improving online learning for adaptive game AI. Much of this work makes use of reinforcement learning (RL) and hierarchical reinforcement learning (HRL) algorithms. These are reviewed in the following section.

Reinforcement Learning in Games

This section on reinforcement learning (RL) discusses three main points. Beginning with a general definition of reinforcement learning, it next discusses how a RL approach differs from familiar search-based artificial intelligence algorithms for agent control and why RL is a good candidate for use in modern computer games. Finally, several examples of reinforcement learning in research on modern computer games are presented.

Reinforcement Learning

Sutton and Barto (1998) define reinforcement learning as “learning what to do (how to map situations into actions) so as to maximize a numerical reward signal.” This broad definition allows for a diverse array of reinforcement algorithms, generally sharing the same features. First, an agent has perceptions that place it in state *s,* where the state s is defined by the features of the current game or simulation state. While in state *s,* the agent must choose and perform an action *a* from the set of all actions available in this state A_s. The policy, p, is the mapping between states and actions. The policy describes the current behavior of the agent by describing what actions it will take in a given state, where p(s, a) gives the probability of choosing action *a* in state *s*. A reward function, *R*(s,a) is provides feedback for performing action a in state s, based on the desirability of the outcome. The reward function encapsulates the goal of the learner, e.g. escaping the maze as quickly as possible.

Q-Learning

The class of reinforcement learning algorithms called Q-learners use the reward function to learn the expected values of actions. Specifically, the value function *Q* is used to predict the immediate and future expected reward of taking action *a* in state *s, Q*(s, a) (Tadepalli, Givan, & Driessens, 2004). The value function provides a long-term prediction for selecting a given action in the current state and following the policy p thereafter. That is, Q-learners learn action values, not a policy. A standard value update function for Q-learning is

$$Q(s,a) = Q(s,a) + a\left[r + g\max_{a'} Q(s',a') - Q(s,a)\right]$$

where α is the learning rate and γ the discount rate (Sutton & Barto, 1998). Another popular Q-learning update function is Sarsa

$$Q(s,a) = Q(s,a) + a\left[r + gQ(s',a')\right]$$

which makes use of the value of the decision actually taken in the next step rather than the maximum value available in the next step (Sutton & Barto, 1998).

Table 1 Example Q-table.

	a1	a2
s1	10	1
s2	7	8

Table 1 provides an example of a Q-table, where the two states are s1 and s2 and the two actions are a1 and a2. Let a1 by the action selected in s1, with a resulting state of s2 and a reward of -3. The resulting update is:

$$Q(s1, a1) = Q(s1, a1) + a\left[-3 + g\,(s2, a2) - Q(s1, a1)\right]$$

Using the Q-table values from the table, and with $\alpha = 0.1$ and $\gamma = 0.8$, we have the following:

$$Q(s1, a1) = 10 + 0.1\left[-3 + 0.8(8) - 10\right]$$

where the new estimate of Q(s1,a1) is set to be 9.34.

As defined previously, a policy p describes the current behavior of the agent, where p(*s*, *a*) gives the probability of choosing action *a* in state *s*. Q-learning and Sarsa are both value function approaches that make use of a fixed policy combined with learned action values. In these algorithms, the policy p (*s*, *a*) is generated from the action value estimates. For example, the Q-value is an estimate of the long-term expected value of action *a* in state *s* and is learned from experience. The policy p (*s*, *a*) is calculated by a fixed algorithm (such as softmax) from the Q-values. That is, the fixed policy of an agent using Q-learning is to use the estimated values of actions to select (proportionately) the action with the highest value in the state where the decision is required.

Policy approaches differ from value function approaches in that the policy of the agent is learned directly. The agent searches through the state space of policies, rather than through the state of estimated values, for one that maximizes the received reward

(Sutton, McAllester, Singh, & Mansour, 2000). The example given by the authors is a policy represented as a neural network. The input to the neural network is the current game state and the output of the network is the policy p(*s*, *a*). What are learned in this example are the connection weights within the neural network. These weights are a set of parameters that are turned directly by a reinforcement learning algorithm based on feedback received from the game.

Hierarchical Reinforcement Learning

Hierarchal reinforcement learning (HRL) is a form of reinforcement learning where the set of available actions, A_s, is extended to include actions that can invoke other actions (Barto & Mahadevan, 2003). Different types of algorithms have different methods of specifying tasks but they share much in common conceptually. HRL methods divide a single learning problem into a number of sub-problems, where each sub-problem has its own policy. In the cases where this policy is learned, each sub-problem can be considered a distinct Q-learning task that may contain a separate reward function and a separate set of action values. An example in a robotic control domain is the open-the-door subtask, which is reinforced by a reward function that provides feedback on how the open-the-door task is proceeding. This sub-problem is only concerned with opening the door and is invoked by a parent learning problem that is attempting to navigate from one room to another. The parent problem has a separate reward function and set of action values.

Learning to complete a subtask can hinder the ability to complete the overall task (Dietterich, 2000). For example, the robot might be able to open the door slowly while conserving power or open the door quickly while expending a lot of power. If the reward function only focuses on opening the door quickly, the latter will be chosen. This expenditure of power may make it more difficult for the robot to complete the overall navigation task. To avoid the problem of local optimality in learning how to perform subtasks, reward policies must be carefully designed (Dietterich, 2000).

HRL algorithms also require some additional specification of the set of states used by the learning algorithm. Some approaches take into account both the local (subtask) and global (game) state when learning subtask action values (Barto & Mahadevan, 2003). In this case, the Q-values take the form of *Q([s,m],a)* where *s* represents the global state and *m* represents the local state. For example, the open-the-door subtask might contain a local variable for number of attempts to grasp the doorknob. A different approach, taken by the MAXQ hierarchical reinforcement learning algorithm, considers only the state information relative to a subtask (Dietterich, 2000). This form of state abstraction reduces the size of *S* for each subtask, which improves the performance of the reinforcement learning algorithm for each subtask as there are fewer states for which Q-values must be learned.

Comparing Q-Learning to Search-based Solutions

The minmax family of algorithms are familiar heuristic search algorithms for controlling the behavior of entities in more traditional games, such as tic-tac-toe or backgammon (Russell & Norvig, 1995). The minmax algorithm selects an action to perform from a given state by searching for an action with the highest utility value (max). The utility value is determined by looking at the resulting state of an action and determine ing the best move the opponent can make in that state (min). This search can be carried out to an arbitrary depth, alternating max and min decisions, as shown in Figure 5.

The expectimax algorithm (Russell & Norvig, 1995) adds a *chance* node to the minmax algorithm, so that games with a random element (such as dice rolls), can be supported. Both algorithms have the same requirements. First, there must be a utility function that can judge the values of the resulting states. Second, there must be a domain model that allows the agent to predict the state that would result after performing an action (which may be a state distribution in the expectimax algorithm).

There are some similarities between Q-learning and search-based solutions. Q-learning has the same goal as the epectimax algorithm – to select the best action given the current game state. Additionally, after an action is selected, a reward (utility) function provides feedback on the value of the new state. However, there are also some striking differences. First, rather than searching for the best utility, the Q-learning algorithm makes use of cached action utility estimates which are constantly updated. This is an important consideration in modern computer games, as the complexity of expectimax is $O(b^d n^d)$, where *b* is the branching factor (number of actions available), *n* is the number of possibilities from a *chance* node, and *d* is the depth of the search tree.

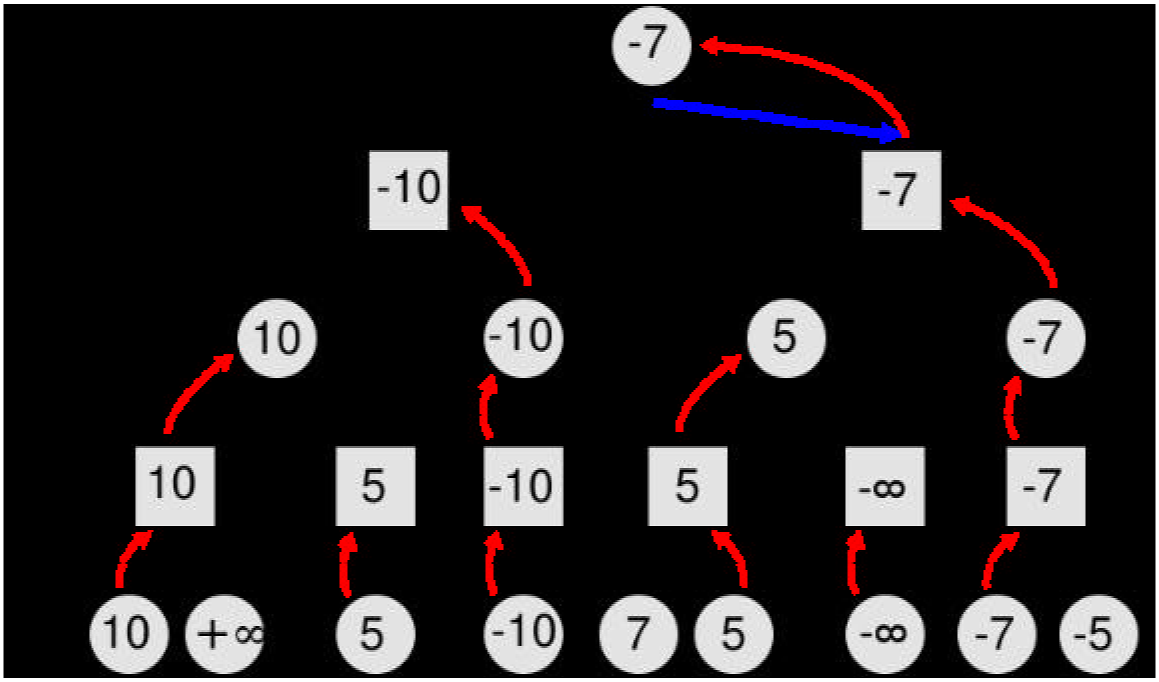


Figure 5 Graphical depiction of the minmax algorithm. The circle at the top is the current state of the player making the decision. This player chooses the action (line) that maximizes utility. The squares represent the predicted action of the opposing player, which will minimize utility. The upward arrows indicate the action selected by the minimizing or maximizing player. The single downward arrow shows the action actually selected by the deciding player (Nogueira, 2006).

Second, reinforcement learning does not require a domain model. This is especially important in modern computer games where the underlying model may be inaccessible or incapable of hypothesizing all of the resulting states quickly enough to support a search algorithm. Third, as pointed out by Sutton and Barto (1998), reinforcement learning techniques learn to play against an actual opponent instead of against a theoretical opponent as found in expectimax. It is these three differences – action utility estimates, lack of domain model, and learning via an actual opponent – that make reinforcement learning techniques especially appealing in modern games and simulations.

Examples of RL in Modern Computer Games

A survey of the 2005 workshop on Reasoning, Representation, and Learning in Computer Games reveals a research community that is very interested in reinforcement learning algorithms in interactive computer games. Andrade et al. use reinforcement learning techniques to provide game balancing in a real-time boxing game (2005): offline RL algorithms are used to learn optimal strategies for the computer fighter and an online selection mechanism is used to choose actions at runtime in order to balance the game. Bakkes et al., (2005) describe an online learning mechanism called TEAM2 (Team-oriented Exploitative Adaptability Mechanism 2) designed to learn team-oriented behavior based on a best-response RL algorithm. TEAM2 selects the best team configuration for the state of the game; each player is assigned to be offensive, defensive, or roaming. Values are assigned to team configurations on state transitions (i.e., the current team configuration resulted in better / worse performance). Maclin et al., (2005) use the Knowedge-Based Kernel Regression (KBKR) technique to increase the performance of RL in a RoboCup soccer simulator. Their algorithm uses advice given by a behavior author (programmer) in the form of IF-THEN rules to set the Q-value for an action in a particular state (or set of states) to be high or low relative to the average Q-value. Marthi et al, (2005) employ a hierarchical RL algorithm to solve a particular scenario in a RTS game, discussed in more detail (below). Ulam et al. (2005) use model-based reflection to determine in which states learning should occur. This is done by dividing agent behavior learning into a number of smaller, distinct learning problems using a form of hierarchical RL. Model based reflection is used to determine which task is ultimately failing and then RL-based learning is performed only for that task. This is a knowledge-intensive mechanism as the model-based mechanism requires significant descriptions of how each task can fail.

Dynamic Scripting

Dynamic scripting (DS) is one example of an online reinforcement learning algorithm developed specifically for games (Spronck et al., 2006). The Rapid Online Learning for Entertaining Computing (ROLEC; http://rolec.unimaas.nl/) project has made significant contributions regarding RL in Game AI. One product of their research is dynamic scripting, a reinforcement learning algorithm designed for game AI. ROLEC tested their software in both role playing games (RPG), such as Neverwinter Nights, and real-time strategy games (RTS) such as Stratagus. They also examined hierarchical reinforcement learning in games using a dynamic scripting reward function (not the dynamic scripting algorithm) (Ponsen, Spronck, & Tuyls, 2006) and extended dynamic scripting to work with case-based selection (Aha et al., 2005). Dynamic scripting has also been extended to add a goal-directed component by a separate research group (Dahlbom & Niklasson, 2006).

At a high level, agents controlled by the dynamic scripting algorithm have much in common with the other behavior models discussed previously: finite state machines, cognitive architectures, Q-learning, and epectimax. In each case, the agent is responsible for selecting an action to perform in the current game state. However, dynamic scripting is most similar to Q-learning in that the goal is to learn the relative values of the available actions.

Before diving into the details of dynamic scripting, it is useful to point out a primary difference between dynamic scripting and standard Q-learning algorithms. In Q-learning algorithms, the number of states in a game increases exponentially as the number of features in a game increases. The result is that in a modern computer game with a complex environment, the number of game states is generally too large to efficiently learn adaptive agent behavior. There are a number of ways to reduce this problem and the dynamic scripting algorithm does so by selecting actions without regard to any features of the current game state. That is, only a single abstract game state is considered. Thus while a Q-learning algorithm learns the value of an action in a particular state, dynamic scripting learns the value of an action.

There are four main components to the dynamic scripting algorithm: a set of actions, script selection, action policy, and action value updating (Spronck et al., 2006). The first component is a set of actions that the algorithm can choose from. Each action may optionally contain an IF clause that limits its applicability based on the current game state, much like a production rule. For example, `IF health < 50% THEN useHealingPotion` could be one action and `useHealingPotion` with no IF clause another action. In the case of dynamic scripting, it is assumed that the person authoring the game behavior is responsible for creating the set of actions, though previous work has focused on automatically creating these types of actions / rules (e.g. Khardon, 1999). Each individual action in the set of actions has a single value associated with it.

The second component of the algorithm is script selection. Before each episode the agent creates a subset of the available actions to use in the episode – this is known as a script. A free parameter *n* determines the size of the script. The script selection component uses a form of fitness proportionate selection to select *n* actions (without replacement) from the complete set of actions based on their assigned value. Action *a* is selected for inclusion in the script with probability *p* given by the softmax algorithm:

$$p = \frac{e^{V(a)/t}}{\sum_{b=1}^{n} e^{V(b)/t}}$$

,where *V(a)* is the current value of action *a* and τ is a temperature parameter (Sutton & Barto, 1998). The temperature parameter adjusts the exploitation / exploration character of action selection, where a higher temperature leads to more exploration by giving less weight to differences in action values and a lower temperature leads to more exploitation by giving more weight to differences in action values. Script selection is the primary use of action values in the dynamic scripting algorithm. This contrasts with Q-learning algorithms, which would select an action from the full set of actions for each decision.

The third component is the action policy, which determines how actions are selected within an episode. This component walks through the script in order and performs the first action that is applicable to the current game state. For example, an action may require that a character’s health be below 50%. If this is not the case then the action does not apply. Actions are ordered by their priority. This is generally assigned by the behavior author, though there is some research on learning action priorities in dynamic scripting (Timuri, Spronck, & van den Herik, 2007) and in production rule scripts in general (e.g. Khardon, 1999). In the event of a priority tie, actions are selected based on the highest action value. This is the secondary use of action values in the dynamic scripting algorithm.

Action value updating is the fourth component of dynamic scripting. The behavior author creates a reward function that provides feedback on the utility of the script as a whole. High rewards indicate strong performance and low rewards indicate low performance. At the end of the episode, this reward function is used to create a single numeric reward for the agent's behavior. The full reward is given to each action in the script that was successfully performed during the encounter. A half reward is given to each action in the script that was not selected, which can happen because the rule was never applicable or because the rule had a relatively low priority. Compensation is applied to all actions that are not part of the script. Through the compensation mechanism, the action value updating component is responsible for distributing the action value "points" among the available actions. For example, if there are 10 actions with an initial value of 100, there are 1000 action points that can be distributed across all actions. Given a list of completed actions, *completed,* a list of actions in the script that were not completed, *incomplete*, and a list of actions not in the script, *notInScript*, the function for compensation is:

$$compensation = -\left(\frac{|completed| * r + |incomplete| * r / 2}{|notInScript|} \right)$$

where the *payout* for an action might be *r*, *r/2*, or *compensation*, the actual value updating algorithm is the same in all cases:

$$V(a) = V(a) + payout$$

Two additional free parameters are applied to all action values: minimum and maximum. The minimum value (e.g. 0) is the lowest action value that an action can have, while the maximum value (e.g. 1000) is the highest allowed value for an action. A remainder distribution method is used to ensure that the sum of the action values remains constant by redistributing the over / underflow caused by value adjustments.

Within the framework of reinforcement learning, the dynamic scripting architecture is closely related to the actor-critic architecture defined by Sutton and Barto (1998). The general model is shown in Figure 6. In dynamic scripting, the script performs the role of the actor and the critic is composed of the action values and the value update function (Spronck et al., 2006). The two architectures differ in that dynamic scripting uses a script as a policy rather than updating a policy directly based on the feedback. That is, dynamic scripting learns a policy directly rather than learning values for actions in particular states. By updating the parameters that affect script generation the agent is searching through the space of available scripts, where each script is a policy. This is seen in the right-hand figure where policy and value updates are separated, instead of joined as in the left-hand figure.

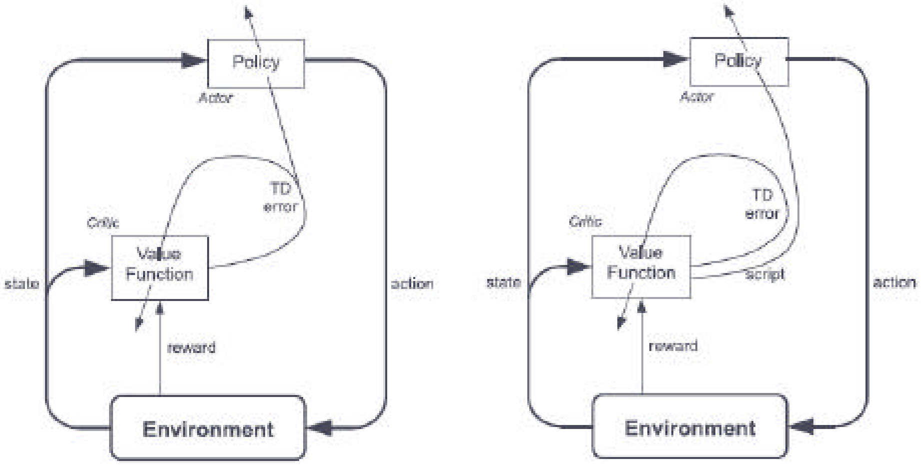


Figure 6 Actor-critic architecture (left) (Sutton & Barto, 1998) compared to dynamic scripting (right) (Spronck et al., 2006).

The following side by side comparison of the algorithms for Q-learning and for dynamic scripting summarizes their differences:

Dynamic Scripting

- Actions have a value *V(s, a)*, where the set of states, *S*, contains only a single abstract state.
- Action values are used to create a script prior to an episode.
- Actions are selected in priority order first, action value second, from the script.
- At the end of episode, action values are updated using dynamic scripting updated function.

Q-Learning

- Actions have a value *Q(s,a)*, where the set of states, *S*, can contain actual or abstract game states.
- Actions are selected during an episode with value proportionate selection, based on their Q-values.
- After each action is performed, Q-values are updated using the Q-learning update function.

Dynamic scripting was designed specifically for modern computer games where significant domain knowledge exists that can be embedded in the behavior model through the construction of actions and priority assignment. Specifically, there should exist some actions that intuitively can be assigned a high priority and that contain a more specialized IF clause. That is, there should be some rules that the behavior author can identify as important in limited situations, e.g. using a *heal* action when a character is low on health. Second, there also needs to be some actions given lower priority which are more generally applicable. Dynamic scripting depends on these two sources of domain knowledge to overcome its limited reliance on the current game state. Additionally, dynamic scripting is not designed for games or decisions where the correct action depends heavily on the current game state. An example of this is simple path planning, where the agent determines how to move through a grid-world to reach a pre-determined goal. That is, in any given game state the agent chooses to move North, East, West, or South. While Q-learning will perform well on this type of problem, dynamic scripting will be unable to create a script capable of solving the navigation with only these four actions.

CHAPTER III
EXTENDED DYNAMIC SCRIPTING

This dissertation proposes three specific extensions to the dynamic scripting algorithm that aim to improve learning behavior and flexibility while imposing a minimal cost. The extensions are: (1) developing a flexible, stand alone, version of dynamic scripting that allows for **hierarchical dynamic scripting**, (2) extending the **context sensitivity** of hierarchical dynamic scripting through abstract state construction, and (3) performing an **architecture integration** where this algorithm is integrated with an existing hierarchical behavior modeling architecture. The result of these changes will be referred to as Extended Dynamic Scripting (EDS). This research begins with extensions that have worked well in the context of standard Q-learning algorithms and recasts them to work in the context of dynamic scripting and complex computer games. The motivation behind each of these extensions and their implementation details are discussed (below). The following list previews the extensions to the dynamic scripting algorithm, where the changes are noted by sub-bullets.

EDS

- Actions have a value *Q(s, a),* where the set of states, *S*, contains only a single abstract state.
    - ➢ **Hierarchical DS:** Choice points are added to the library so support task decomposition, dividing the learning problem into a hierarchy of sub-problems.
    - ➢ **Context Sensitivity:** The set of states, *S*, is expanded through both manual and automatic state abstraction. The value of an action, *V(s,a),* now includes a state component that represents the branch taken in the manually or automatically constructed hierarchy.
- Action values are used to create scripts prior to an episode.
    - ➢ **Hierarchical DS:** Reward points allow for varying episode lengths, where episodes can contain 1…*n* actions, for each individual choice point.

- Actions are selected in priority order first, action value second, from the script.
    - **Architecture Integration:** This is altered to work in the context of a behavior modeling architecture.
- At the end of each episode, actions are updated using dynamic scripting value update function.
    - **Hierarchical DS:** The value update function is changed to support a wider variety of reward configurations.
    - **Architecture Integration:** Game specific reward objects are created by the behavior author that plug-in to the EDS algorithm.

Hierarchical Dynamic Scripting

The first enhancement is to design a dynamic scripting-based architecture capable of representing multiple decision points in a hierarchical fashion. This involves three specific design extensions: (1) choice points, (2) reward points, and (3) scaled value updating.

Choice Points

A choice point defines a distinct learning problem, where the agent makes a decision by choosing a single action to perform from some number of available actions using the dynamic scripting algorithm. The value of choice points is that an agent can use them to break a larger learning problem down into a number of sub-problems in a hierarchical fashion and then learn separately how to accomplish each subtask (Dahlbom & Niklasson, 2006). Choice points for task decomposition, as described in this section, are similar to the choice points found in the Hierarchy of Abstract Machines and ALisp architectures (Andre & Russell, 2002) and goals in the GoHDS algorithm (Dahlbom & Niklasson, 2006).

As an example, an agent may be capable of carrying out four attack actions: *knockdown, melee, ranged,* and *sneak.* A choice point can be used to learn the best script for completing the attack action. A graphical description of this choice point is shown in the behavior Figure 7. The choice point is indicated by the *choose(…)* action, where the possible choices are the directed connectors leading out of the choose node to the rectangles (actions). The oval (predicate) *canKnockdown* preceding the *knockdown* action represents the IF clause of the *knockdown* action.

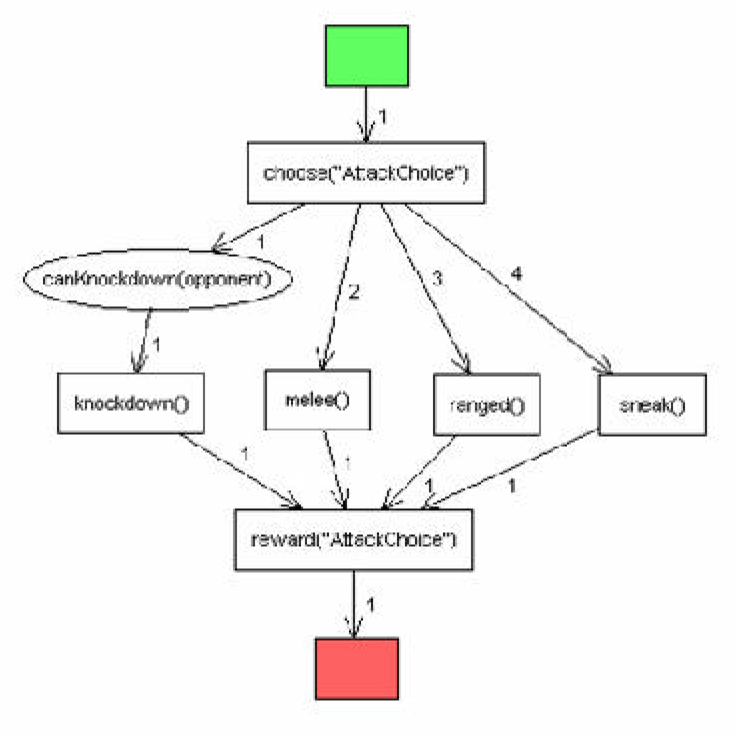


Figure 7 AttackChoice choice point. This choice point chooses between on of the four available actions at each decision point.

This choice point is a distinct instance of the dynamic scripting algorithm. To support this, the choice point also contains the data shown in Table 2 . The information for each possible selection includes the value of the action, its priority, and whether it is in the current dynamically generated script (indicated by the *).

Table 2 Data for the AttackChoice choice point.

Action	Value	Priority	Script
1 (knockdown)	112	High	*
2 (melee)	88	Low	*
3 (ranged)	50	Low	
4 (sneak)	117	Med	

The dynamically generated script in this case based on the action priorities and the action values is:

1. IF canKnockdown(opponent) THEN knockdown
2. ELSE melee

Choice points can be combined in a hierarchical fashion in order to decompose complex tasks. An example of this is shown in Figure 8, where the top-level choice point, *ResponseChoice*, is used to choose a subtask at the top level of the hierarchy. In this case, the choice point learns whether *AttackChoice* or *DefendChoice* is more likely to generate a higher reward. The agent learns the utility of attacking vs. defending separately from how to carry out either of these. The lower level choice point *AttackChoice* learns which of the four primitive attacks are likely to generate a higher reward. Likewise, the subtask *DefendChoice* learns which of the primitive defenses are likely to generate a higher reward. These two subtasks are actually responsible for learning how to carry out an attack or defense.

Within Figure 8, the flow of control starts in the top, right-hand, shaded node. The directed connector is followed to the *choose* action, which transfers control to either the *Attack* or *Defend* sub-behaviors. Once the sub-behavior is finished, control transfers back to the parent behavior and a transition is made to the empty action node. From here, the transition labeled “1” is tried first. If the *episodeOver* predicate returns true, then a transition occurs to the *reward* action and then back to the *choose* action. If the predicate is false, the transition labeled “2” is taken, which goes directly to the *choose* action.

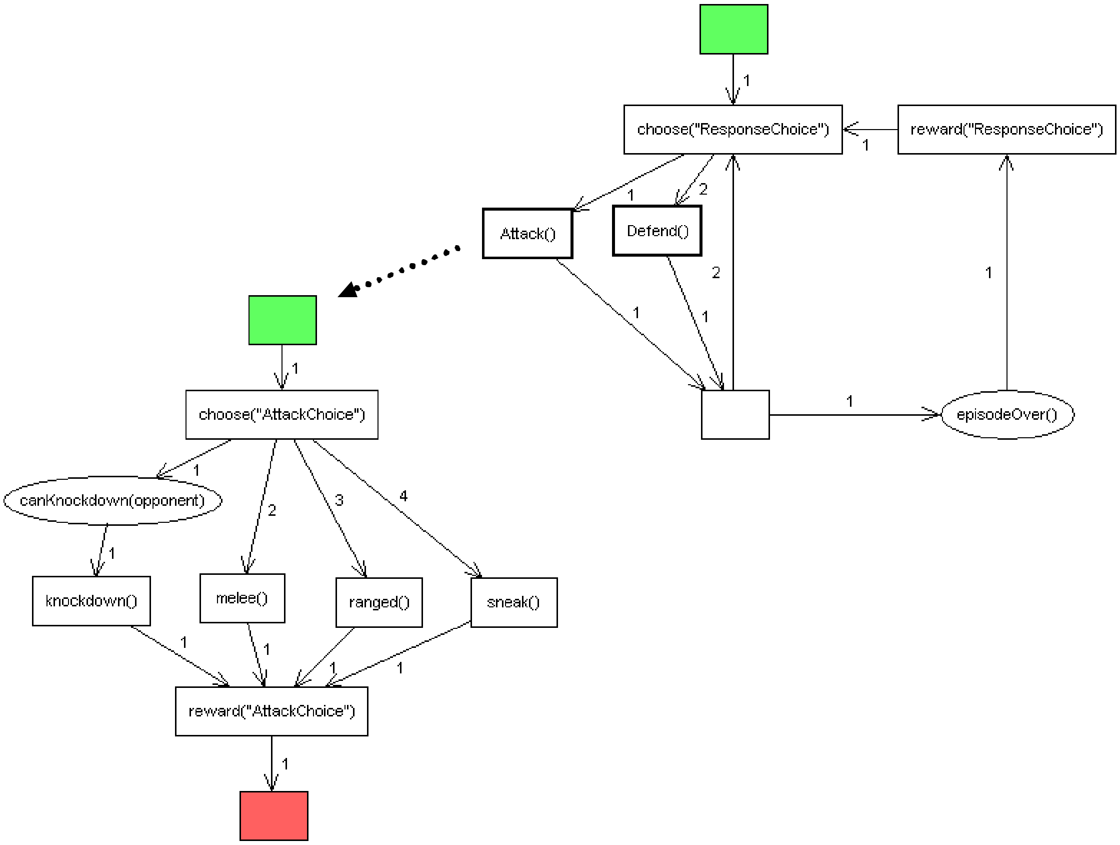


Figure 8 ResponseChoice with AttackChoice as a subtask.

Reward Points

There are situations in agent behavior in modern computer games where immediate learning can have a significant impact. For example, an EDS agent is creating an attack script that contains two of four primitive attack actions. After performing an action, feedback on its success can be immediately taken from the game state and used to adjust the likelihood of choosing the same attack type in the next step in the same episode. A flexible behavior modeling tool needs to support immediate rewards in addition to the episodic rewards supported by dynamic scripting.

To support rewards at arbitrary points in the behavior, each choice point also has a corresponding reward point that updates the values of its actions. Examining the previous example, the *AttackChoice* point can be updated immediately, based on the amount of damage caused by executing one of the attacks. The top level choice point

ResponseChoice that chooses between the Attack and Defend subtasks might only be updated at the end of the episode. In either case, after some number of action selections by a choice point its reward point is reached. The reward point then updates the values of all of the actions in the corresponding choice point, based on the dynamic scripting value update function. These reward points are included in the graphical behavior descriptions, Figure 7 and Figure 8 (above). The choice points are represented by the *choose* actions and the reward point by the *reward* actions. Figure 7 illustrates an immediate reward for *AttackChoice*, which creates a script, selects a primitive action, and then immediately transitions to a *reward* action. Figure 8 illustrates an episodic reward for *ResponseChoice*, which chooses from either Attack or Defend subtaks. The corresponding reward point is activated whenever the predicate *episodeOver* (oval) returns true. This is an example of an episodic reward.

Scaled Value Updates

In the existing value updating mechanism, each action receives the same reward no matter how often it is used. For long episodes with a small number of actions, it is likely that all the actions will be selected in the episode. If episodic learning is used in this case, the resulting reward will then change all of the action values in the same way and learning will only occur very slowly. This is an example of a credit-assignment problem (Minsky, 1961), where a reward must be appropriately divided among an extended sequence of actions for optimal learning. Sutton and Barto (1998) note that reinforcement learning algorithms are “in a sense, directed toward solving this problem.”

To address this particular credit-assignment problem, a new method of scaling the value updates was developed. We calculate the total amount of reward “points” to be given for all the different actions and then divide this by the number of times an action was performed to determine an action specific reward. For example, assume the reward is equal to 50. Under the non-scaling value update, if action A was completed 9 times and action B completed 1 time they would each receive the same value update of 50. However, with value scaling A would receive 90 and B would receive 10.

In both update algorithms, all actions in the script that have been completed at least once form the *completed* set. All actions in the script that are not completed are part of the *incomplete* set. All actions that were not part of the script are in the *notInScript* set. In the original value update algorithm, actions received a full reward, *r*, a half reward, *r/2*, or compensation, *c*, based on membership in one of the three sets. The algorithm for the new update mechanism is:

1. (Sum reward.) Set *rewardSum* $\leftarrow$ r $\times$ |*completed*| + (r/2) $\times$ |*incomplete*|
2. (Sum actions). Set *actionSum* $\leftarrow (\sum_{j=1}^{n} timesCompleted(a_j)) + |incomplete|$
3. (Set reward per completion.) Set *slice* $\leftarrow$ *rewardSum / actionSum*
4. (Reward completed actions.) For each *a* in *completed*
    a. Set $completions_a \leftarrow$ *timesCompleted(a)*
    b. Set $reward_a \leftarrow completions_a \times slice$
5. (Reward incomplete actions.) For each *a* in *incomplete,* set $reward_a \leftarrow slice$

Following the assignment of rewards to actions, the value update mechanism is carried out normally. This includes applying compensation to the actions in the *notInScript* set as well as performing remainder distribution for under / over flow based on the minimum and maximum action values.

Context Sensitivity

The second enhancement to dynamic scripting to be considered improves the ability of the algorithm to take the game context into account when making decisions. Following background information on the problem of state abstraction this section describes two methods of creating abstract states, the first performed manually by behavior authors and the second performed automatically by learning algorithms. The third part of this section covers the implementation details of the automatic method of state abstraction.

Background

Reinforcement learning algorithms, such as Q-learning, are a valuable tool for adapting agent behavior in computer games. However, the number of states in a game increases exponentially as the number features in a game increases. The result is that in a modern computer game with a complex environment the number of game states is generally too large to efficiently learn to adapt agent behavior with standard Q-learning. To address this problem, it is important to determine which game features can be ignored in order to reduce the number of game states. The dynamic scripting algorithm addresses this problem by selecting actions without regard to any features of the current game state. That is, only a single abstract game state is considered. This algorithm results in very efficient learning consistent with online learning requirements, but misses the opportunity to improve performance by taking into account some aspects of the game state. The research described in this section is aimed at specifying additional abstract states which grow an abstract state tree, either manually or automatically, from the single state currently considered in dynamic scripting towards the full set of leaf nodes considered by standard Q-learning algorithms.

The second extension to dynamic scripting improves the ability of the algorithm to take game context into account when making decisions. The motivation behind this extension is that the context (i.e. game state) in which an action is performed can have an effect on its outcome. For example, in the role playing game NeverWinter Nights encounters with enemies are generally designed around the level of the player. The player might encounter one enemy of equal strength, a few enemies that are slightly weaker, or a large number of significantly weaker enemies. Area effect spells are much more effective on groups of weaker creatures, so rules that select area-based spells should be more likely to be selected when facing the latter. On the other hand, individual effect spells tend to be more effective when applied to a single, more powerful, enemy.

State Specialization

There are two different approaches that identify the important features of the game, i.e. identify abstract game states, for a choice point. The first makes use of author provided information and the second learns to make state distinctions based on the performance of the learning algorithm.

The first method of state abstraction makes use of knowledge supplied by the behavior author, either directly as part of the behavior specification or indirectly as a supplied model. For example, a rule states that the size of the opposing party is an important feature and a new action value set (another state space) should be generated for that case. This is similar to what Marthi et al. (2005) does by giving feature set descriptions of the important features for the choice points in the Stratagus playing AI. This is also what Ponsen & Spronck (2004) do, where they create abstract states based on the available buildings in Stratagus. In either case, information is supplied by the programmer that can be used to determine when a new abstract game state would be useful for a particular learning task.

An example of an author defined abstract state is shown in Figure 9. This behavior utilizes a predicate (getNumberEnemies() == 1) to divide the problem of learning which type of spell to cast into two distinct sub-problems – one for when there is only a single enemy and another for when there are multiple enemies. In this behavior there are two separate choice points, *OneEnemySpell* and *MultEnemySpell*, where each choice point has its own set of choice point data as shown in Table 3 and Table 4.

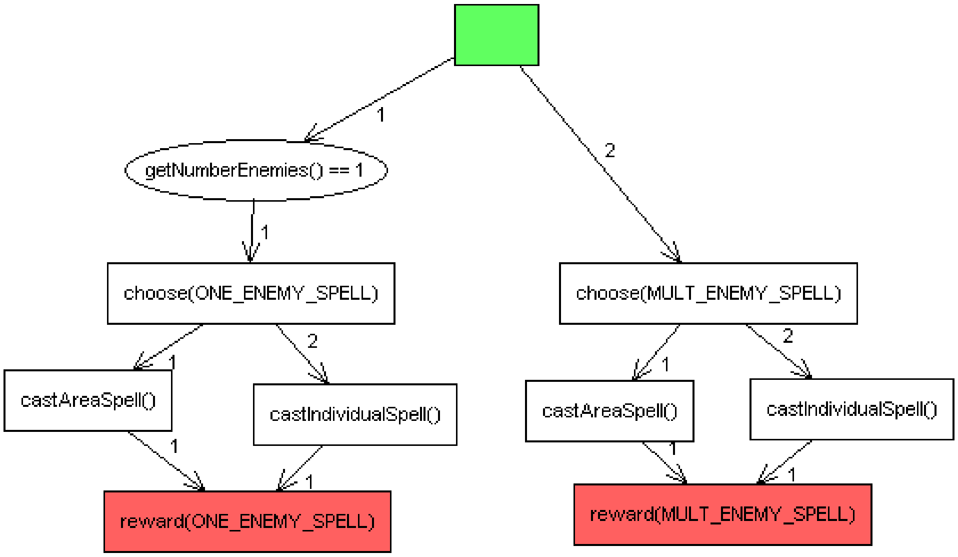


Figure 9 An author defined state abstraction, where the learning problem is divided into two choice points: single enemies and multiple enemies.

Table 3 Choice point data for ONE_ENEMY_SPELL.

Action	Value	Priority	Script
1 (area)	88	Med	
2 (individual)	112	Med	*

Table 4 Choice point data for MULT_ENEMY_SPELL.

Action	Value	Priority	Script
1 (area)	150	High	*
2 (individual)	50	Low	

The same abstract state distinction could also be automatically learned for a choice point by a meta-level system without requiring any additional information from the user. This meta-level system could employ a machine learning algorithm to automatically learn when a distinct set of action values is called for based on the behavior of the learning algorithm. In our case, a meta-system would look for patterns in the past performance of the reinforcement learning algorithm using knowledge discovery / data mining methods. These patterns are used to create abstract game states, where each abstract state contains its own set of action values and its own dynamic script.

An example of an automatically defined abstract state is shown in Figure 10 and Table 5. This behavior, and the associated choice point data, illustrate how the meta-level system can be used to create multiple sets of action values, each with its own dynamically generated script, for a single choice point. In this particular instance, the meta-level system creates two distinct sets of action values for a single choice point – one for when there is only a single enemy and another for when there are multiple enemies. That is, if the choice point is facing a single opponent and selects an action, it should be selected from the script derived from the Enemies = 1 action value set (and not the Enemies > 1 action value set).

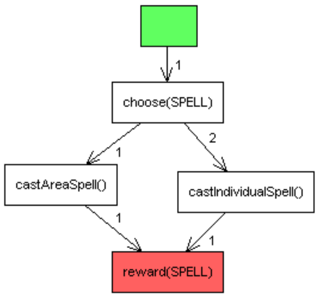


Figure 10 A choice point that makes use of automatic state abstraction.

Table 5 An automatically learned state abstraction, where the learning problem addressed by a single choice point is divided into two learning problems, single enemies and multiple enemies, by making use of different dynamic scripting data for the same choice point.

Enemies = 1				Enemies > 1			
Action	Value	Priority	Script	Action	Value	Priority	Script
1 (area)	88	Med		1 (area)	150	Med	*
2 (individual)	112	Med	*	2 (individual)	50	Med	

Both of these methods of state abstraction are supported in EDS. The first form of game state abstraction can be performed by behavior authors as they construct hierarchal behaviors with choice points in the modeling architecture. EDS also extends the hierarchical dynamic scripting algorithm to support the automatic construction of abstract game states as described below.

Automatic State Specialization

EDS extends the hierarchical dynamic scripting algorithm to support the automatic construction of game state abstraction, building on the success of decision-tree based algorithms that have been successfully applied to Q-learning problems (see Chapman & Kaelbling, 1991; McCallum, 1996). Below is a discussion of the process of creating and using abstract state trees.

Creating Abstract State Trees

Our mechanism for automatic state specialization involves three steps: (1) collecting a series of training instances, (2) building a classification tree with the given reward as the target attribute, and (3) using the training instances to develop action values for each leaf node. We build upon the Weka (Witten & Frank, 2005) data mining architecture to complete each of the steps. The algorithm for this is:

```
1. (Compile decisions.)
    a. Set $Instances$ ← empty set
    b. For each choice point decision
        i. Set $instance$ ← game state information at time of decision
```

```
            ii. Set Instances ← Instances + instance
2. (Build classifier.)
    a. (Clean data.) Set Instances ← cleanData(Instances)
    b. (Automatically divide the real-valued reward attribute into 5 bins using
       Weka methods if necessary)
       Set Instances ← binData(Instances, 5)
    c. (Create a Weka classifier with the reward attribute as the target.)
       Set classifier ← buildClassifier(Instances, “Reward”)
3. (Update action values.)
    a. For each distinct episodeIndex ∈ Instances
         i. (Get all of the episodes that were rewarded in a single episode.)
            Set EpisodeInstances ← getInstances(Instances, episodeIndex)
        ii. For each instance ∈ EpisodeInstances
             1. (Classify the instance according to the newly created tree.)
                Set index ← classifier.classify(instance)
             2. (Place the instance in a list with other instances from the
                same episode.)
                Set LeafInstances[index] ← LeafInstances[index] + instance
       iii. For each index ∈ LeafInstances
             1. (Choose the largest absolute reward of all of the matching
                instances.)
                Set reward ← getLargestReward (LeafInstances[index])
             2. (Get the action values for this leaf node.)
                Set ActionValues ← getActionValueSet(index)
             3. (Apply the reward from all of the instances that match a
                particular episode and leaf node AS IF they had been in a
                single script.)
                Set ActionValues ← applyReward(ActionValues,
                LeafInstances[index], reward)
```

For step two, building a classifier, we rely on two specific algorithms for building tree-based classifiers: Decision Stump and J48. Both of these algorithms are described by Witten and Frank (2005) in the Weka documentation. A decision stump is a limited tree that makes a binary classification, resulting in three leaves – two for the binary classification of the attribute and one in case the value is missing. The decision stump works with numeric target attributes, such as action rewards, so no additional processing is needed. The J48 algorithm builds an arbitrary decision tree, but has a limitation in that this algorithm requires that the target attribute (reward) be non-numeric (e.g. categorical). J48 requires the optional binning step as described in the pseudo-code.

In step three, updating the action values, the algorithm results are an approximation of what the action values would look like had the classifier been in use the entire time. This is because each leaf node contains its own script and we don't know what the script would have been if the classifier had been in use the entire time. As an approximation, it is expected that learning performance might temporarily decrease after a classifier is first created.

Using Abstract State Trees

Once a classification tree is created, the process of generating scripts and selecting actions are both modified slightly. First, where previously the choice point would generate a single script, now a script will be created for each leaf node in the tree. This takes advantage of the different action value sets contained in each leaf node. Second, when an action is to be selected the current state information is gathered into an *instance*. Based on the trees mapping of an *instance* to a leaf node, the action is selected from the appropriate leaf node script.

Architecture Integration

Extended dynamic scripting as described can readily be integrated with a hierarchical task network agent architecture. Task networks are a standard and straightforward way of representing agent behavior in games (Fu & Houlette, 2003b). Task networks generally consist of actions, conditions, and connectors that describe agent behavior. These conditions, actions, and connectors encompass the traditional representation elements of finite state machines (a set of states, *S*, a set of inputs, *I*, and a transition function from one state to another, *T(s,i)*) where the states represent the state of the behavior, not the state of the game. A directed graph of actions, conditions, and connectors is known as a behavior. Within a behavior, the actions and conditions can reference other behaviors to form hierarchical task networks.

Task networks may also support additional features such as local and global variables, ordered transitions, blackboard communication, debugging, exception handling, and others, but they tend not to support more deliberative functions such as planning and learning (Ludwig & Houlette, 2006). Some exceptions to this are the MicroSaint task network environment, which has been augmented to support case-based learning (Warwick & Santamaria, 2006) and the previously mentioned ALisp that supports hierarchal reinforcement learning (Marthi et al., 2005).

As part of this dissertation, an existing commercial task network architecture is extended to include extended dynamic scripting nodes. The SimBionic hierarchical task network programming environment (Fu & Houlette, 2002; Fu, Houlette, & Ludwig, 2007) was selected as the commercial AI middleware to integrate with hierarchical dynamic scripting. This decision was made based on the similarity of feature sets in this application to other solutions and on the availability of and familiarity with the SimBionic implementation. The core of SimBionic is a visual authoring tool that allows users to draw flow chart-like diagrams that specify sequences of conditions and actions.

The integration was accomplished by adding dynamic scripting choice points and reward points to the graphical task network programming language. A dynamic scripting choice point action is added when the behavior can choose between two or more actions, indicated by the *choose* keyword. The *choose* keyword will also contain the name of the behavior state, which is used to retrieve the appropriate set of action values. The softmax algorithm, a kind of fitness proportionate selection mechanism, uses the action values to dynamically select the script for the choice point. The behavior is then executed until a reward point is reached, at which time the action values for the given choice point are updated and a new script generated.

Related Work

There is a significant amount of existing research related to dynamic scripting and its extensions in addition to the previous discussion on reinforcement learning in games.

This includes research on dynamic scripting as well as research from the reinforcement learning literature on decision tree state abstraction and behavior modeling architectures. Each of these topics is covered below.

Dynamic Scripting

While the work described in this dissertation builds primarily on the dynamic scripting algorithm as presented by Spronck et al. (2006), there are a number of other related papers based on dynamic scripting.

Hierarchical Reinforcement Learning

The dynamic scripting value update function has been used in research on hierarchical reinforcement learning (Ponsen, Spronck, & Tuyls, 2006). In this work, the authors compared the performance of the Q-learning value update function

$$Q(s,a) \rightarrow Q(s,a) + a\left[r + g \max_{a'} Q(s',a') - Q(s,a)\right]$$

to a dynamic scripting based value update function, $Q(s,a) \rightarrow Q(s,a) + r$. In the case of the dynamic scripting function, the minimum and maximum action values, and value redistribution, are applied as described in the dynamic scripting section. For a real-time strategy game subtask, similar to the Worker Control game in the preliminary evaluation (Figure 17), they found that using the dynamic scripting-based updating resulted in faster learning than using the standard Q-learning update function. The success of the dynamic scripting update algorithm provides evidence for the utility of using the entire dynamic scripting algorithm as part of a hierarchical behavior modeling system. Our work differs from their approach in that we create a hierarchical version of dynamic scripting rather than using part of dynamic scripting in the context of Q-learning.

Hierarchical Dynamic Scripting

There are two related papers that include work on hierarchical dynamic scripting. In both cases, EDS is an extension and generalization of the existing work.

The first example of hierarchical dynamic scripting related to this dissertation involves using dynamic scripting to play “Wargus”, a real-time strategy game (Ponsen & Spronck, 2004). In their work, Ponsen and Spronck divide the overall learning problem of playing Wargus into a number of sub-problems depending upon the available buildings. In Wargus, the available buildings determine the actions available to the player. In each sub-problem, dynamic scripting was applied independently to learn the value of available actions. That is, they expand the abstract set of states, *S*, from one state to twenty states, where each state *s* represents all game states with a particular building configuration.

The work described in this dissertation differs from their work in two main ways. First, choice points and reward points, by being embedded in a behavior modeling architecture, provide a general learning solution that provides significant representation power beyond the manual state abstraction implemented by Ponsen and Spronck – task decomposition being the foremost example. Second, the work outlined in this dissertation includes methods for automatic state abstraction in addition to manual state abstraction.

The second example of hierarchal dynamic scripting, also in the context of real-time strategy games, focuses on learning in multiple goals rather than in multiple states (Dahlbom & Niklasson, 2006). That is, rather than learning to create an action script (rules in their terminology) for a particular game state, the agent learns to create an action script for a particular goal in the game. This is illustrated in Figure 11, where each goal creates a separate script and selects action from a subset of the complete set of action. This captures the notion that some actions only apply to particular goals. Each goal also has a distinct set of action values (weights) to determine which actions, from its subset, will be included in the dynamically generated script.

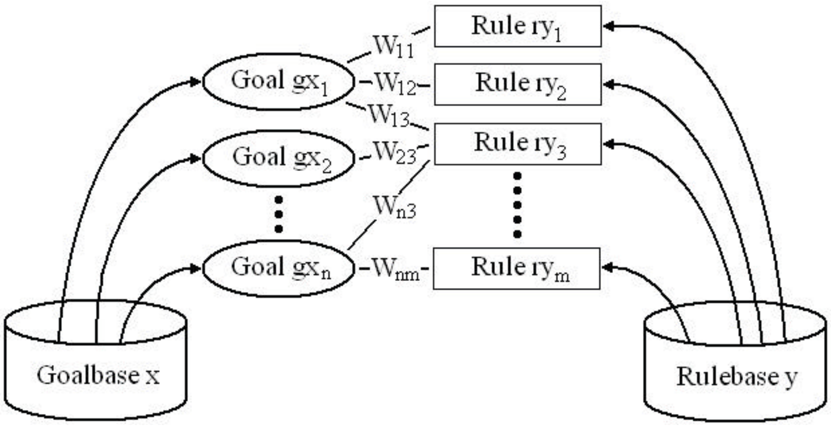


Figure 11 Illustration of the goal-rule layout (Dahlbom & Niklasson, 2006).

While their evaluation is limited to two distinct goals with non-overlapping action subsets in a single abstract game, they develop the idea of creating a goal hierarchy where actions can refer to other goals that need to be completed as part of the action. An example of the goal hierarchy is shown in Figure 12.

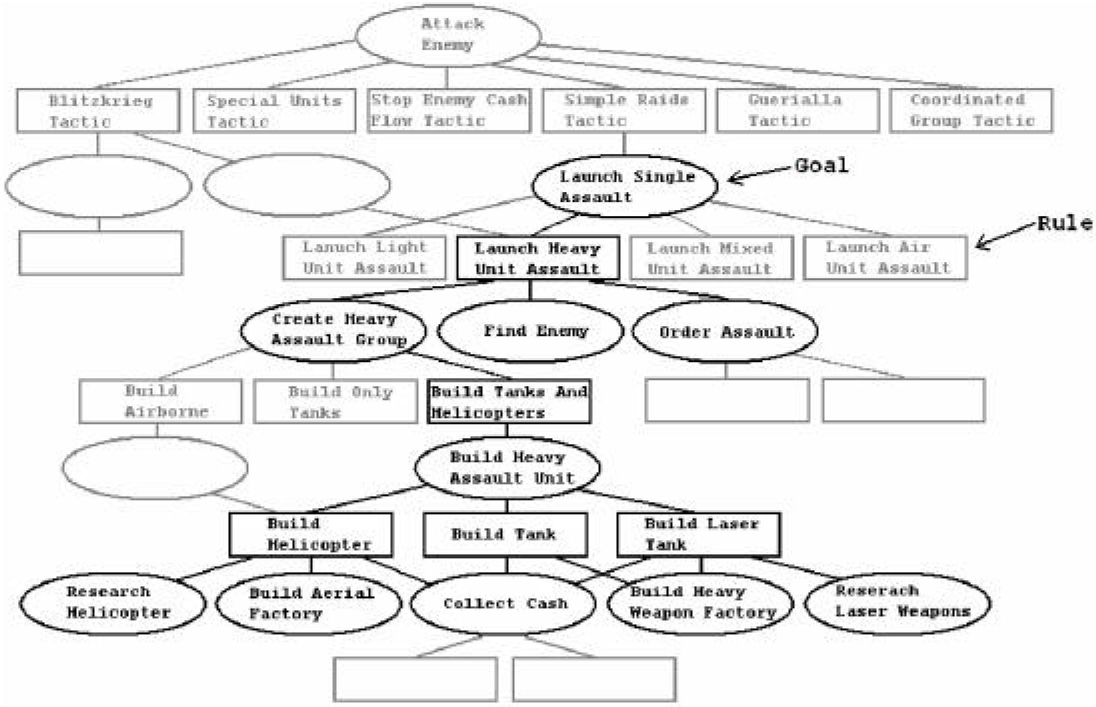
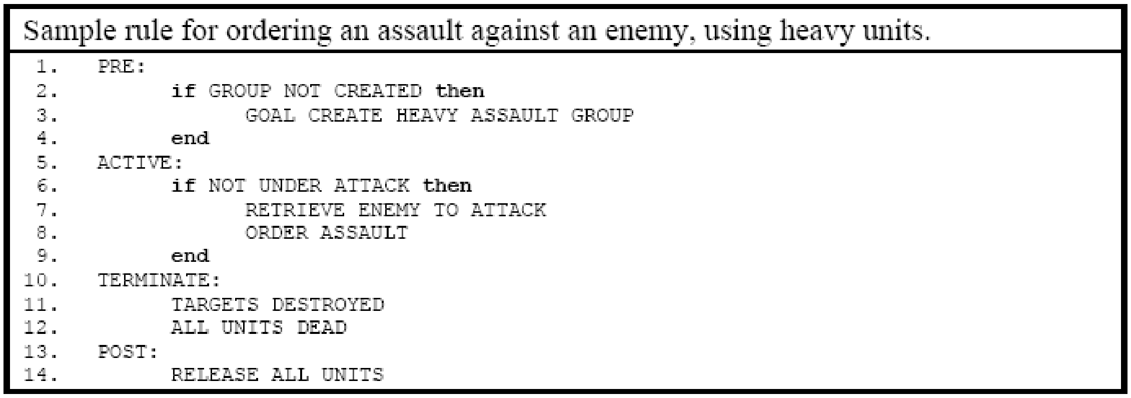


Figure 12 Illustration of a goal-action hierarchy (Dahlbom, 2006).

For example, in the figure the current goal is *Launch Single Assault*. This goal uses dynamic scripting to choose from the available actions that satisfy this goal, selecting the *Launch Heavy Unit Assault* action shown in Figure 13. This action, in turn, can request other goals that must be satisfied as a precondition of the action.

While EDS as described in this dissertation bears much in common with the dynamic scripting based goal node system (GoHDS), there are several significant differences. First, dynamic scripting-based choice points in EDS and dynamic scripting-based goals in GoHDS are designed to be embedded in different types of architectures: goal nodes would be used as part of a goal oriented action planner while choice points are used as part of hierarchical finite state machine.



Sample rule for ordering an assault against an enemy, using heavy units.

```
PRE:
      if GROUP NOT CREATED then
            GOAL CREATE HEAVY ASSAULT GROUP
      end
ACTIVE:
      if NOT UNDER ATTACK then
            RETRIEVE ENEMY TO ATTACK
            ORDER ASSAULT
      end
TERMINATE:
      TARGETS DESTROYED
      ALL UNITS DEAD
POST:
      RELEASE ALL UNITS
```

Figure 13 Example action in GoHDS (Dahlbom, 2006).

These two different architectures have significantly different representation requirements, as can be seen by comparing an EDS graphical behavior (e.g. Figure 7) to the rule given in Figure 13. Second, EDS introduces several concepts not found in GoHDS: reward nodes, adjustments to the underlying dynamic scripting value update algorithm to support the use of dynamic scripting in more situations, and automatic state abstraction. Finally, as implemented GoHDS is a stand-alone library for specifying dynamic scripting-based goals as shown in Figure 11 (Dahlbom & Niklasson, 2006). From reading their paper, it does not appear that GoHDS was integrated with a goal action planner capable of developing a structure like that shown in Figure 12. EDS, on the other hand, is part of a complete behavior modeling package capable of representing a hierarchical state machine version of the given hierarchy. Taken together, these three differences demonstrate that EDS is a significant generalization and extension of the research performed by Dahlbom and Niklasson (2006).

Decision Tree State Abstraction in Reinforcement Learning

This section describes reinforcement learning research that makes use of decision trees to reduce the state space of a game by creating abstract states that represent a number of different game states. These algorithms all start with a single abstract state as found in dynamic scripting, which is the root node of the tree. The algorithms then add leaves to the tree nodes based on features of the game state, where each leaf node represents a collection of similar game states. A *splitting criterion* determines when such a split occurs. Using this tree, reinforcement learning algorithms learn the value of actions for each node in the tree rather than for each state in the game.

The G-Algorithm (Chapman & Kaelbling, 1991) is an early example of creating abstract game states using a tree structure in a Q-learning framework. This algorithm constructs a tree based on the bit values (features) of the game state. An example tree is shown in Figure 14.

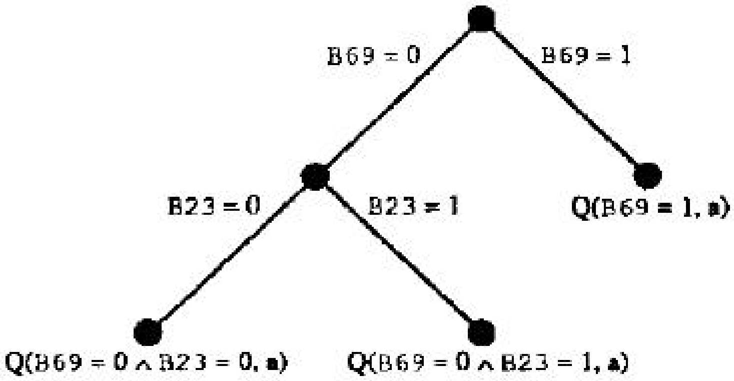


Figure 14 Example G tree (Chapman & Kaelbling, 1991).

In the example G tree, Bit 69 and Bit 23 of the game state are used to choose between three different abstract states. Each abstract state has its own Q-table for tracking the values of actions in this state. The question of when to split a node, the splitting criterion, is answered by using the *t* test to determine the effect of a particular input bit being on or off, on the distribution of rewards (both immediate rewards and discounted future rewards) given to actions. When a node split occurs under the G-Algorithm, the Q-table for the two new nodes is zeroed out, losing all of the learning that occurred prior to the split.

Parti-game (Moore & Atkeson, 1995) is an algorithm that bears some resemblance to a decision-tree based state abstraction algorithm. Parti-game is designed to plan a path from an initial state to a goal state in a continuous environment. It partitions the state space by placing a grid over it, where each grid cell is considered an abstract state. Any individual cell can be divided into further cells, thus allowing for variable resolution as shown in Figure 15.

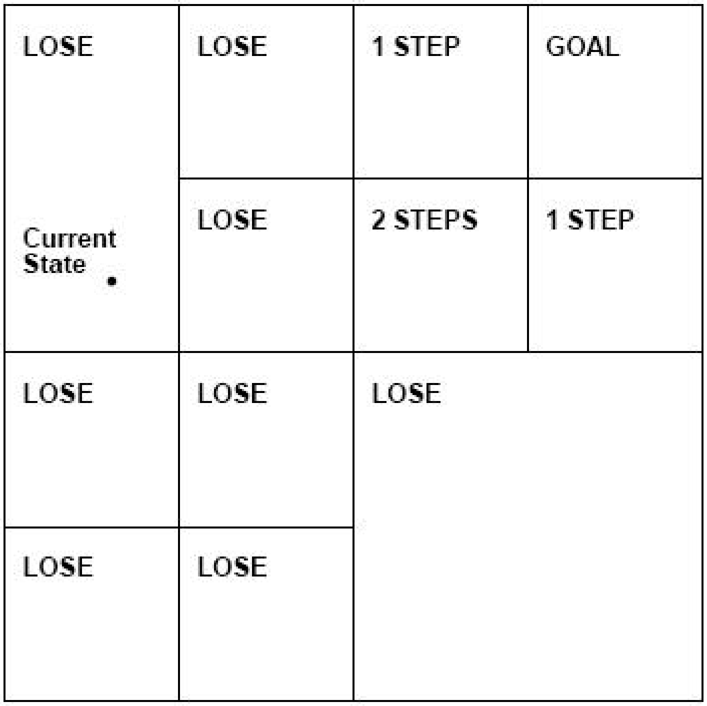


Figure 15 Variable resolution in the Parti-game algorithm, where the entire lower-right hand corner is a single abstract state while the rest of the state space is divided into smaller abstract states(Moore & Atkeson, 1995).

The splitting criterion for this algorithm is the failure of a local greedy controller in a current abstract state (cell).

The U-Tree algorithm (McCallum, 1996) is another extension of the G-Algorithm. Unlike G-Algorithm, U-Tree is instance based. The algorithm keeps a sequential (with respect to time) list of instances, where each instance contains the action taken, the observed state, a value defined by the immediate and discounted future reward. Every *k* steps, the algorithm examines leaf nodes and tests if a split based on a feature or previous action will improve the predication of the reward. The splitting criterion is the Kolmogrorov-Smirnov (KS) test (instead of the *t* test as in he G-algorithm), where the test compares the distribution of instance values across the two nodes. If a node is split, the instances that match each new node are then used to initialize its Q-table; a significant improvement over G-Algorithm which loses all learning after a split.

There have been a number of papers that extend the basic U-Tree algorithm. Continuous U-Tree (Uther & Veloso, 1998) builds on U-Tree by adding the capability to work with continuous features in the game state. The authors also experiment with two different splitting tests: the KS test as used in the G-Algorithm and a test based on the sum-squared error. Further, they examine testing the distribution of rewards for each action individually and combining the results in addition to testing the distribution of instance values. Au and Maire (2004) perform experiments comparing the KS splitting criterion to Information gain ratio, determining the latter performs better in the three test domains. Asadpour et al. (2006) present the SANDS algorithm as the basis for two alternative splitting criteria designed for hierarchical reinforcement learning, which take the magnitude of the instance values into account along with their distribution.

The U-Tree has also been applied to hierarchical reinforcement learning tasks. Jonnsson and Barto (2000) apply a separate instance of the U-Tree algorithm to each subtask in a hierarchical reinforcement learning problem. Each *option* (subtask) has its own list of history instances, which results in separate state abstractions. However, the action values learned in one option can be transferred to other options that contain the same action.

Our work differs from previous work in that we are applying decision tree-based state abstraction to the dynamic scripting algorithm rather than the standard Q-learning algorithm. Additionally, we make use of a boosting algorithm, Decision Stump, in addition to a more standard decision tree algorithm, in the hopes of improving performance in the context of dynamic scripting and modern computer games. A final difference is that our classifier is completely rebuilt every *k* runs rather than splitting existing nodes of a decision tree. The result is a dynamic abstraction that determines the features that are currently relevant to learning.

Hierarchical Reinforcement Learning in Behavior Modeling Architectures

Integrating a dynamic scripting algorithm into a hierarchical behavior modeling system is a primary goal of this dissertation. This section reviews related research by discussing existing game behavior architectures that include hierarchical reinforcement learning. Hierarchical reinforcement learning (Barto & Mahadevan, 2003; Dietterich, 2000) has been previously applied to the problem of online learning for agent behavior in a number different architectures. This section discusses three related architectures that include HRL: ALisp, Icarus, and Soar.

The main difference between the HRL implemented in these three modeling architectures and the work undertaken in this dissertation is the type of reinforcement learning being used. ALisp is based on the standard Q-learning algorithm and both Soar and Icarus use Sarsa derivatives, while the work described in this dissertation is based on the dynamic scripting algorithm. A second difference is that Soar and Icarus makes use of author defined features to create a set of abstract states, while EDS allows for both manual and automatic state abstraction. ALisp does include automatic state abstraction via linear function approximation and also has hooks to allow the user to insert his or her own function approximation algorithm (e.g. decision trees) as desired (Marthi). Finally, there are also significant differences in the agent architectures themselves outside of the learning algorithms, but these differences can safely be ignored.

ALisp

ALisp is a language for specifying partial policies that incorporates features from the MAXQ, options, and hierarchy of abstract machine reinforcement learning algorithms (Andre & Russell, 2002). Partial policies are specified by creating Lisp programs that contain three ALisp methods: *call*, *action*, and *choice*. The *call* method starts a new subroutine, while the *action* method starts a primitive action. The *choice* method chooses a primitive action or subroutine from the list of possibilities given to it, matching up with the *choice points* outlined in the hierarchies of abstract machines algorithm. A combination of call, action, and choose methods are used to create a partially specified hierarchy of behaviors, where the policies for the choose methods are learned at runtime.

The work of Marthi, Russell, and Latham (2005) creates ALisp behaviors for the real-time strategy game Stratagus (a clone of the commercial Warcraft game). The character must learn how to gather resources, build buildings, train more workers, and train enough soldiers to kill the opposing team (an ogre) as quickly as possible. They make use of a concurrent version of ALisp where there are multiple lisp threads running simultaneously. For example, there could be a separate thread for each peasant (resource gatherer) in the game.

Threads can contain *choices,* as described above, at which the lisp program chooses what to do next. For example, the peasant control thread can choose to have the peasant gather resources or construct a building. When the peasant is told what to do a new thread will be spawned to control the peasant. The top level behavior spawns several concurrent behaviors: allocate workers, train workers, allocate gold, train soldiers, and tactical decision. The lower level threads, of which peasant control is one, get to make *choices* such as whether enough soldiers have been trained to attack the ogre. Each choice point represents a distinct learning problem, where the choice point relies on a form of the standard Q-learning algorithm for updating the value of an action with respect to a particular game state:

$$Q(w,u) = Q(w,u) + a\left[r + g \max_{u'} Q(w',u') - Q(w,u)\right]$$

where the state *s* has been replaced by the joint space *w* and the action *a* has been replaced by a set of actions *u*. The joint space is defined as *w* = (*s, q*), where *q* is the current state of the partial program and includes the program counter, call stack, and global memory. The set of actions, *u*, defines the actions that should be taken at each choice point currently active in the partial program.

The size of the learning space is reduced in most choice points as they depend only upon a subset of the joint state space variables, as defined by the behavior author. This author-defined feature vector contains both local (behavior) and global (game/environment) features for each action at each choice point. The number of features at a choice point determines the size of the state space for the Q-learning algorithm. The Q-learning update function used by ALisp contains additional parameters that allow it to take into account only the state features relevant to a given choice point.

Soar

The Soar cognitive architecture has been used to create agent behavior in a number of modern games and simulations (e.g. Jones et al., 1999; Wray et al., 2005).

The basic component of the Soar architecture is the problem space (J.E. Laird & Congden, 2005). An example problem space is shown in Figure 16. To achieve *goals,* Soar uses *operators* to move through a *problem space* represented by *states* that consist of attributes with values and contain a goal and possibly parent and/or child states. The states, with parents and children, form a goal hierarchy. Long term memory (LTM) is made up of productions, which are rules of the form IF x THEN y. LTM represents general knowledge such as object types as available actions. Working memory (WM) contains the current state, such as knowledge about a particular object, as well as the state hierarchy.

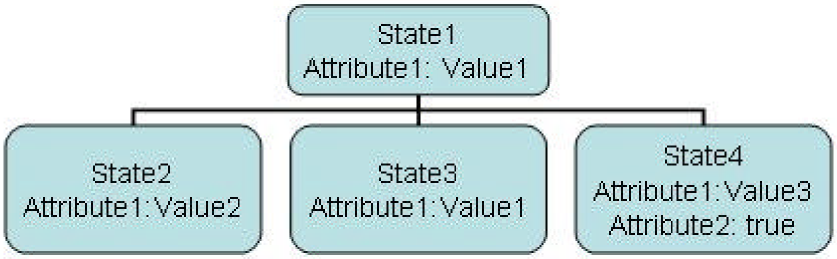


Figure 16 Soar problem space.

The *decision cycle* applies LTM to the current state. There are three stages in the decision cycle: propose operators, select operator, and apply operator. In the propose operator phase, all *elaborations*, operator propositions, and operator comparisons are fired in parallel.[1] This phase continues until no more productions apply (quiescence). Then, from the proposed operators, one is selected. Proposed operators can be given preferences to direct the operator selection mechanism by operator comparison rules. These preferences include: acceptable, reject, better, worse, best, worst, indifferent, numeric-indifferent (biased indifference), require, and prohibit.

Finally, if an operator cannot be selected an *impasse* is reached. To resolve this impasse, a new *substate* is created in which Soar attempts to resolve the impasse. This new state is a copy of the current state, but with a goal of resolving the impasse. If resolved, Soar’s *chunking* mechanism creates a new LTM production to remember what

[1] Newell (1990) describes perfect intelligence as bringing all relevant knowledge to bear, which is what happens during the decision cycle.

to do if this impasse arises again. This new production contains the relevant features of the state prior to the impasse with the relevant action (e.g., operator comparison; operator proposal). One problem with this is that once productions are created they will continue to be used even if they result in undesirable behavior (Newell, 1990).

Nason and Laird (2004) address this shortcoming by adding reinforcement learning operators to Soar, which learn numeric operator preferences in addition to the author determined symbolic (better, worse, etc.) preferences. The IF portion of a production rule defines an abstract state, *s*, where the value of an action, *a*, is *Q(s,a)*. If more than one RL operator matches the current state, then the value *Q(s,a)* is summed across the matching rules as shown in the example below. Actions are selected based on their values using softmax selection. The immediate reward for performing an action is brought into Soar as part of the environmental input. Finally, Soar-RL uses a specialized variant of the SARSA algorithm to update the Q-values:

$$Q(s,a)=Q(s,a)+a\left[r+gQ(s',a')-Q(s,a)\right]$$

where α is the learning rate and γ is the future discount rate. The value update algorithm determines the amount to change the value of *Q(s,a)* based on the reward and divides this amount among all of the rules that gave an operator preferences in state s.

A simplified example, based on the authors' example, has two rules: (1) IF there is a monster to the east and a proposed action is move *east* THEN adjust that operator's value by -5, and (2) IF the proposed operator is to move *east* THEN adjust that operator's value by -.82. If both of these rules match the current state, *s*, then *Q(s, east)* = -5.82. If the agent chooses the east move and receives a reward of -1, the reward is divided between the two rules (-0.5 each). This means that the new preference adjustment for rule (1) would be -5.5 and the new adjustment for rule (2) -1.32. However if only rule (1) matched, then the value of *Q(s, east)* = -5 and the reward of -1 would cause the preference of rule (1) to change to -6. By using operator preference rules, Soar can have a mix of general abstract states and specific abstract states, depending on how the behavior author defines the operator preference rules. The usefulness of this feature was further demonstrated by Wang and Laird (2007).

Icarus

Langley and Choi (2006) describe the Icarus cognitive architecture, which has much in common with the Soar architecture described previously. For example, an action (skill) can either be a primitive action or a subgoal that requires additional actions to fulfill. The hierarchical reinforcement learning in Icarus (Shapiro, Langley, & Shachter, 2001) shares much in common with both ALisp and Soar. First, Icarus makes use of choice points in its formulation of hierarchical reinforcement learning. Icarus is a planning based language, where each plan contains a set of requirements, the objectives it achieves, and the methods to accomplish the plan. Hierarchies are formed by allowing plans to point to actions. Each plan can choose between its methods (actions) to complete the objective – this is where the choice points are. The second similarity is that each choice point in Icarus can perform state abstraction by making use of a limited number of features from the environment/game state.

Icarus makes use of the SHARSHA algorithm to perform value updates. SHARSHA is derived from the Q-learning Sarsa algorithm but takes into account the current execution stack. The same underlying algorithm is used, $Q(s,a)=Q(s,a)+a\left[r+gQ(s',a')\right]$, but it is applied to all of the methods in the current call stack. An example given by Shapiro, Langley, & Shachter (2001) is a hierarchical call stack in state *s* that contains three methods: Drive, GetToTarget, and Accelerate, where Drive calls GetToTarget which calls Accelerate. When a reward is received, *Q(s, Drive)*, *Q(s, GetToTarget)*, and *Q(s, Accelerate)* are all updated.

Contribution

In the light of this related work, the specific contributions of this dissertation are:

1. To extend and generalize previous research on the dynamic scripting and hierarchical dynamic scripting algorithms through the addition of choice points, reward points, scaled value updating, and architecture integration.
2. To apply previous work in decision tree state abstraction for Q-learning in a dynamic scripting context and to extend this previous work by evaluating the effectiveness of the simple Decision Stump algorithm.
3. To extend previous research on hierarchical reinforcement learning in behavior modeling architectures by introducing arbitrary reward nodes.

4. To integrate a fast and efficient reinforcement learning algorithm with a graphical task network architecture with the goal of making machine learning in games more accessible to behavior modelers.

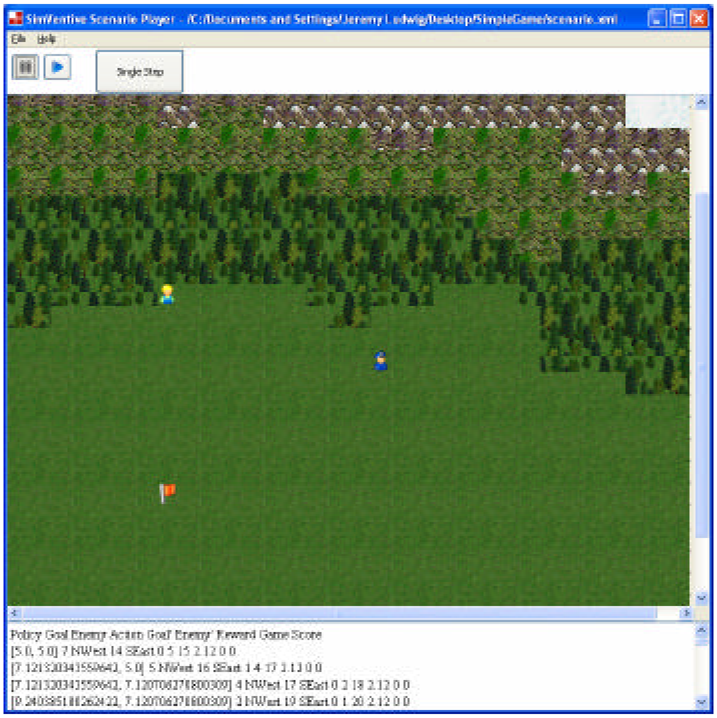
Illustrative Example

Two simples games, based on a real-time strategy sub-problem, are used to illustrate the dynamic scripting extensions. These experiments are based on a preliminary version of the extensions and were presented in June of 2007 (Ludwig & Farley, 2007). The first game examines the ability to create hierarchical learners that make use of automatic state construction. The goal of these agents is to perform the task as well as possible. The second game builds on the first by creating a meta-level behavior that performs game balancing in a more complex environment.

Using EDS to Control a Worker

The *Worker* game attempts to capture the essential elements of a behavior learning problem by reproducing a game used by Ponsen, Sponck, & Tuyls (2006). While they used this game to compare dynamic scripting and Q-learning algorithms in standard and hierarchical forms, this work uses extended dynamic scripting to make higher level decisions.

This simple game, as shown in Figure 17, involves three entities: a soldier (blue agent on the right), a worker (yellow agent on the upper left), and a goal (red flag on the lower left). The soldier randomly patrols the map while the worker tries to move to the goal. All agents can move to an adjacent cell in the 32 by 32 grid each turn. A point is scored when the worker reaches the goal; the game ends when the enemy gets within eight units of the worker. If either the worker reaches the goal, or the game ends, then the goal, soldier, and worker are placed in random locations.


SimVentive Scenario Player - /C:/Documents and Settings/Jeremy Ludwig/Desktop/SimpleGame/scenario.xml
File
Help
Single Step

```
Policy Goal Enemy Action Goal' Enemy' Reward Game Score
[5.0, 5.0] 7 NWest 14 SEast 0 5 15 2.12 0 0
[7.121320343559642, 5.0] 5 NWest 16 SEast 1 4 17 2.12 0 0
[7.121320343559642, 7.120706271800309] 4 NWest 17 SEast 0 2 18 2.12 0 0
[9.240385108262422, 7.120706271800309] 3 NWest 19 SEast 0 1 20 2.12 0 0
```

Figure 17 Worker control game.

The flat behavior of the worker uses a choice point to decide between moving a) directly towards the goal or b) directly away from the enemy. Each move is immediately rewarded, with +10 for scoring a point, -10 for ending the game, and a combination of the amount towards the goal and away from the enemy (1 * distance towards goal + 0.5 * distance away from enemy). This differs from previous work that used dynamic scripting only to learn how to perform a) and b) not decide between the two. That is, this choice point is learning to make a tactical decision—not how to carry out the decision.

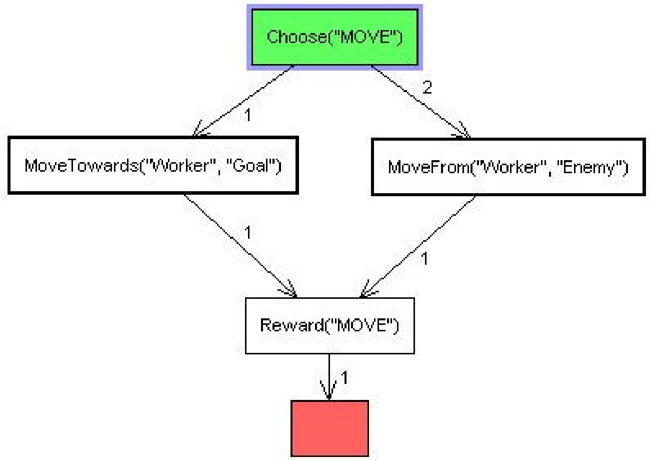


Figure 18 Basic worker behavior in the worker control game.

The behavior for this worker is shown in Figure 18. In this case, the primitive action *MoveTowards* selects the move that gets the worker closest to the goal and the primitive action *MoveFrom* selects the move that gets the worker as far away from the enemy as possible. For the MOVE choice point, and all other choice points in these two sample games, the actions all have the same priority and the script size is equal to the number of available actions. The reward point updates the action value associated with each action using the EDS value update mechanism. Based on the choice point settings, script ordering is determined solely by the value associated with each action.

The hierarchical version of the worker behavior, H_Worker, replaces the primitive action *MoveTowards* with a sub-behavior and introduces the PURSUE choice point in the new MoveTowards sub-behavior as seen in Figure 19. This version of the behavior allows the agent to choose between moving directly to the goal and selecting the move that moves towards the goal while maintaining the greatest possible distance from the enemy. The reward function for this choice point is the same as that for the MOVE choice, and the action values updated using the EDS mechanism whenever the reward action is reached in the sub-behavior.

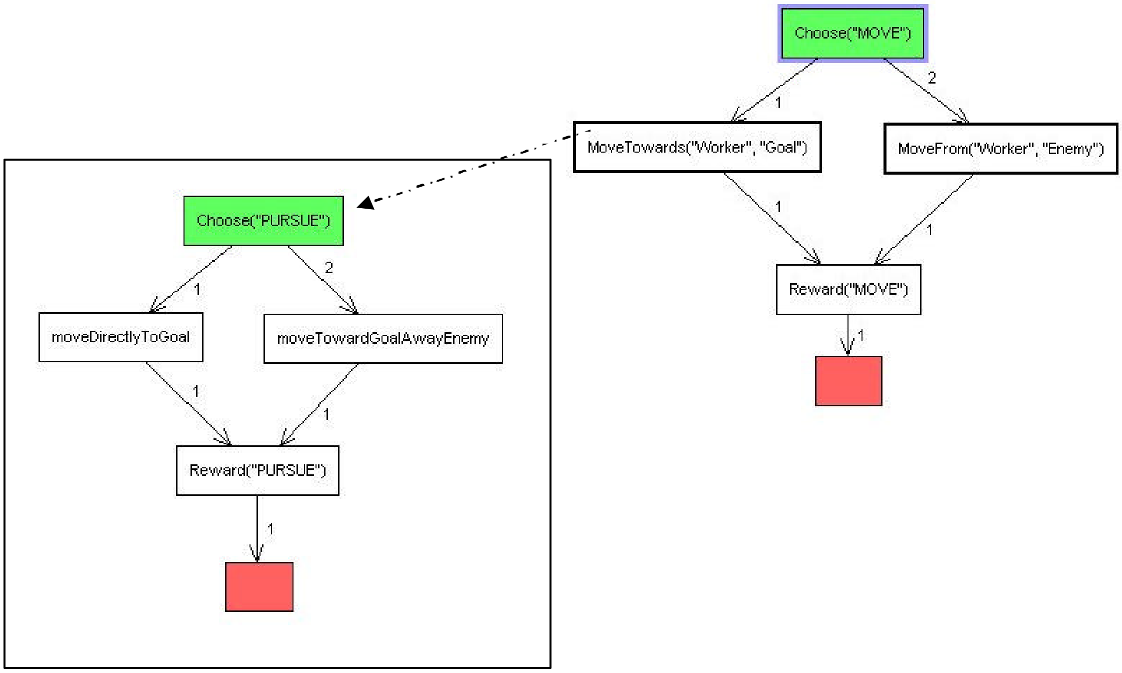


Figure 19 Hierarchical version of the worker behavior.

The Worker and H_Worker behaviors were each used to control the worker agent in 200 games with the given choice points, where all of the agents were positioned randomly at the start of each game and the values for all actions were set to 5. A game ends when the soldier catches the worker, so the worker can score more than one point during a game by reaching the flag multiple times. The dynamic scripting learning parameters were fixed using the reward function described previously, a maximum action value of 30, and a minimum action value of 1. Compensation and value redistribution were not enabled in this preliminary work.

The Worker scored an average of 2.2 goals over the 200 games, and this result functions as a base level of performance. The learned policy generally hovered around [30 (to goal), 1 (from enemy)] for the MOVE choice point, which essentially causes the agent to always move directly towards the goal. In certain random instances the worker might lose more than once in a row, reversing the policy to favor moving away from the soldier. This would cause the agent to move to the farthest edge until it eventually moves back towards the policy that directs the agent to the goal ([30, 1]). Note that for this work, the value distribution mechanism involving compensation and redistribution was not used so the action value sums do not remain constant.

With the goal of improving behavior, automatic state construction was used to classify the game state so that one of two policies would be used. A feature vector was generated for each choice point selection that included only the features previously identified (Ponsen, Spronck, & Tuyls, 2006): relative distance to goal, direction to goal, distance to enemy, direction to enemy, and the received reward (see Table 6).

Table 6 Feature vector in the worker control game.

Goal Distance	Goal Direction	Enemy Distance	Enemy Direction	Reward
6	S	9	SE	1
5	S	7	SE	-10

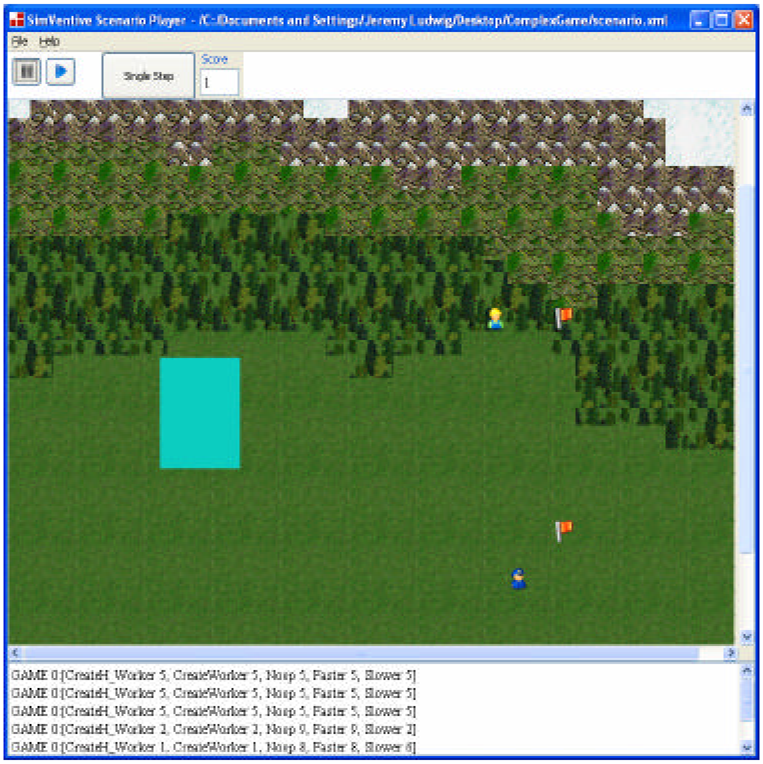
In this initial work, automatic state specialization was only performed with the Decision Stump algorithm. In this case, automatic state specialization was performed after 1, 2, 5, 10, 20, and 100 games to partition the game state with varying amounts of data. The algorithm for creating the classification tree and training the resulting leaf nodes is described in the section titled Automatic State Specialization. After creating the classification tree and update the action values for the leaf nodes, each leaf node contains its own set of action values and a distinct dynamically generated action script. When the choice point makes a decision, it classifies the current game state using the generated classifier to determine which script should currently be used to select an action.

After 1 game the created classifier divided the game state into one of two policies based on Distance_Goal <= 1.5, which had no significant effect on agent behavior. In all other cases, the generated classifier was Distance_Enemy <= 8.5. The DS algorithm with this classifier improved significantly (p < .01), scoring an average of 2.9 goals over the 200 runs. Visually, the worker could be seen sometimes skirting around the enemy instead of charging into its path when it was nearly within the soldiers range.

The H_Worker, without state construction, performed significantly better than either version of the Worker behavior, with an average score of 4.2 (p < .01) over the 200 runs. Similar to the Worker behavior, the MOVE choice point generally hovered around [30 (to goal), 1 (from enemy)]. For the PURSUE choice point, the values generally favored moving towards the goal but away from the enemy rather than moving directly to the goal [1 (direct), 30 (to goal from enemy)]. Visually the H_Worker will generally spiral to the goal, which allows for moving toward the goal while maintaining the greatest possible distance from the enemy. Applying the Decision Stump classifier after 1, 2, 5, 10, 20, and 100 games always resulted in creating the Distance_Goal <= 1.5 classifier which had no significant effect on the average score.

Using EDS for Game Balance

The second experiment builds on the Worker and H_Worker behaviors by creating a behavior that learns how to balance the expanded version of the Worker game shown in Figure 20. These two behaviors were chosen as the low and high performers of the previous experiment. For both workers, automatic state construction is turned off.


SimVentive Scenario Player - /C:/Documents and Settings/Jeremy Ludwig/Desktop/ComplexGame/scenario.xml
File
Help
Single Step
Score
1

```
GAME 0:[CreateH_Worker 5, CreateWorker 5, Noop 5, Faster 5, Slower 5]
GAME 0:[CreateH_Worker 5, CreateWorker 5, Noop 5, Faster 5, Slower 5]
GAME 0:[CreateH_Worker 5, CreateWorker 5, Noop 5, Faster 5, Slower 5]
GAME 0:[CreateH_Worker 2, CreateWorker 2, Noop 9, Faster 9, Slower 2]
GAME 0:[CreateH_Worker 1, CreateWorker 1, Noop 8, Faster 8, Slower 6]
```

Figure 20 A screen shot of Worker 2, the expanded version of the worker game.

In Worker 2, one random goal is replaced with two fixed goals. There is still one soldier that randomly patrols the board. The starting location of the agents is also fixed to be the top of the square barracks in the figure.

At the beginning of each game, the soldier starts out in the same location and performs the same random patrol of the game board to allow for easy comparison across different runs. During its patrol, the soldier will sometimes hover around one of the goals, in the middle of the goals, at the worker creation point, or somewhere on the outskirts of the game board. The random path of the soldier serves as the dynamic function that the game balancing behavior must react to and demonstrate different levels of human player competency for (or attention to) a subtask within a game.

Workers are created dynamically throughout the game and multiple workers can exist at any time. All workers share the same choice points. That is, all instances of Worker and H_Worker share the same set of values for the MOVE choice point and all H_Worker instances share a single set of values for the PURSUE choice point. So, for example, if one worker is caught all of the remaining workers will be more likely to move away from the enemy. In this game, every time a worker makes it to the goal the workers score one point. Every time the soldier captures a worker, the controller scores one point. At the end of each episode the score decays by one point, with the idea that it isn't very interesting when nothing happens and the decay will eventually cause workers to be created.

The Worker and H_Worker behaviors were modified to work in the context of this new game. First, the MOVE choice point in both the Worker and H_Worker is used to decide among moving towards goal 1, towards goal 2, or away from the enemy as shown in Figure 21.

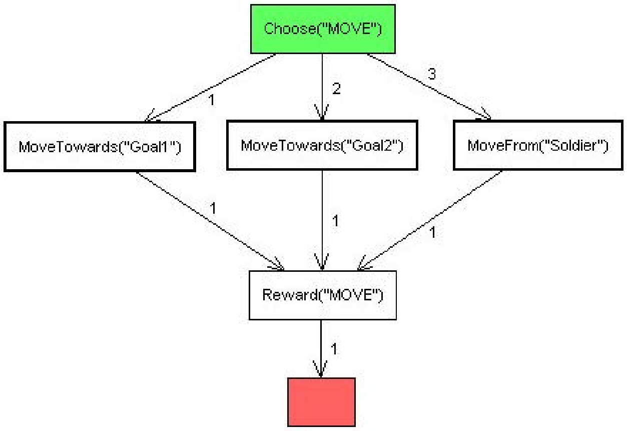


Figure 21 Basic worker behavior in the Worker 2 game.

Figure 22 shows the modification of the H_Worker MoveTowards sub-behavior. Now this behavior chooses from moving directly to the goal, moving towards the goal and maximizing the distance from the enemy, or moving towards the desired goal and minimizing the distance between the worker and the other goal.

The behaviors of the modified Worker and H_Worker agents are very similar to the behaviors of the version described previously. The reward function and learning parameters were not changed for these behaviors, so the system is attempting to learn the best possible actions for these agents. At the MOVE choice point, the main difference is that workers will all go to one goal until the soldier starts to capture workers heading to that goal. At this point, the workers quickly switch to moving towards the second goal. This works very well if the soldier is patrolling on top of one of the goals. At the H_Worker PURSUE choice point, moving towards the goal but away from the enemy was generally preferred over the other two possible actions.

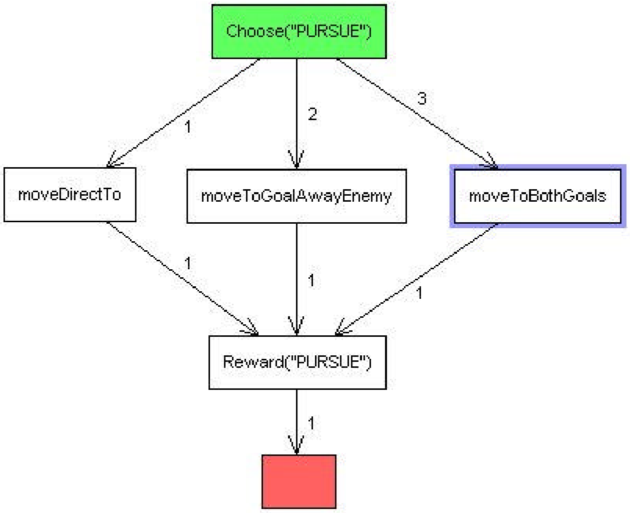


Figure 22 Sub-behavior used by the hierarchical worker in the Worker 2 game.

The game balancing behavior shown in Figure 23 attempts to keep the score as close to zero as possible by performing a number of possible actions each episode (30 game ticks). The available actions are creating a Worker, creating an H_Worker, doing nothing, speeding up (performing more actions each episode) and slowing down (performing fewer actions each episode). This meta-level behavior was created using the same choice point infrastructure used in the agent behaviors.

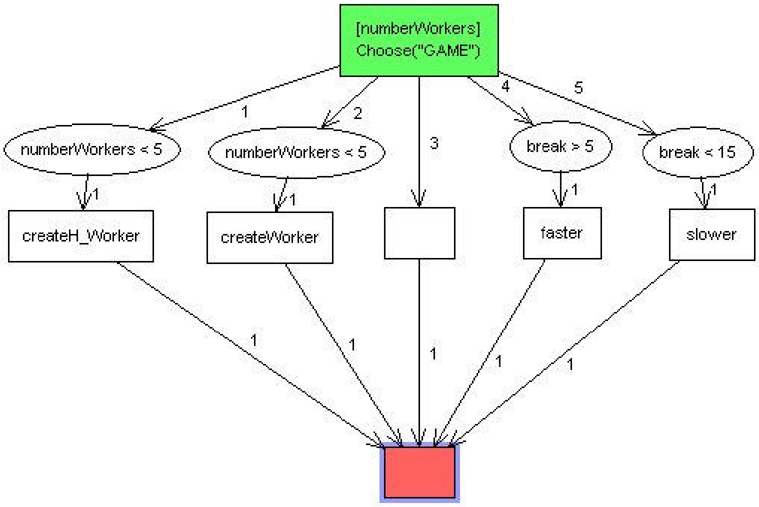


Figure 23 Game balancing behavior.

The learning parameters for this choice point were different than the parameters for the worker agents. First, the minimum action value was set to 1 and the maximum action value was 10. The temperature of the SoftMax selection algorithm remained at 5. The reward function was defined as:

$$\left|previousScore\right| - \left|currentScore\right|$$

This rewards actions that bring the score closer to zero. For example, if the score changes from -5 to -10, a reward of -5 is given while if the score changes from -5 to 2 a reward of 3 is given. Unlike the worker agents which are rewarded immediately, the game balancing behavior only receives a reward at the end of an episode. Similar to the original Worker game, the *distributeRemainder* function was not used in this preliminary investigation so the action value sum does not remain constant (the complete algorithm is evaluated in Chapter 0).

The game balancing behavior was allowed to run for 100 episodes (3,000 actions) to form a single game. In each game, the soldier starts at the same position and makes the exact same movements. That is, there is a single randomly generated path that the soldier follows in each game. For comparison, two other reward functions were also tested. The first doubles the reward, making the system adjust values in bigger increments. The second halves the reward so the system adjusts values in smaller increments. To provide an upper level bound on behavior, a fourth algorithm was created where the GAME choice point was replaced by a random decision.

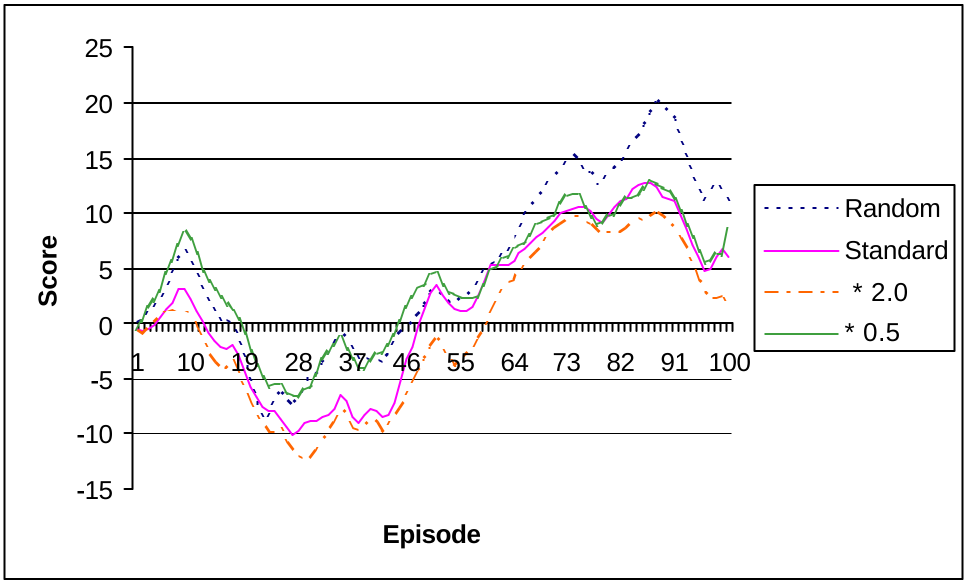


Figure 24 Game balance results.

The average score after each episode for the four different cases is presented visually in Figure 24. Remember, the ideal performance on this task is to always keep the score at zero, which would be a flat line x-axis at zero. This graph indirectly indicates the position of the soldier at various times during the game (100 episodes) as it follows its fixed path. Initially the soldier starts near the goals. In the middle portion of the game the soldier is located so it tends to capture all of the workers as soon as they are created, driving the score down. Towards the end of the game the soldier wanders around the periphery and scores tend to go up.

To capture a quantitative measurement of the difference between the actual scores and desired score of zero, the mean absolute error was also computed across the eight games for each type and is shown in Table 7. The mean absolute error is defined as:

$$\frac{1}{n}\sum_{i=1}^{n}\left|e^{i}\right|$$

where e^i is the mean of the absolute scores (across the eight runs) at time *i* for a single learner. The standard, double, and half reward cases are all statistically significant improvements (p < .01) over the random agent.

Table 7 Mean absolute error of the four different learners. The closer the value to zero, the better the relative performance.

Random	Standard Reward	Double Reward	Half Reward
12.0	8.3	9.8	6.9

The game choice behavior did perform as expected in some games. For example, it could be seen that if the score was positive and the workers were being captured then it would ratchet the speed to maximum and create standard workers (thus raising the score by having more get captured). On the other hand, if the score was negative and the workers were being captured, the speed would be slowed to the minimum and only H_Workers created as they were less likely to be caught. While promising, these two games are only an initial test of a preliminary version of the extended dynamic scripting algorithm. Later chapters evaluate the performance of EDS in the context of several abstract tactical games and demonstrate its capabilities in the published computer game NeverWinter Nights.

CHAPTER IV
EVALUATION

In order to better evaluate the performance of EDS, we created an abstract game framework based on high-level (tactical) decisions in games and simulations. This framework is designed to allow for the construction and playing of abstract games, which can be used to test the efficacy of different game playing agents equipped with learning algorithms. The first section in this chapter is devoted to describing the abstract framework, while the second section evaluates EDS on four different abstract games.

Tactical Abstract Game Framework

The Tactical Abstract Game (TAG) framework is derived from simple decision simulations such as the *n*-armed bandit problems, where each of *n* actions has a different reward associated with its completion (Sutton & Barto, 1998). We extend this type of simulation to include decision aspects commonly encountered in games and simulations: applicability of actions, priority of actions, and a limited influence of game state. With these additions the TAG framework can be used to model some instances of the class of decision problems referred to as Markov Decision Process (MDP).

An MDP is a sequential decision problem. A sequential decision problem is one in which the agent interacts with the environment, the state of the environment changes as a result of the interaction, the agent interacts with the environment, etc., until some termination state is reached. An MDP is a particular class of decision problem where "… the transition probabilities from any given state depend only on the [current] state and not on the previous history" (Russell & Norvig, 1995). More formally, an MDP is defined by the tuple <*S, A, T, R*> (Kaelbling, Littman, & Cassandra, 1998), where:

- *S* is a set of states in the environment
- *A* is the set of actions
- *T* is the state transition function, where *T(s, a, s')* defines the probability of ending up in state *s'* when action *a* is performs in state *s*

- *R* is the reward function, where *R(s, a)* is the expected immediate for performing action *a* in state *s.*

While there are a number of existing languages in which to describe MDP-based games, such as the Game Description Language (Genesereth & Love, 2005), the TAG framework is designed to capture the essential characteristics and considerable role of randomness found in modern computer games while at the same time minimizing the amount of the game that must be specified. To this end, TAG can only represent some MDPs as the language has limited representational capabilities with respect to the set of games state, the transition function, and the reward function.

Below we define the two main components of the TAG framework, game and player, and the major attributes of these components. Following this, we illustrate how to run a TAG experiment with these two components.

TAG Game

A game in the TAG framework is made up of a number of components: a game feature set that corresponds to the agent observations, *F,* a set of actions, *A*, and a set of state transition rules for the observation features, *R.* The tuple <*F, A, R*> defines a game with a particular set of observations and actions available to the agent.

The observation feature set of a game, *F*, contains the game features observable to a player when it selects an action and defines the set of game observation states, *O*. Each observation state is defined as a distinct set of feature values. This is a significant departure from MDPs, which contain the actual set of game features, F_s, and the corresponding set of game states, *S*, in addition to the set of observation features, *F,* and observation states, *O*. The TAG framework simplifies the process of game construction by relying on the significant amount of randomness seen in modern computer games instead of an underlying game state model.

Each feature, *f*, is either defined as a Boolean or integer feature. In the case of Boolean, the value of the feature is either TRUE or FALSE. For a numeric feature a range is given (e.g. 0 to 64) and the value of the feature will always be in this range. For example, let f1 and f2 be the two features of a game, both of which are Boolean. This leads to four game observation states in the set *S*: s1 = (true, true), s2 = (true, false), s3 = (false, false), and s4 = (false, false). Note that the feature sets only describe the features available to a player when it selects an action, not all of the features available in the game. Optionally, a feature can be assigned an initial value; otherwise it will be determined randomly.

The second basic component of a game definition is the set of actions, *A*. Each action is defined by a set of parameter values that vary depending on the current game state. For a given set of game observation states, *O*, based on the observation feature set, *F*, a parameter tuple is defined as $< O, p, r-, r+, g>$.

- ***O***: a set of observation states where this action is available
- **p**: a positive reward likelihood, which defines the probability of receiving a reward in the positive range rather than the negative range
- **r+**: a positive reward range, where the given reward is selected randomly from within the given range (inclusive) when a positive reward is given
- **r-**: a negative reward range, where the given reward is selected rangomly from within the given range (inclusive) when a negative reward is given
- **g**: the probability that the action can be applied in the current state, given that it is available according to *O.*

An action, *a*, is composed of multiple tuples: $a = \{< O_1, p_1, r\text{-}_1, r\text{+}_1, g_1 >, < O_2, p_2, r\text{-}_2, r\text{+}_2, g_2>, < O_3, p_3, p\text{-}_3, p\text{+}_3, g_3 > \ldots\}$. Across the attributes sets that define an action, the observation state sets (O_1, O_2, etc.) are distinct. All other attributes may be the same or different in each attribute set.

Positive reward, p, is a value between 0 and 1 that captures the likelihood of a “good” reward if the action is selected. Zero indicates that only negative rewards will be generated and one indicates only positive rewards will be generated. The actual amount of reward is determined randomly between a minimum and maximum for each of the two cases, to capture the inherent stochastic nature of modern computer games and simulations and to simulate the effect of the opponent’s actions. r+ defines the range of positive reward (e.g. 0 to 100) and r- defines the range of negative rewards (e.g. 0 to -70). That is TAG is only capable of producing a reward function, *R(s,a)*, that generates a random reward distribution within the given bounds.

The applicability*,* g*,* of an action lies between 0 and 1 and describes how often an action is available to be applied. Zero indicates that an action is never available while one indicates that the action is always available. In a MDP, the current game state *s* would be used to determine whether or not an action is available. While this type of applicability can be defined within the TAG framework by limiting the observation state *o* and assigning g = 1.0, the framework also supports a second method by adjusting the applicability value. The applicability value is a significant simplification that instead determines the availability of an action randomly without the need for information on the actual or observed game state. For example, g could be used to specify that an action is applicable 10% of the time, irrespective of the current game observation state *o*.

The set of state transition rules, *R,* move the agent through the observation state based on the completed actions. Each rule, $r \in R$, contains both an action *a* and a feature *f*. By default, when action *a* is selected while running a game, the observation feature *f* is randomly changed to create a new observation state. Alternate rule types exist to change the observation state in a more principled way, such as setting a feature to a particular value or (in the case of integers) adjusting the existing value up or down. Additionally, no-op actions are defined by not including any transition rules for a given action.

TAG Player

A TAG Player is responsible for making decisions in a TAG Game, where each player can implement a different action selection method. There are three different player implementations examined in this dissertation: Random, Q, and EDS.

The Random player randomly selects an action at each decision point. This player is used to define a (hopefully) lower bound on performance, with which the other learners can be compared.

The Q player illustrates the performance of a standard reinforcement learning algorithm on the same problem. It is based on the Q-learning algorithm as described by Sutton and Barto (1998). The Q player selects actions in a fitness proportionate manner, using soft-max selection, whenever an action is requested. The Q player updates the action values using the Q-learning update function, where *r* is the reward received, *Q(s,a)* is the value of action *a* in state *s*, and *s'* is the resulting state when action *a* is selected in state *s:*

$$Q(s,a) = Q(s,a) + a\left[r + g \max_{a'} Q(s',a') - Q(s,a)\right].$$

The parameters were set to the following values based on Sutton and Barto (1998): α = 0.1 and γ = 0.8. The Q-player provides a second type of baseline – the performance of a standard reinforcement learning algorithm on the same problem. This is meant to be representative of the dominant paradigm but not necessarily complete. There are a number of ways the basic Q-learning algorithm could be extended to improve performance that are not investigated in this dissertation.

For EDS, the extended dynamic scripting player, a number of parameters determine how the player's learning algorithm works. The EDS player parameters take the form of a tuple <*I*, *V,* m+, m-, n, e> where *I* is a set of priorities (one value for each action), *V* is a set of action values (one for each action), m+ is the maximum action value, m- is the minimum action value, n is the size of the script for this choice point, and e is the number of actions taken in each episode. The priority*, i,* is an integer where 0 has the greatest priority (to avoid confusion, we use English terms such as High, Medium, and Low to describe priority). The assignment of priority represents subject matter knowledge from the behavior author, in both determining the number of priorities and in assigning them. An action value, *v*, is used by the dynamic scripting algorithm to determine if the action is included in the agents behavior script. The m+ and m- parameters constrain the possible values of *v.* The script size, n, is the number of actions that are selected to be part of the dynamic generated script at the beginning of each episode. Episode length*,* e*,* is the number of actions selected before learning occurs – the EDS player tracks the number of action selections. For example, if e = 10, after ten actions the action values will be updated and a new script is generated.

<u>TAG Experiment</u>

Running a TAG experiment to generate empirical results involves both a TAG game and player. The primary measure when performing an experiment is the reward received after each action selection. The player selects an applicable action, *a*, from *A* based on the current settings of the player, the current observation state *o* and the applicability threshold g. The information in *a* is used to supply a numeric reward to the player for the selected action. After the reward is given, the game play rules, R, are used to change the game observation state *o*.

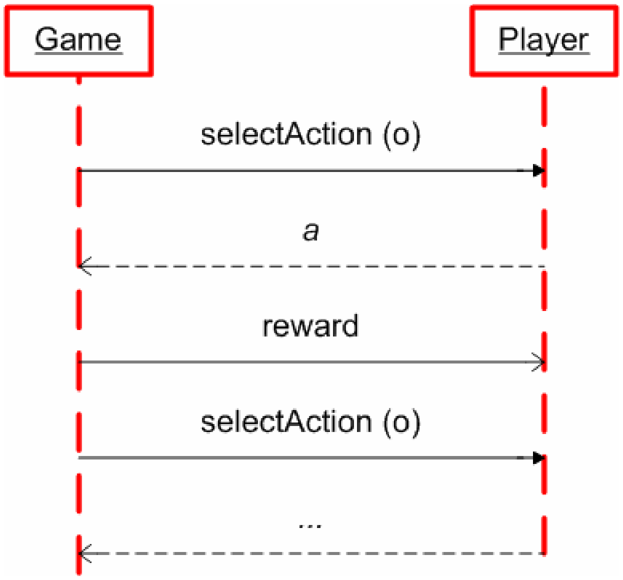


Figure 25 An interaction diagram that describes the relationship between a TAG game and player, where *o* is the game state, g the applicability threshold, and *a* the action selected by the player

As shown in Figure 25, the game asks the player to select an action while providing it information on the current state *o*. The player then returns the selected action *a*, which has an applicability value >= g, or null if no action is applicable based on the value of g. Following this, the game sends the player a message with a reward generated based on the settings of action *a*, or null in the case that no action was selected.

Evaluation of EDS in Four Abstract Games

We define four distinct games in the TAG framework, with a number of different players for each game: Weather, Anwn, Get the Ogre!, and Resource Gathering. The first is adapted from a weather prediction game, previously used to study both human and machine learning (Santamaria & Warwick, 2008). The second is an abstract role-playing game, based in part on the NeverWinter Nights computer game (Bioware, 2002). The third game, Get the Ogre!, is derived from a real-time strategy game subtask, previously explored with Concurrent ALisp (Marthi et al., 2005). The Resource Gathering game builds on another real-time strategy subtask studied by (Mehta, Ray, Tadepalli, & Dietterich, 2008).

These four games represent wide range of game types. The weather prediction game, Weather, is extremely context dependent, in a small state space. Q-learning should perform much better than extended dynamic scripting in this type of game. In the abstract role playing game, Anwn, the current observable state plays little role in the expected utility of the high-level decisions made in the game. In this type of game, extended dynamic scripting should perform very well. In the third game, Get the Ogre, actions must be performed in a sequence in order to solve the problem. While action priority in EDS can capture some elements of sequence, it is expected that context sensitive learners (such as table-based Q-learning or EDS with automatic state abstraction) will perform better than learners without context information (EDS with only a single abstract state). The fourth game, Resource Gathering, also requires a sequence of actions to complete. However, the expected sequence is much more constrained than that found in the previous game. As predicted in Get the Ogre!, EDS will perform very poorly on the Resource Gathering game unless additional domain knowledge is included.

For each of these games, we first present a description of the game and players. This is followed by a set of results and discussion of their significance. Additionally, for the Anwn game performance results are given that summarize the computational cost of EDS relative to the Random and Q players. Anwn was selected as it represents the most complex game with respect to the number of available actions and the size of the observation state space.

Weather Prediction

The Weather Predication game is adapted from a simple game previously used to study both human and machine learning (Santamaria & Warwick, 2008). A participant in this game is presented with one or more of four possible cards. Based on the visible cards, the player predicts one of two outcomes: Fine or Rainy weather. The actual weather outcome is determined as described in Table 8. Based on the random element included in determining the actual outcome, the upper bound on the prediction accuracy of the participant is roughly 83%.

Table 8 The 14 possible card patterns and the probability with which they predicted fine weather.

Pattern	Cues (cards present)				Probability of fine weather
1	1	2	3		0.895
2	1	2		4	0.778
3	1	2			0.923
4	1		3	4	0.222
5	1		3		0.833
6	1			4	0.500
7	1				0.895
8		2	3	4	0.105
9		2	3		0.500
10		2		4	0.167
11		2			0.556
12			3	4	0.077
13			3		0.444
14				4	0.105

This table can be summarized as: Card 1 strongly indicates Fine, Card 2 weakly indicates Fine, Card 3 weekly indicates Rainy and Card 4 strongly indicates Rainy.

Game Definition

The definition for the Weather game includes *F, A,* and *R.* The game contains four boolean observation features corresponding to the four cards. There are two actions available to the player, predicting Fine or Rainy weather, with context specific rewards that correspond to the probabilities given in Table 8.

- *F* = { Card1 :Boolean, Card2 :Boolean, Card3 :Boolean, Card4 :Boolean }
- *A = {Fine, Rainy}* where both actions have 14 different parameter sets < *O*, p, r-, r+, g >
    - *Fine* = {
        - <*{1, 2, 3}*, 0.895 , 50 to 50, -50 to -50, 1.0>,

        - <*{1, 2, 4}*, 0.778 , 50 to 50, -50 to -50, 1.0>,
        - …
        - <*{4}*, 0.105 , 50 to 50, -50 to -50, 1.0>}
    - *Rainy* = {
        - <*{1, 2, 3}*, 0.105 , 50 to 50, -50 to -50, 1.0>,
        - <*{1, 2, 4}*, 0.222 , 50 to 50, -50 to -50, 1.0>,
        - …
        - <*{4}*, 0.895 , 50 to 50, -50 to -50, 1.0>}
- *R* =
    - Randomly change all four features.
    - If no action is applicable, i.e. all cards are present or no cards are present, then a new game state is randomly generated.

Player Definition

We define a number of different players for this game: (1) Random, (2) EDS, and (3) Q. The random player selects an applicable action each time a decision is required. This player is used to provide a lower bound on the prediction accuracy, contrasted by the upper performance bound of 83% accuracy.

The EDS player makes use of a fixed set of algorithm parameters in a number of different learning configurations. The fixed parameters are the priority, where both Rainy and Fine have the same priority, the set of initial action values, V = {1000, …, 1000} , the maximum action value, m+ = 1500, and the minimum action value, m- = 500. The size of the script was fixed at 2, which would always include all of the available actions. The episode length was restricted to 1. The only parameter varied across runs for the EDS learner is the type of state abstraction. State abstraction for EDS in this game takes one of three forms: single, manual, and automatic. By default, only a single abstract game state is considered. This is illustrated in the Simbionic behavior shown in Figure 26.

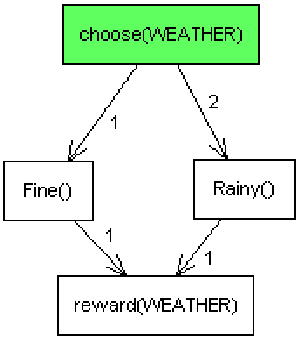


Figure 26 EDS in Weather with a single abstract state.

In the manual configuration, the behavior author uses predicate nodes in SimBionic to select one of three different choice points for the same decision. This is shown in Figure 27, which makes use of Card 1 and Card 4 – the two strong state predictors. One choice point includes the case where Card 1 is present and Card 4 is not, another where Card 4 is present and Card 1 is not, and a final choice point that includes all other states. This abstract state configuration is meant to make use of some, but not all, of the knowledge about how the game works. Based on this set of abstract states, the maximum expected prediction accuracy is roughly 70%.

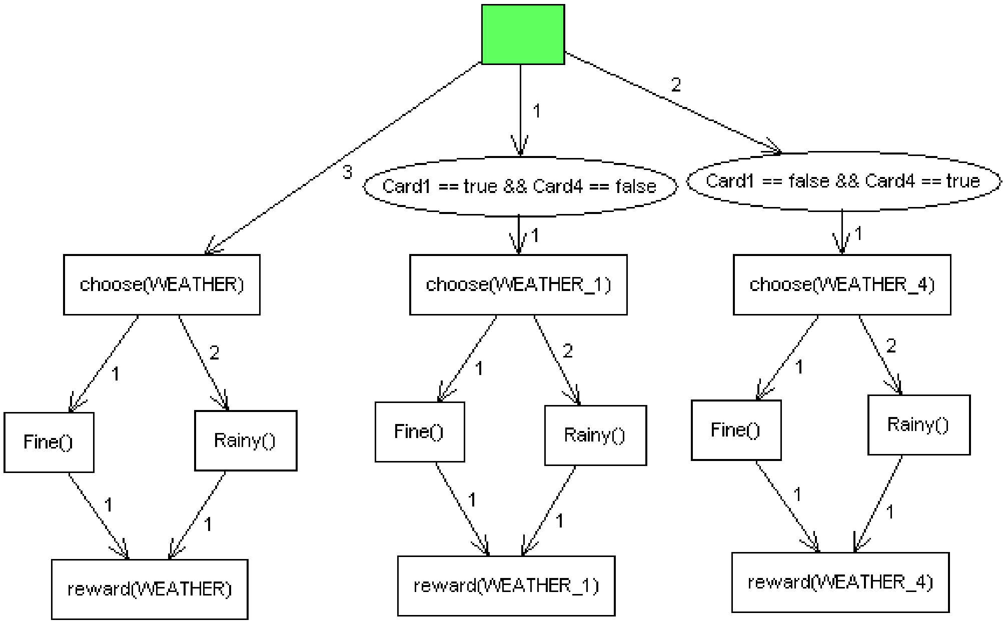


Figure 27 EDS in Weather with manual abstract state.

The behavior of the third state configuration, automatic state abstraction, closely resembles that of manual except that the single set of action values has been replaced by a classification tree that contains a number of action value sets. With automatic state abstraction, building a classification tree was tested at three different points (after 5, 20, and 50 episodes) and with two different tree algorithms (J48 and Decision Stump). The maximum expected performance of Decision Stump is 70% (the same as the manual abstract state). J48 could theoretically be used to achieve the maximum accuracy of 83%.

The Q player makes use of the Q-learning algorithm, where the initial value of each action is set to 100. The Q player has a single configuration parameter that varies the state abstraction across game runs. The state abstraction parameter can be one of two values: none, or abstract. When state abstraction is set to none, the Q player makes use of a standard Q-learning table, where a separate set of action values exist for each game state. In the abstract configuration, the Q-learner learns the value of all action in a single abstract state, similar to EDS. The manual version of state abstraction was not tested in this Q-learner, for reasons discussed in the results section.

Results and Discussion

This section contains the results generated by running the different learners in the context of the Weather game. The x-axis contains the number of decisions that have been made in the game, from 1 to 100. The y-axis is the mean score of the player over the last 10 runs (i.e. size 10 window). The goal of the learner in all cases is to maximize the score received from playing the Weather game as quickly as possible. The score represents the percentage of correct predictions made by the learner. The maximum attainable score in this game is 83%, with a lower maximum score of 70% if only two or three game states are considered.

In all cases, the learners played the Weather game 20,000 times, with 100 decisions made each run.

The Q player results are primarily presented in Figure 28. This figure shows that the version of Q using a single abstract state (Q Abstract) performs the same as randomly selecting an action over the course of 100 decisions. This is primarily due to the large state-dependent nature of the two available actions. The standard table-based Q player (Q) shows significantly better learning performance, which is expected given the small size of the state space (n=14).

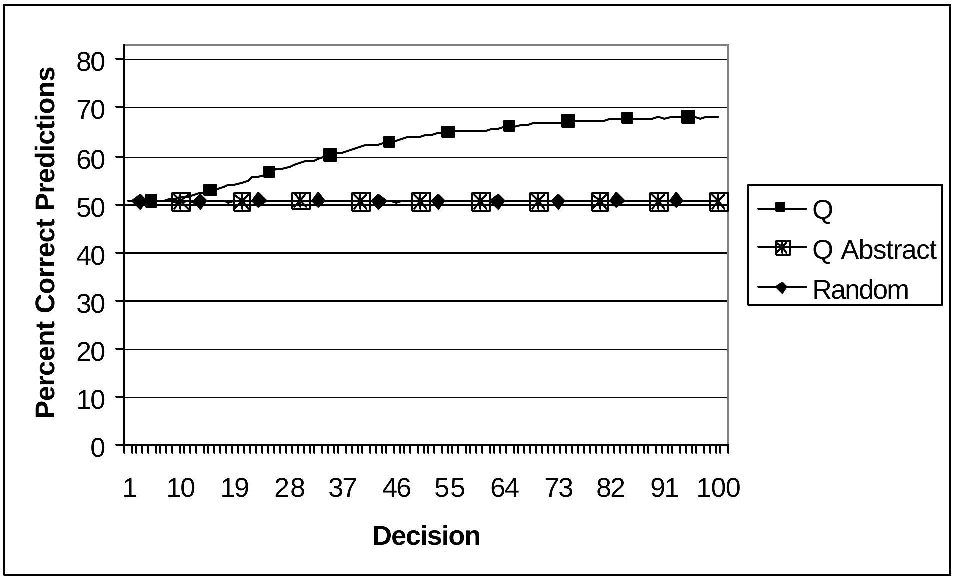


Figure 28 Q-Learner results.

The next graph, Figure 29, shows the results generated by EDS players on the same problem. All of the players have a dynamically generated script that contains five actions and are rewarded immediately after each decision. The two players tied for the lowest score are extended dynamic scripting and extended dynamic scripting with no learning, performing the same as the random player. This is improved upon by a version of extended dynamic scripting that learns based on a manual state abstraction created by a behavior author. The EDS Manual learner quickly achieves the maximum possible score given the three manually constructed abstract states.

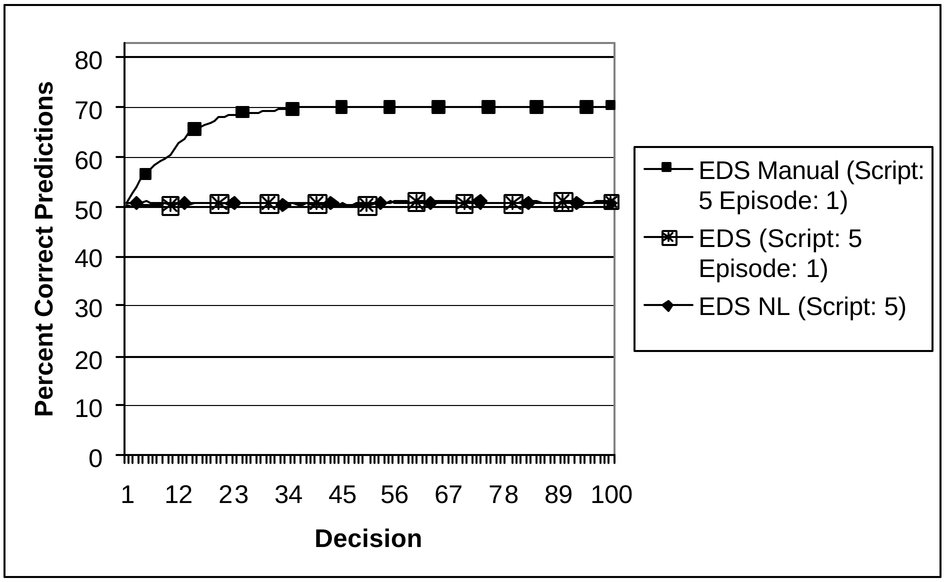


Figure 29 EDS with a script size of 5 and an episode length of 1 in three different configurations: manual state abstraction, no state abstraction, and with learning disabled.

The next four figures show the effect of automatic state abstraction. In all cases, a decision tree algorithm is used to build a classification tree based on the rewards received for actions in previous game states. Each leaf node in the classification tree represents a distinct abstract state, where each leaf node has its own set of action values. For example, a simple tree could be used to separate the game states where the current action values result in a positive reward from the games states where the current action values result in a negative reward. Two different types of decision tree algorithms are tested: Decision Stump and J48. Both of these decision tree algorithms are tested under the following two conditions. In the first, the classification tree is created once after 5, 20, or 50 decisions and then this tree is used to make rest of the decisions. In the second, a new tree and its corresponding leaf nodes are created every 5 or every 20 decisions.

In Figure 30, the extended dynamic scripting learner makes use of the Decision Stump algorithm to create a classification tree at three specific points. These results demonstrate that a larger history of decisions generates a better set of abstract states. That is, in Weather the Stump created after 50 decisions performs better than that created after 20 decisions, which in turn outperforms the Stump created after 5 decisions. As the amount of historic performance data increases, it becomes more likely that a Decision Stump can be created to partition the state space in a useful way.

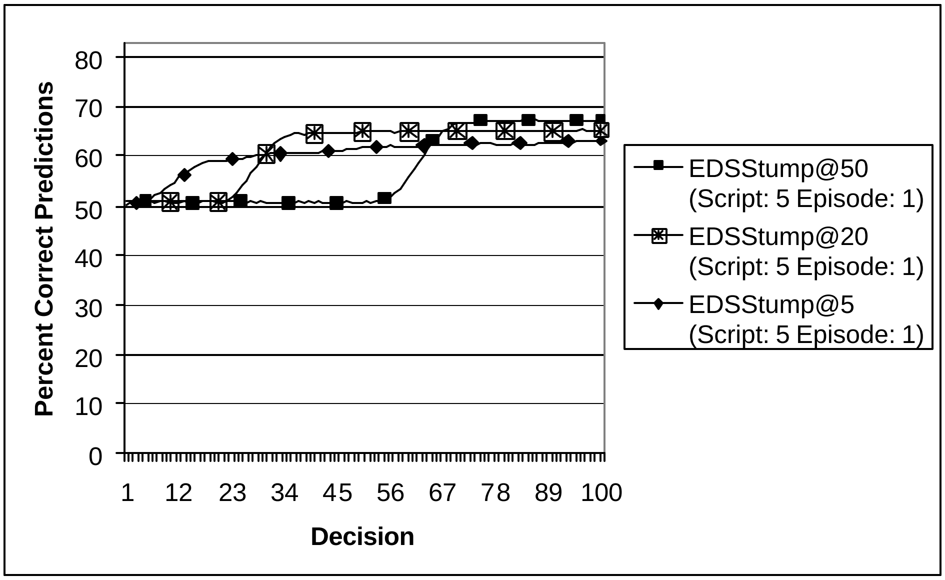


Figure 30 EDS with stump-based automatic state abstraction, where the tree is built once.

In Figure 31 a new classification tree (Decision Stump) is generated after every 5 or 20 episodes, the learner performs the same as if only a single tree was created after 5 or 20 episodes. The additional historic performance information does not greatly improve the capabilities of the learners with automatic state abstraction. If Decision Stump is made based on Card1 or Card4, the expected maximum performance is around 70%, and if based on Card2 or Card3 the maximum expected performance is around 55%.

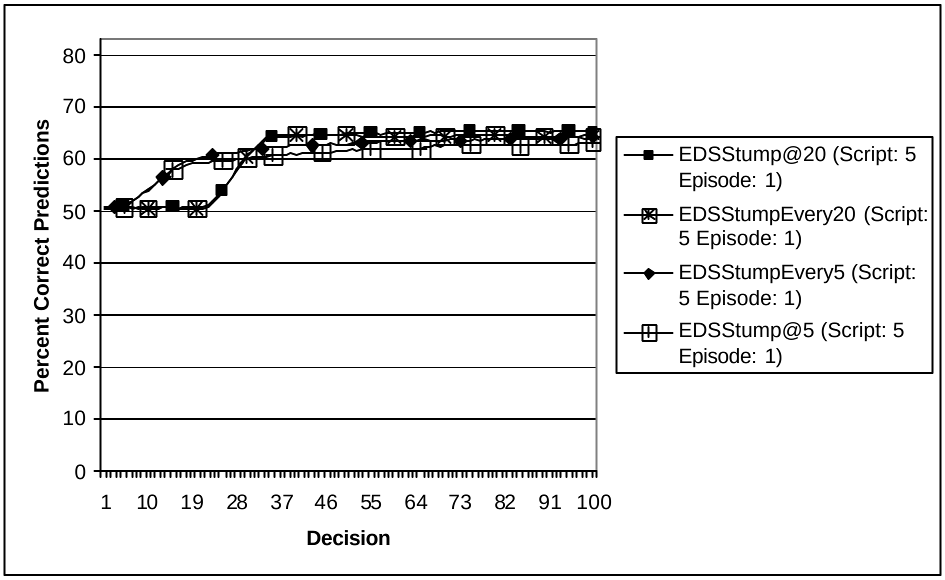


Figure 31 EDS with stump-based automatic state abstraction, rebuilding every *n* decisions.

Figure 32 presents the results generated by the same learners utilizing the J48 decision tree algorithm. The tree produced after 50 decisions, with the most historic performance information, performs better overall than the tree formed after 20 decisions and better than the tree formed after only 5 decisions. Figure 33 compares these results with those generated by constructing J48 trees after every 5 or 20 decisions. As found with Decision Stump, generating a single tree after 20 episodes and generating a tree every 20 episodes produces the same results. However, the learner that creates a new tree every 5 episodes does seem to be able to bootstrap. EDSTreeEvery5 performs quite a bit better than EDSTree@5.

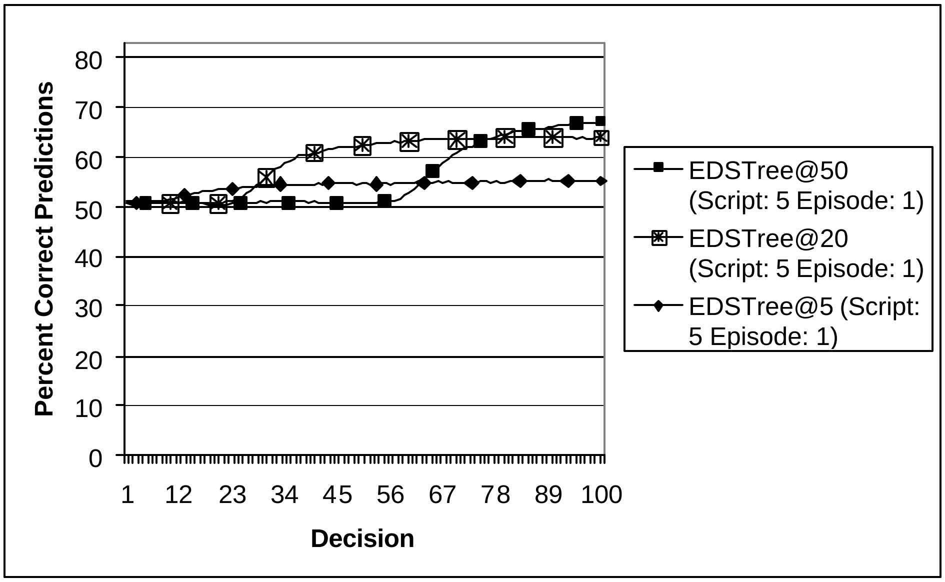


Figure 32 EDS with tree-based automatic state abstraction.

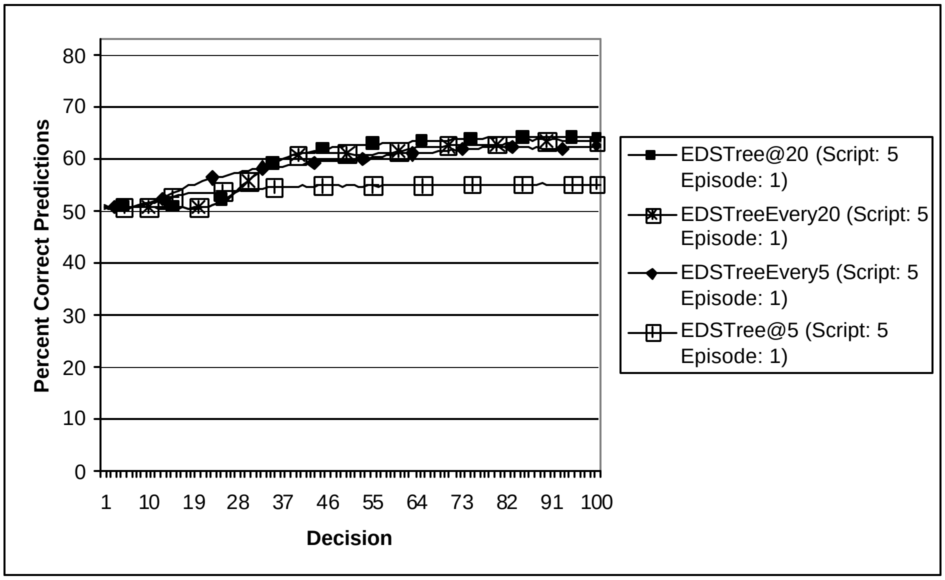


Figure 33 EDS with tree-based automatic state abstraction, rebuilding every *n* decisions.

The final graph provides an overall comparison of the different learners, with upper and lower bounds by which to measure their capabilities. Figure 34 presents the table-based Q player and multiple EDS players (EDS Manual, Stump@50, StumpEvery5, and EDS). The clear winner in this graph is Q, which does not perform as well as EDS Manual but does not require the behavior author to manually identify a state abstraction. That is, Q is learning the appropriate action to take in each of the 14 states. It is interesting to note that EDS Manual does perform quite well, relative to Q, with its abstract state space. Neither of the automatic state abstraction learners performs better than Q over the entire decision space. StumpEvery5 does initially scores higher than Q, but then lags behind for the last 60 decisions. Stump@50 does rise to the same performance level as Q, but only after 55 decisions.

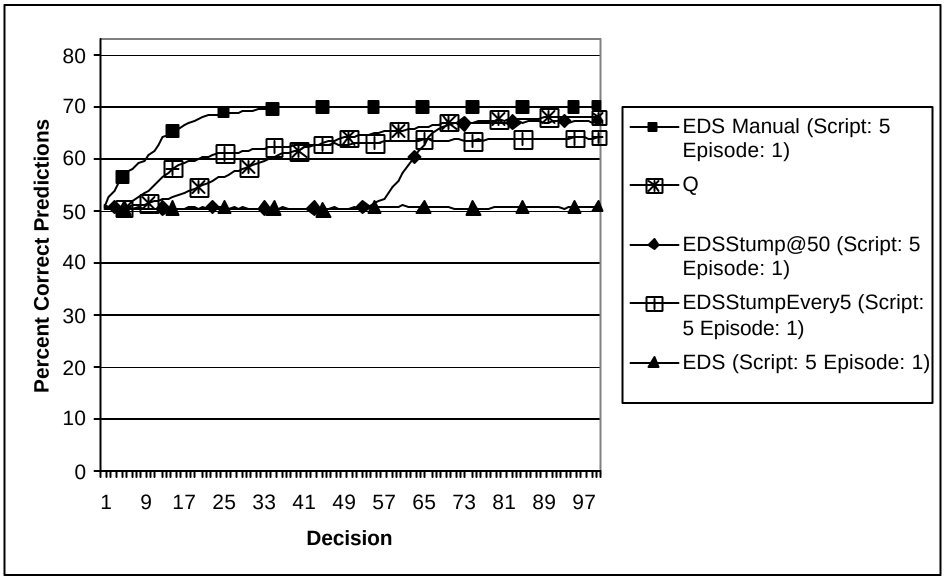


Figure 34 Overall comparison of weather results.

The primary conclusion drawn from the Weather domain is that in games where the effects of actions are primarily context-dependent, and the state space is relatively small, the standard table-based Q-learning algorithm would be expected to outperform extended dynamic scripting, unless additional knowledge is encoded in the form of abstract states. Without this knowledge, and without action priorities and large number of actions to choose from, extended dynamic scripting was still able to perform competitively with automatic state abstraction. Without manual or automatic state abstraction, EDS was unable to learn on this problem at all. These results generally match our expectations, though we were surprised to the degree by which a little bit of knowledge in the form of manually defined abstract states can improve the performance EDS. The combined results validate the extended dynamic scripting algorithm despite the fact that EDS was not the highest performer.

A secondary conclusion developed from working on this particular game is the importance of negative rewards for dynamic scripting when the script size is greater than or equal to the number of available actions. For example, let there be two actions, a and b, each with a value of 100 and the same priority. The two possible rewards are 10 and 0. The dynamically generated script, in order, will be either {a,b} or {b,a}. Now, assuming the former, if a reward of 10 is given to action a, we have a=105 and b=95 (due to the weight update algorithm). The next dynamically generated script will always be {a,b}, since the two actions have the same priority. At this point, the value of action a can only increase, because a reward of 0 will not change either action value and a reward of 10 will always increase the value of a more than b is increased. Since the value of action a is always greater than b, every script generated from this point on will always be {a,b}. No exploration is possible, because the size of the script is 2, so both rules will always be selected. The two solutions to this problem are to either provide negative rewards or to ensure that the script size is always less than the number of available actions.

Anwn

Anwn is an abstract role-playing game, based in part on the NeverWinter Nights computer game (Bioware, 2002). In Anwn, the player chooses from a wide array of possible actions, of which only a subset are applicable in the current game state. This represents the spells, special abilities, and combat tactics used by a player to attack the opposing forces in a role playing game like NeverWinter Nights, where the availability and effectiveness of actions are limited both by the players current state and the state (which may not be known) of the opposing agents.

Game Definition

The definition for the abstract version of a role-playing game starts with the sets *F, A,* and *R.* The game contains ten observation features, two of which are integers and the rest boolean. There are forty actions available to the player. The first ten can be roughly grouped as good (lower applicability, higher reward), the middle twenty as medium (higher applicability, lower reward), and the last ten as poor (greatest applicability, lowest reward). Generally, the actions have little dependence on the observation state, except for five state-dependent rules.

- *F* = {Health: Integer 0-60, Enemies: Integer 1-4, F3 :Boolean, F4 :Boolean, F5 :Boolean, F6 :Boolean, F7 :Boolean, F8 :Boolean, F9 :Boolean, F10 :Boolean}
- *A = {a1 … a 40}* where *a* = {< *O*, p, r-, r+, g >, < *O*', p', r-', r+', g'>, < *O*'', p'', p-'', p+'', g'' > …}
    - *a1* = {<*O'*, 0.1, 0 to -50, 0 to 100, 0.4>, <*O''*, 0.99, 0 to -50, 0 to 100, 0.4> }
        - O' = O – O''
        - O'' = F3 = true
    - *a2* = {<*O'*, 0.1, 0 to -50, 0 to 100, 0.4>, <*O''*, 0.99, 0 to -50, 0 to 100, 0.4> }
        - O' = O – O''
        - O'' = F3 = false
    - *a3 – a4* {<*O*, 0.7, 0 to -50, 0 to 100, 0.1> }
    - *a5 – a10* = {<*O*, 0.4, 0 to -50, 0 to 100, 0.1> }
    - *a11* = {<*O'*, 0.2, 0 to -50, 0 to 100, 0.4>, <*O''*, 0.8, 0 to -50, 0 to 100, 0.4> }
        - O' = O – O''
        - O'' = F3 = true
    - *a12* = {<*O'*, 0.2, 0 to -50, 0 to 100, 0.4>, <*O''*, 0.8, 0 to -50, 0 to 100, 0.4> }
        - O' = O – O''
        - O'' = F3 = false
    - *a13 – a20* = {<*O*, 0.5, 0 to -50, 0 to 100, 0.4> }
    - *a20 – a30* = {<*O*, 0.3, 0 to -50, 0 to 100, 0.4> }
    - *a31* = { <*O''*, 0.6, 0 to -50, 0 to 100, 0.6>, <*O'''*, 0.1, 0 to -50, 0 to 100, 0.6> }
        - O'' = F3 = true
        - O''' = F3 = false
    - *a32 – a40* = {<*O*, 0.4, 0 to -50, 0 to 100, 0.6> }

- *R* =
  - For actions a1 and a2, the Health and F3 state attributes are randomly updated
  - For all other actions: two state attributes are selected at random and randomly updated

Player Definition

We define a number of different players for this game: (1) Random, (2) EDS and, and (3) Q. The Random player selects an applicable action each time a decision is required. This player is used to provide a lower bound on the results expected from the other two players.

The EDS player makes use of a fixed set of algorithm parameters in a number of different learning configurations. The fixed parameters are the priority, I = {High: a1-a10, Medium: a11-a30, Low: a31-a40}, the set of initial action values, V = {100, …, 100} , the maximum action value, m+ = 2000, and the minimum action value, m- = 0. The EDS learner also contained a number of parameters that varied across game runs. These included the script size, n = 5 or 10, the length of episodes, e = 1 or 10, and the type of state abstraction. State abstraction in EDS takes one of three forms: single, manual, and automatic. By default, only a single abstract game state is considered. This is illustrated in the Simbionic behavior shown in Figure 35.

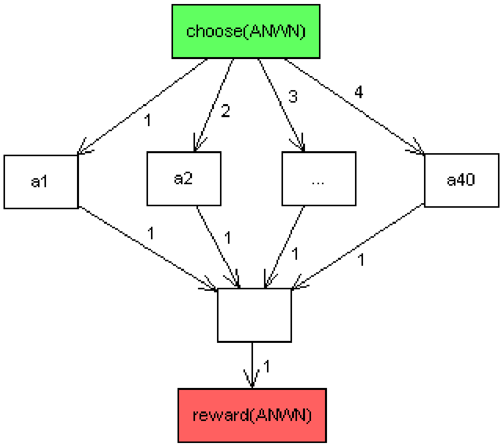


Figure 35 EDS in Anwn with a single abstract state.

In the manual configuration, the behavior author uses predicate nodes in SimBionic to select one of two different choice points for the same decision. This is shown in Figure 36. The behavior of the third state configuration, automatic state abstraction, closely resembles that of manual except that the single set of action values has been replaced by a classification tree that contains a number of action value sets. With automatic state abstraction, building a classification tree was tested at three different points (after 5, 20, and 50 episodes) and with two different tree algorithms (J48 and Decision Stump).

The Q player makes use of the Q-learning reinforcement learning algorithm as described previously. The Q player also has a number of configuration parameters that vary across game runs. The first parameter is whether the initial action values are uniform or biased. Biasing initial action values is one way that a behavior author can enhance initial learning performance in reinforcement learning, and was added as an analog to the priorities assigned in EDS. When uniform, all actions have the same initial value. When biased, initial action values are adjusted slightly based on the priority assigned to an action by the extended dynamic scripting algorithm. The first 10 actions are given a slightly higher initial value, the middle twenty remain the same, and the last 10 receive a slightly lower initial value.

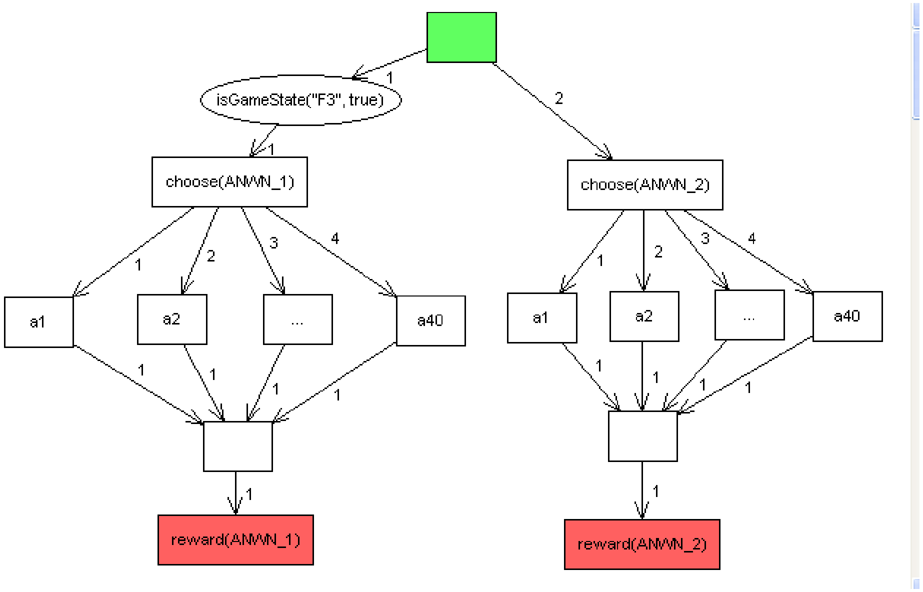


Figure 36 EDS in Anwn with manual abstract state.

The Q player also has a state abstraction parameter that can be one of three values: none, abstract, or manual. When state abstraction is set to none, the player makes use of a standard Q-learning table, where a separate set of action values exist for each game state. In the abstract configuration, the Q player learns the value of all action in a single abstract state, similar to EDS. In the manual configuration, the player learns the value of actions using a manually created state abstraction based on the value of the observable feature "F3". This represents domain knowledge that could be encoded by the behavior author as shown in Figure 36.

Results and Discussion

This section contains the results generated by running the different learners in the context of the Anwn game. The x-axis contains the number of decisions that have been made in the game, from 1 to 100. The y-axis is the mean score of the player over the last 10 runs (i.e. size 10 window). The goal of the learner in all cases is to maximize the score received from playing the Anwn game as quickly as possible. While the score has no intrinsic meaning, it is a valuable measure in comparing the relative performance of the different learners. In all cases, the learners played the Anwn game at least 10,000 times, with 100 decisions made each run.

The Q results are primarily presented in figures Figure 37 and Figure 38. In the first figure (Figure 37), we can see that standard, table-based Q player performs the same as randomly selecting an action over the course of 100 decisions. This is primarily due to the large state space (n = ~60,000) – it is unlikely that the same state will be visited multiple times in such a short number of decisions so no learning occurs. Both Q Abstract (a single game state) and Q Manual (two game states) show significantly better learning performance, with Q Manual performing the best overall.

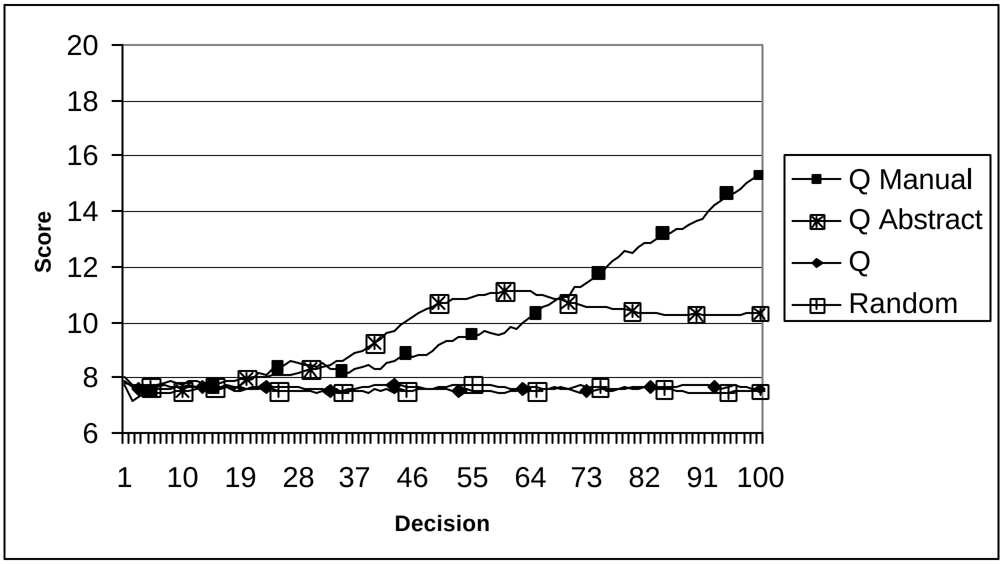


Figure 37 Unbiased Q-Learner results.

The second graph (Figure 38) presents the same learners, but this time all of the initial action values were biased according to the priority assigned by the dynamic scripting algorithm. This means that "good" actions would be more likely to be selected, "medium" less likely, and "poor" least likely. The priority-based bias improves the learner performance immediately, and is most noticeable for the table-based Q player. Initial performance is also improved for Q Abstract and Q Manual, where it takes nearly 35 decisions for the unbiased learners to catch up with the biased learners. However, the initial bias is unlearned over time and appears to have little effect on learning after this initial period.

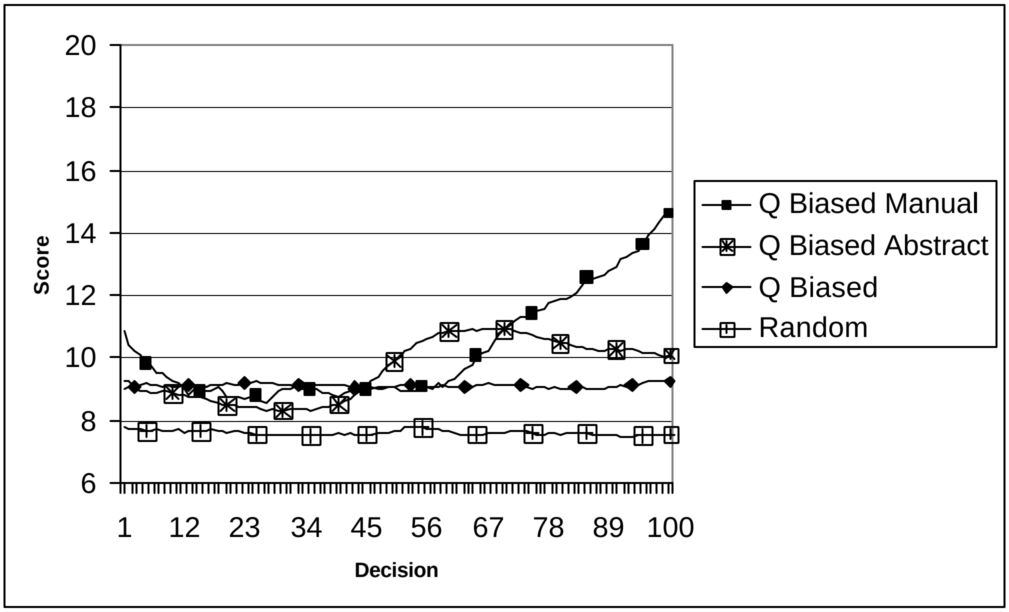


Figure 38 Biased Q-Learner results.

The next two graphs, Figure 39 and Figure 40, show the results generated by the EDS players on the same problem. In Figure 39, all of the players have a dynamically generated script that contains 5 actions. The lowest scoring EDS version has value learning turned off. This is improved upon by two versions of EDS that learn in a single abstract state. The two best learners in this figure are EDS with manually created abstract states, with the single episode learner performing best of all.

For the two versions of EDS in a single abstract state, the player with the episode length of one performs significantly better than the player with an episode length of ten. This is due to the problem of credit assignment, and this pattern will be repeated numerous times in the EDS results. Supplying an immediate reward for a single action selected with a script results in faster learning than supplying a combined reward generated from selecting 10 actions with the same script. For example, if action *a1* is selected with a reward of 50, each action in the set of completed actions (which consists only of *a1)* will receive a reward of 50. If the episode length is two, with actions *a1* and *a2* selected and corresponding rewards of 50 and -30, the picture is quite different. Now the set of completed actions contains *a1* and *a2*, and each action receives a reward of 20.

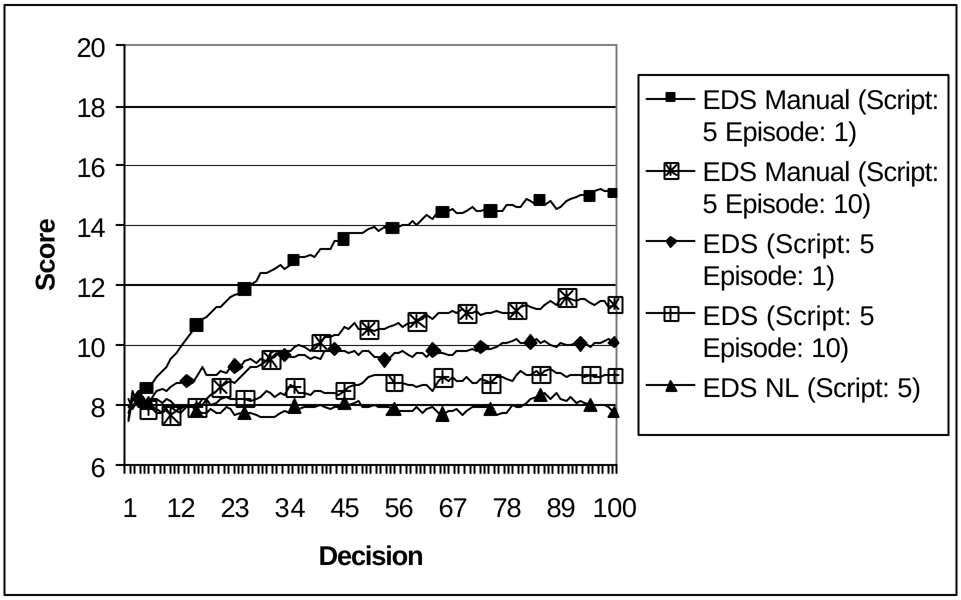


Figure 39 EDS with script size of 5 and varying episode length. The bottom series (EDS NL) has learning disabled.

Figure 40 presents results form the same learners, except this time size of the dynamically generated script has been doubled to contain ten actions. That is, the script now includes ten of the available actions (IF/THEN rules) rather than 5. The initial scores generated by the single state learners are nearly the same as the final scores generated by the learners with the smaller script size. In general, the trends seen in the previous graph are also seen in the graph: manual state abstraction performs better than learning in a single abstract state and immediate learning performs better than episodic learning. From looking at the greatly improved results, the extended dynamic scripting algorithm appears to be very sensitive to the script size.

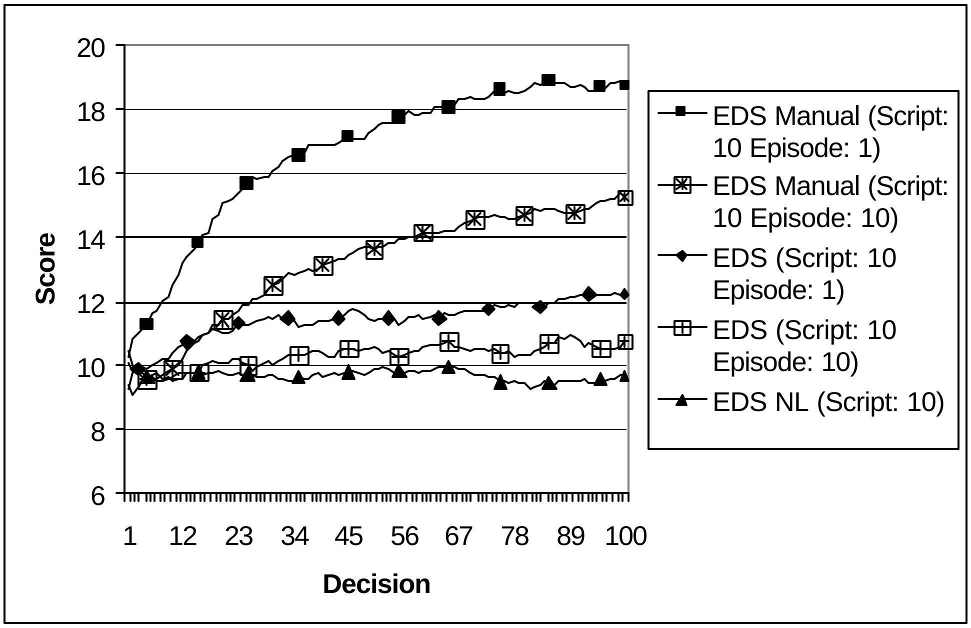


Figure 40 EDS with script size of 10 and varying episode length. The bottom series (EDS NL) has learning disabled.

Building from the best standard EDS configuration, the next four figures show the effect of automatic state abstraction. In all cases, a decision tree algorithm is used to build a set of abstract states for the single choice point. Each state is represented in a classification tree as a leaf node, where each leaf node has its own set of action values. Two different types of state abstraction are tested: Decision Stump and J48. Both of these decision tree algorithms are tested under the following two conditions. In the first, abstract states are created once after 5, 20, or 50 decisions and then this tree is used to make rest of the decisions. In the second, a new tree and its corresponding leaf nodes are created every 5 or every 20 decisions.

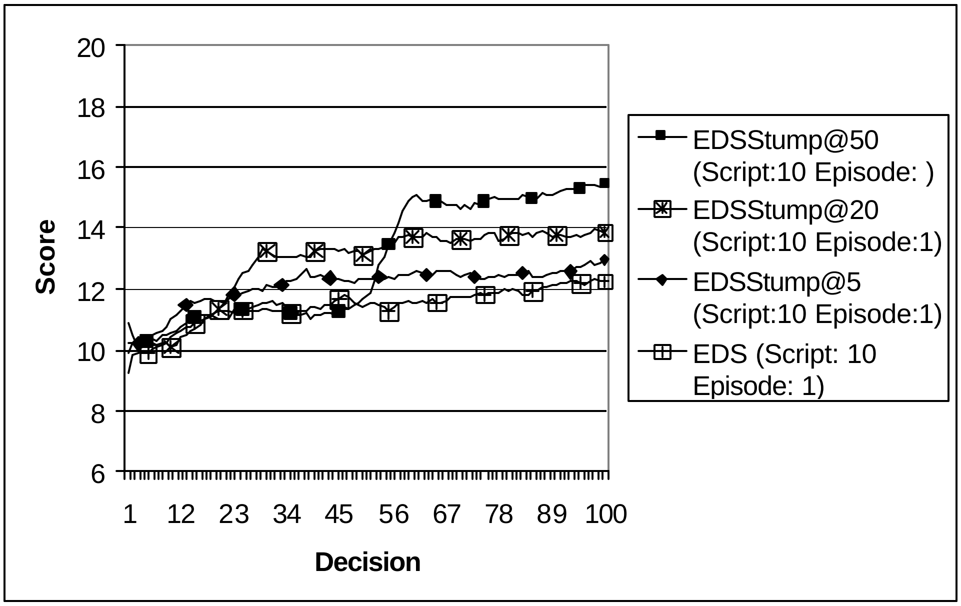


Figure 41 EDS with stump-based automatic state abstraction.

In Figure 41, the EDS player makes use of the Decision Stump algorithm to create a classification tree. Here is one example of a decision stump, which divides the state space into three abstract states depending on the value of the observation feature F3:

```
F3 = false : 25.833333333333332
F3 != false : 2.1875
F3 is missing : 10.7
```

The results in this graph demonstrate that a larger historic data set generates a better set of abstract states, where the historic data is a sequence of tuples of the form <<game state>, reward received>. This finding is validated in Figure 42, where the learner that rebuilds the decision tree every five decisions, EDSStumpEvery5, is shown to outperform the other EDS learners with automatic state abstraction.

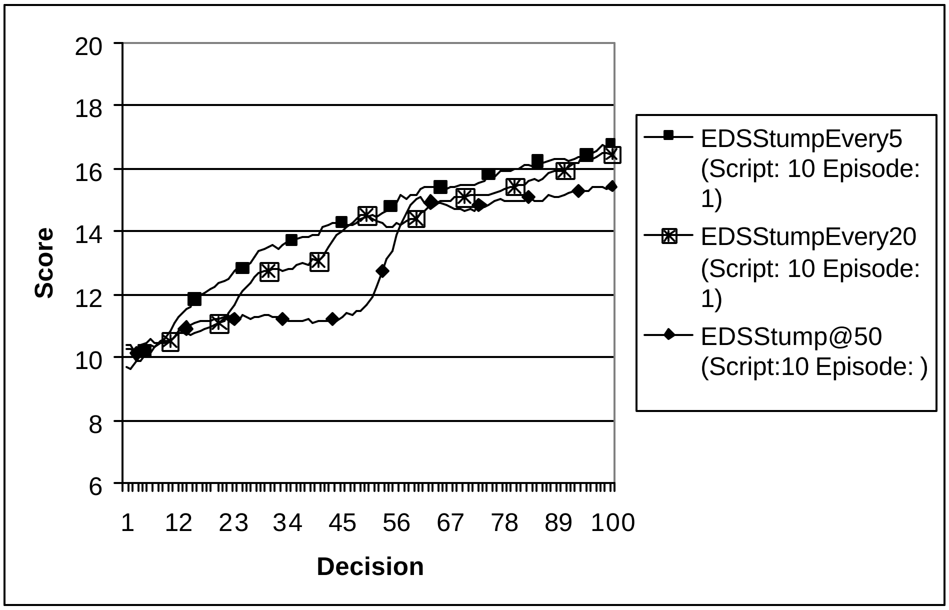


Figure 42 EDS with stump-based automatic state abstraction, rebuilding every *n* episodes

Figure 43 presents the results generated by the same learners utilizing the J48 decision tree algorithm. Here is an example of an automatically generated tree, which creates abstract states based four different features:

```
F6 = true
|   F5 = true
|   |   Enemies <= 2
|   |   |   F3 = true: '(67.6-inf)' (4.0/2.0)[0]
|   |   |   F3 = false: '(-17.6-10.8]' (8.0/3.0)[1]
|   |   Enemies > 2: '(-inf--17.6]' (3.0/1.0)[2]
|   F5 = false: '(10.8-39.2]' (3.0/1.0)[3]
F6 = false: '(10.8-39.2]' (2.0)[4]
```

The learner that creates a tree after 20 decisions (EDSTree@20) outperforms the other EDS learners with J48 state abstraction, with a higher score for most of the decisions and an end score similar to that of creating a tree after 50 decisions. In this case, the simpler tree produced after 20 decisions, with fewer instances, performs better overall than the tree formed after 50 decisions. Unlike what was seen with the Decision Stump algorithm, more training instances did not result in better abstract trees. The likely cause for this is that in the Anwn game, the performance of actions primarily depends upon the value of the F3 attribute. While both the Decision Stump and J48 algorithms usually identify this attribute, the trees created by J48 generally make use of a number of additional attributes. With only this relatively small amount of historic data, the J48 algorithm is not able to determine that the only interesting attribute is F3.

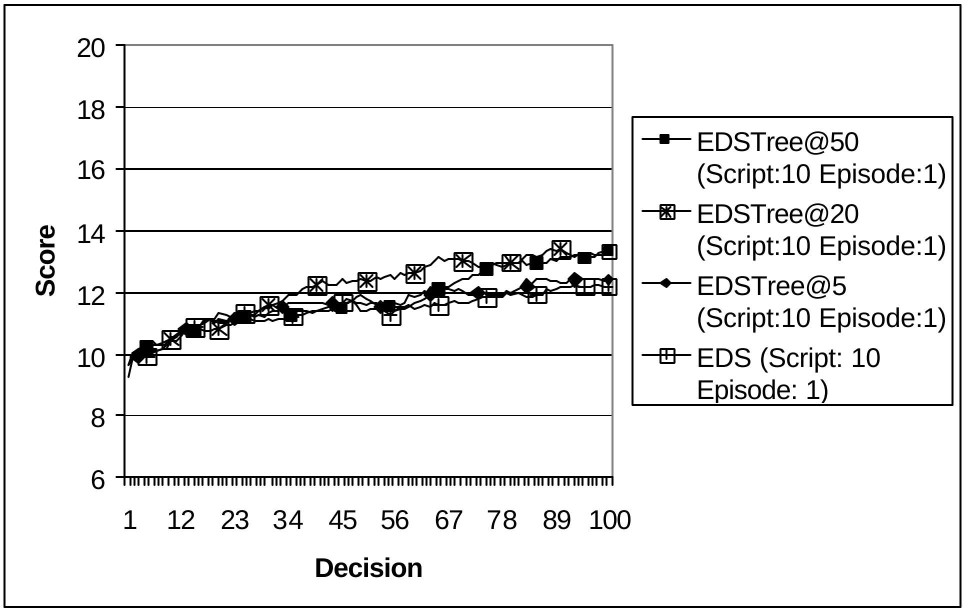


Figure 43 EDS with tree-based automatic state abstraction.

Figure 44 shows that building and training a new decision tree from the historic data every 5 or 20 decisions does not improve performance above what was found when creating a tree one time after 20 decisions. These results are contrary to the results generated by the Decision Stump, where performance improves over time. One explanation is that in the case of the Decision Stump, creating and training a new state abstraction tree resulted in a tree very similar to the previous one in the case where F3 had been identified, or a tree where F3 was more likely to be identified when it previously was not. This would account for improved performance over time. This does not seem to be the case for the J48 derived trees, where more historic data does not result in the creation of more accurate trees. Additionally, as can be seen for EDSTreeEvery20, creating and training a new state abstraction tree generally results in an immediate, small, performance drop due to the approximate nature of the training algorithm.

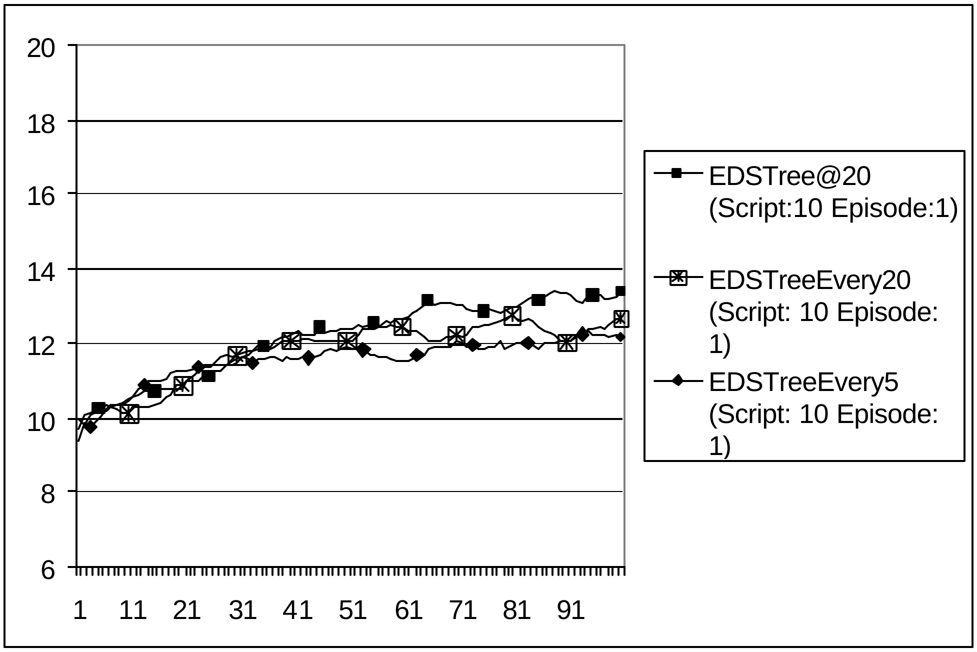


Figure 44 EDS with tree-based Automatic state abstraction, rebuilding every *n* episodes.

In general, the Decision Stump algorithm outperforms J48 in every configuration. Overall, these four graphs demonstrate the relative utility of the Decision Stump algorithm over J48 in the type of environments where dynamic scripting is expected to excel.

The final two graphs compare the performance of the best learners in two different situations, providing upper and lower bounds by which to measure the capabilities of the automatic state abstraction extension. Figure 45 presents two Q players and two EDS. In this graph, it is clear that EDS Manual is able to learn much more quickly than either of the Q players with the same state abstraction. The EDS player that makes use of automatic state abstraction (StumpEvery5) performs almost as well as the EDS Manual, outperforming the Q-learners for almost all of the decisions.

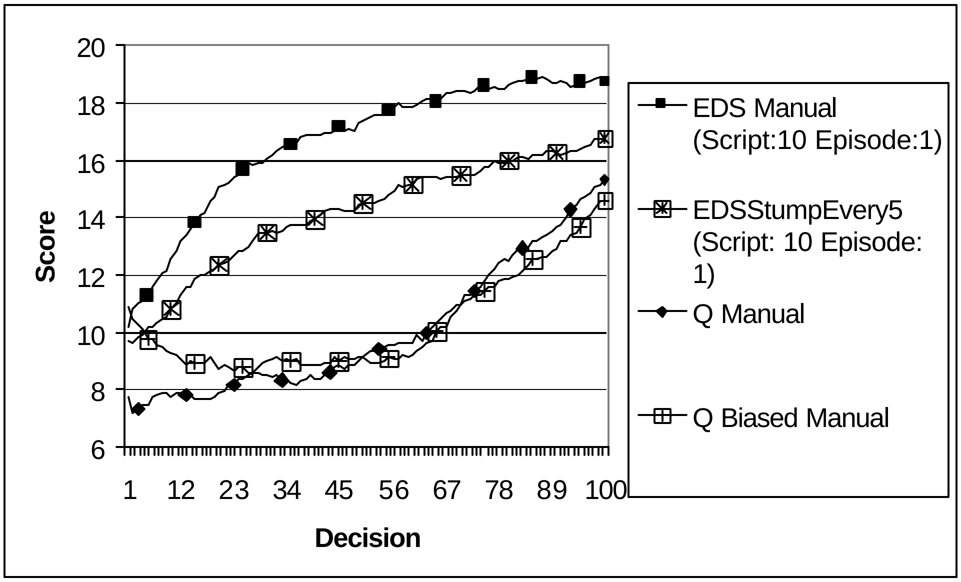


Figure 45 Comparison of best manual and automatic state abstraction results.

In Figure 46, results are presented for the Q and EDS players that use only a single abstract state. Their performance is compared to the EDS player that includes automatic state abstraction (StumpEvery5). This figure provides additional evidence for the utility of automatic state abstraction, as the StumpEvery5 learner outscores both of the other learners with no additional knowledge from a behavior author.

The primary conclusion drawn from the Anwn domain is that in highly-stochastic games where the effects of actions are primarily context-independent, the extended dynamic scripting algorithm should demonstrate higher learning efficiency (i.e. faster learning) than would be found with either standard dynamic scripting or Q-based algorithms. The results from this game provide evidence for the efficacy of the extensions to dynamic scripting undertaken in this dissertation. A secondary conclusion from these results is that in addition to being sensitive to the reward function (a common issue in reinforcement learning) the extended dynamic scripting algorithm is also sensitive to the size of the dynamically generated script (i.e. the number of available actions).

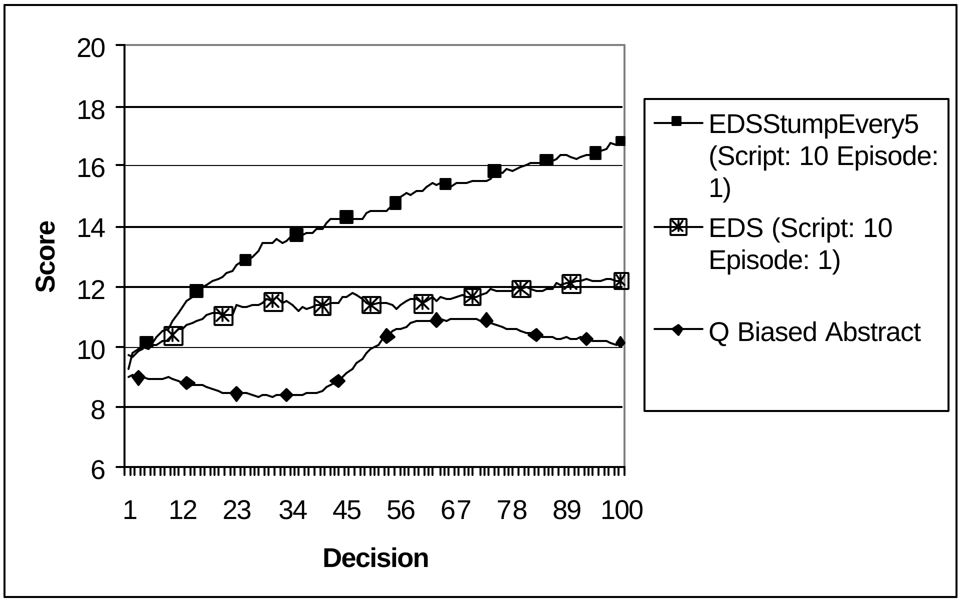


Figure 46 Comparison of best single state and automatic state abstraction results.

These results, combined with the Weather results, also demonstrate that the effect of automatic state abstraction is heavily dependent on the amount of historic data, but provides little guidance on how to determine this a priori. A preliminary conclusion to be drawn with respect to automatic state abstraction seems to be that it is better to start creating decision trees early in the learning process and to continue to create trees as learning progresses if the goal is to achieve maximum performance throughout the learning process. These results also demonstrate the Decision Stump trees are a better choice for use with EDS than the more complicated trees produced by J48.

Performance Results

Of the four abstract games Anwn is the most complex, both in terms of the number of actions available and the size of the state space. Computational performance results were gathered for this game in order to represent the relative cost of the EDS players compared to the other players.

Two specific metrics were gathered for the Random, Q, and EDS players. The first was the amount of time to select a single action. This was calculated by selecting 10,000 actions in a row, in order to ensure that enough time passed to be captured in milliseconds. The second metric was the amount of time to perform an action selection, update the game state, and then update the action values. This sequence was also performed 10,000 times in a row. In both cases, the value was divided by 10,000 to determine an average time for a single selection or selection + udate. These metrics were gathered on a single computer in order to compare relative times. The absolute times are only representative.

For the EDS players that incorporate automatic state abstraction, a slightly different tactic was used. First, for the action selection metric the player was run for 100 selection + update sequences to generate a set of abstract states prior to the 10,000 selections. This ensured that the test results included selecting an action from an Decision Stump or J48 tree. Second, for the action selection + update metric, rather than running 10,000 selection + update sequences we ran 100 selection + update sequences 100 times. The reason for this is that as the amount of historic data increases so does the update time. In order to capture the relative performance as would be seen in the Anwn game, the updated measurement provides the mean performance as the set of historic instances as it grows from 5 to 100.

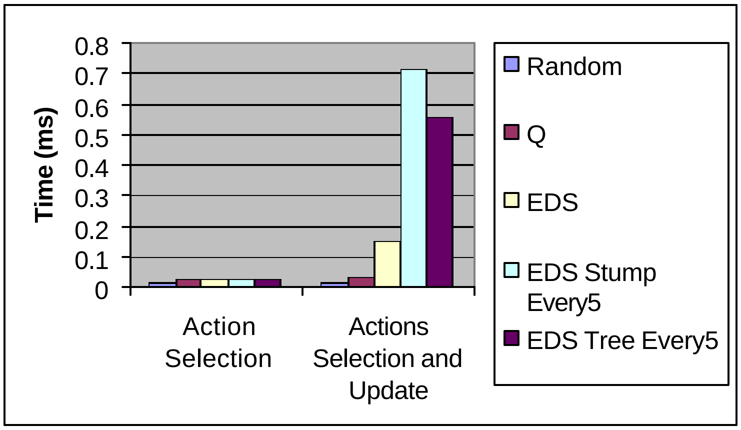


Figure 47 Relative performance for five different players when selecting a single action or when selecting and performing an action followed by updating the action values.

As shown in Figure 47, there is little difference in the time required to select an action across the five different learners. The cost of updating the action values does differ considerably across the different learners. First, EDS takes approximately 4-5 times as long to update the action values as does the Q-learning algorithm. As the numbers are still quite small, this does not seem a cause for concern. The EDS players equipped with automatic state abstraction pose more of a problem, taking roughly 20 times longer to perform action value updates. Most of the additional time is actually being spent every five episodes, when the tree is rebuilt and trained. This means that, on average, the update every fifth episode would take 100 times longer. Earlier updates would consume less time and later updates consume more time, as the amount of historic data grows.

The important result from this is that performance could become a problem if automatic state abstraction is used. If the amount of historic data is small, this would only become apparent if a large number of players (e.g. 1000) were all required to make 30 decisions per second. Performance would also become an issue as the size of the individual historic instances grows (i.e. feature vector length) or as the number of historic instances increase. Future research on algorithms for automatically determining when to initiate the creation and training of abstract state trees should take current performance metrics into account. Additionally, it would be useful for the creation and training of abstract state trees to be performed in a separate, lower priority, thread so that their construction does not impact the player's ability to select an action on demand.

A secondary issue is that the current, non-optimized, method of importing feature vectors into Weka involves writing them out to a comma separated file and then reading them in with Weka. As this step could be avoided through a straightforward, but tedious, optimization, the time for reading/writing files was not included in the results. However, if EDS with automatic state abstraction is used in a production environment, then this optimization will need to be carried out.

Get the Ogre!

The Get the Ogre (GtO) game is an abstract version of common problem found in real-time strategy games, previously examined by (Marthi et al., 2005). In GtO, an Ogre exists just outside of the base camp and the goal of the player is to build up a group of soldiers to dispatch the Ogre. In order to do succeed, the player must create resource gathering “peasants”. The gathered resources can then be used to build farms (to support the peasants), more peasants (to gather more resources), or soldiers. The player must also decide how many soldiers are required to efficiently dispatch the Ogre.

With too few soldiers there is a risk that the Ogre wins; too many soldiers are a waste of resources.

Game Definition

GtO is defined by the sets *F, A,* and *R.* The game contains four integer observation features. There are seven actions available to the player. State transitions are not made randomly in this game, but instead depend explicitly on the action selected.

- *F* =
    - Food: Integer (Initial value = 4)
    - Wood: Integer (Initial value = 100)
    - Gold: Integer (Initial value = 0)
    - Soldiers: Integer (Initial value = 0)
- *A* =
    - BuildFarm (must have 100 Wood)
    - CreatePeasantWood (must have 1 Food)
    - CreatePeasantGold (must have 1 Food)
    - CreateSoldier (must have 100 Gold and 50 Wood)
    - AttackOgre3Soldiers (must have 3 Soldiers)
        - p = 0.4, r- = -100, r+ = 100
    - AttackOgre4Soldiers (must have 4 Soldiers)
        - p = 0.9, r- = -100, r+ = 100
    - AttackOgre5Soliders (must have 5 Soldiers)
        - p = 0.9, r- = -100, r+ = 80
- *R* =
    - BuildFarm (+4 Food, -100 Wood)
    - CreatePeasantWood (+200 Wood, -1 Food)
    - CreatePeasantGold (+200 Gold, -1 Food)
    - CreateSoldier (-100 Gold, -50 Wood, +1 Soldier)

The GtO game requires that the player choose a number of actions before any reward is given. There are three different ways that a player could get a reward. The first is that they perform some combination of the actions that ends with one of the

AttackOgreXSoldiers actions. The reward provided in this case is based on the action as described above. If the player ever gets to a position where it is impossible to select another action (for example if Food, Wood, and Gold are all 0), the player receives a reward of -110. It is also possible for the player to get into an infinite loop. If the player exceeds 50 actions, a reward of -111 is given. An optimal action sequence is:

CreatePeasantGold (3F 100W 200G 0S),
CreatePeasantGold (2F 100W 400G 0S),
CreatePeasantWood (1F 300W 400G 0S),
CreateSoldier (1F 250W 300G 1S),
CreateSoldier (1F 200W 200G 2S),
CreateSoldier (1F 150W 100G 3S),
CreateSoldier (1F 100W 0G 4S),
AttackOgre4Soldiers.

While an optimal sequence is shown, the reward a player receives is based on whether or not it can solve the problem, not on how quickly it solves the problem.

Player Definition

We define a number of different players for this game: (1) Random, (2) EDS, and (3) Q. The Random player selects an applicable action each time a decision is required and provides a performance baseline.

The EDS player makes use of a fixed set of algorithm parameters in a number of different learning configurations. The first fixed parameter is the action priorities:

- Higher { AttackOgre3Soliders, AttackOgre4Soliders, AttackOgre5Soliders }
- High { CreateSoldier }
- Medium { CreatePeasantGold }
- Medium { CreatePeasantWood }
- Low { BuildFarm }

The set of initial action values, V = {1000, …, 1000} , the maximum action value, m+ = 1500, and the minimum action value, m- = 0 are also fixed parameters. The size of the script varies in size from 4 to 7 actions. The episode length in this problem is also variable, as there are three different ways to end the game and receive a reward. The other parameter varied across runs for the EDS player is the type of state abstraction. State abstraction in EDS takes one of two forms: single and automatic. By default, only a single abstract game state is considered.

In the second state abstraction configuration, automatic, the single set of action values is replaced by a classification tree that contains a number of action value sets. With automatic state abstraction, building a classification tree was tested at two different points (after 5 and after 20 episodes) and with two different tree algorithms (J48 and Decision Stump).

Q, the third type of player tested in GtO, makes use of the standard Q-learning reinforcement learning algorithm, generating results by which the performance of the EDS player can be compared. The Q player has a number of configuration parameters that vary across game runs. The first parameter is whether the initial action values are uniform or biased. Biasing initial action values is one way that a behavior author can enhance initial learning performance, and is added as an analog to the priorities assigned in EDS. When uniform, all actions have the same initial value of 100. When biased, initial action values are adjusted slightly based on the priority assigned to an action by the extended dynamic scripting algorithm. The Q player also has a state abstraction parameter that can be one of two values: none or abstract. When state abstraction is set to none, the player makes use of a standard Q-learning table, where a separate set of action values exist for each game state. In the abstract configuration, the Q player learns the value of all action in a single abstract state, similar to EDS.

Results and Discussion

This section contains the results generated by running the different learners in the context of the GtO game. The x-axis contains the number of episodes, from 1 to 50, where each episode contains a variable number of decisions. The y-axis is the mean score of the player over the last 10 runs (i.e. size 10 window). The goal of the learner in all cases is to maximize the score received from playing the game as quickly as possible. While the score has no intrinsic meaning, it is a valuable measure in comparing the relative performance of the different learners. In all cases, the results are the average performance of 1000 learners.

Figure 48 summarizes the results from the different Q players. In this figure, the table-based Q-learner with biased initial weights (Q Biased) is the top performer, solving the problem roughly ½ of the time. This player is able to learn the proper actions in the states where Wood for Food is low, while maintaining a reasonable level of performance in less-visited states by use of the biased weights. The biased, single-state, Q player (Q Biased Abstract) starts off well, but then quickly drops off as the algorithm starts to unlearn the initial values. The unbiased, table-based Q player (Q) performs at the same level as the Random player, while the single-state version of Q (Q Abstract) performs even worse.

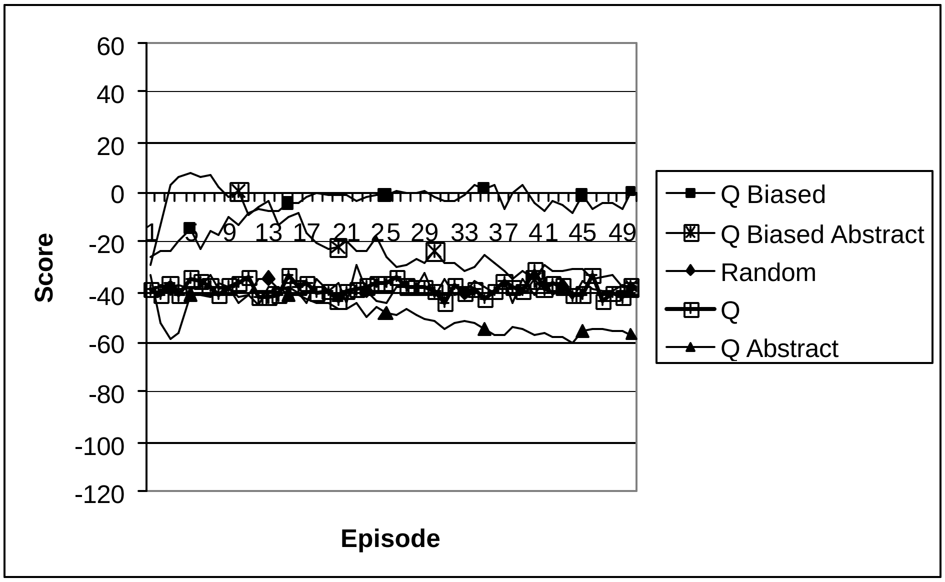


Figure 48 Q-Learner performance in Get the Ogre.

The performance of the EDS players is summarized in Figure 49. Regardless of script size, the EDS learners either do not solve the problem within the maximum number of steps (infinite loop) or get into a position where no more actions are possible (e.g. running out of Food and Wood at the same time). This is due to the type of scripts that can be generated from the given actions. For example, assume that the script contains exactly the four actions required to complete the problem: AttackOgre4Soldiers, CreateSoldier, CreatePeasantGold, CreatePeasantWood. A dynamically generated script, in priority and value order, might look like:

- AttackOgre4Soldiers (Priority: Higher)

- Create Solder (Priority: High)
- CreatePeasantGold (Priority: Medium; Value: 1100)
- CreatePeasantWood(Priority: Medium; Value: 900)

By following the script in order, the EDS player will continue to create gold until all the Food has run out. Since the script will never create the wood required, the soldiers cannot be built. If CreatePeasantGold and CreatePeasantWood have different action values, then only Wood will be created and never Gold.

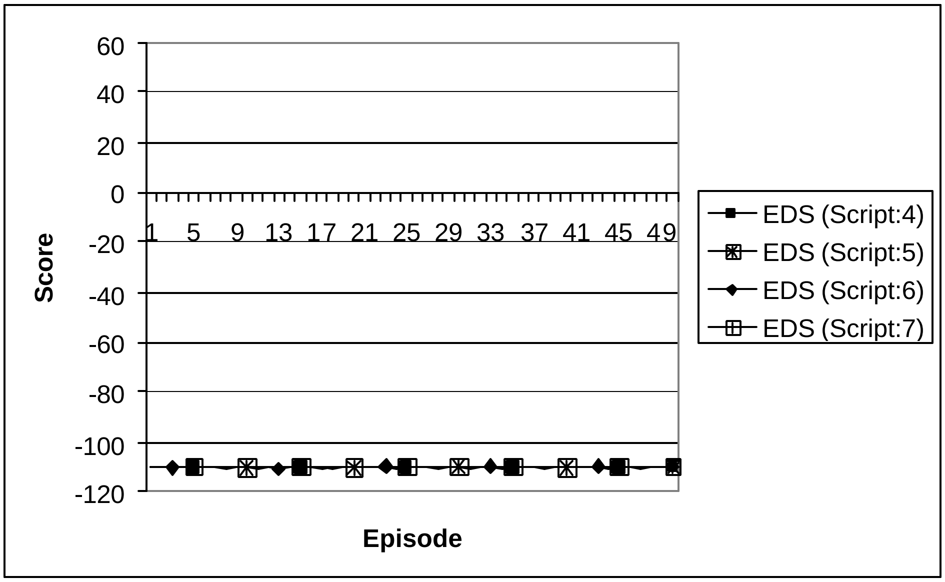


Figure 49 Performance of EDS with only a single abstract state.

The next two figures, demonstrate the utility of EDS with automatic state abstraction enabled. Figure 50 shows the performance of EDS when a decision stump is created after five episodes, across a variety of different script sizes. In this case, the classification tree is not rebuilt once it is created, though learning does occur in the leaf nodes. With the exception of the case where the script size equals seven (i.e. includes all of the available actions), the use of automatic state abstraction clearly improves performance. A typical decision stump classification tree looks like:

```
Wood <= 200.0 : -110.04347826086956
Wood > 200.0 : -110.93333333333334
Wood is missing : -110.48351648351648
```

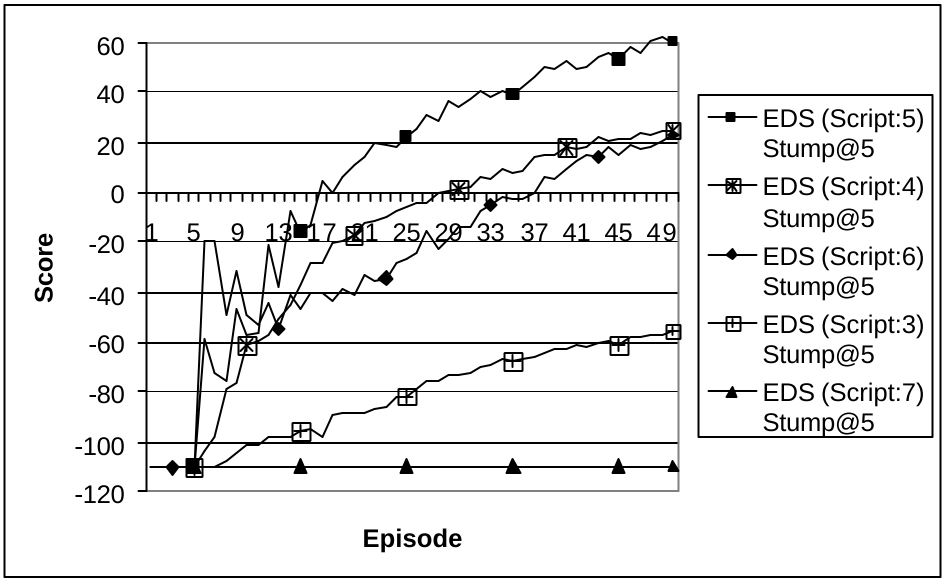


Figure 50 Performance of EDS when using stump-based automatic state abstraction at episode 5, with a variety of script sizes.

Figure 51 summarizes the best EDS performers utilizing J48 or decision stump-based state abstractions. In this figure, a classification tree is created one time, after either 5 or 20 episodes. In this particular game there is generally little difference in the trees created by the two different algorithms, or in the trees created at episode 5 vs. episode 20. The primary variable in determining learner performance is the amount of learning that has taken place after the tree has been created.

The effect of learning once the tree has been created is demonstrated in Figure 52. Learner performance is initially increasing after the construction of the first state abstraction tree. However, when the tree is created every 5 or 20 episodes performance quickly levels off, where the creating of trees actually lowers future performance. In the Every5 case, for both J48 and decision stump, subsequent trees are seen to perform either equivalent to, or less than, previous trees. This result is directly the opposite that we seen in the first two games.

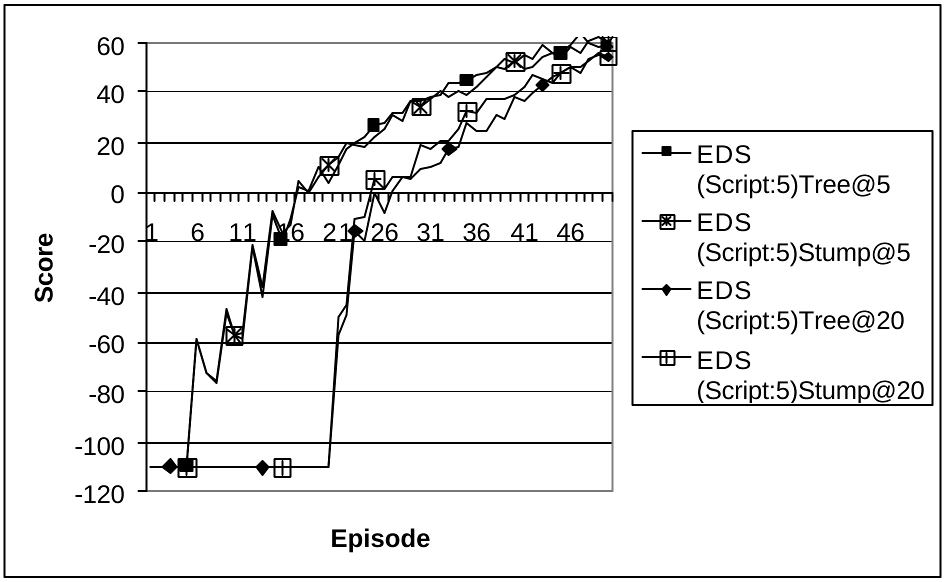


Figure 51 Best results of EDS with automatic state abstraction with a tree built at episode 5 or episode 20.

So, what is happening when the abstract state tree is created after an initial tree? Assuming a useful abstract state tree was initially created, all of the leaves are now receiving the exact same reward for the selected actions due to the episodic nature of the Get the Ogre game. The historic instances will contain the various game states identified by the abstract state tree – but all with the same reward. This means that when another tree is created it is likely that the abstract states, which are currently performing well with separate scripts, will get merged back together into a single leaf node; the feature vector consists only of the game state and the classification target is the reward received. This is not the case in both the Anwn and Weather games, where the leaves in the abstract state tree would be expected to generate distinct reward values such that particular abstract states could be maintained as abstract state trees are created.

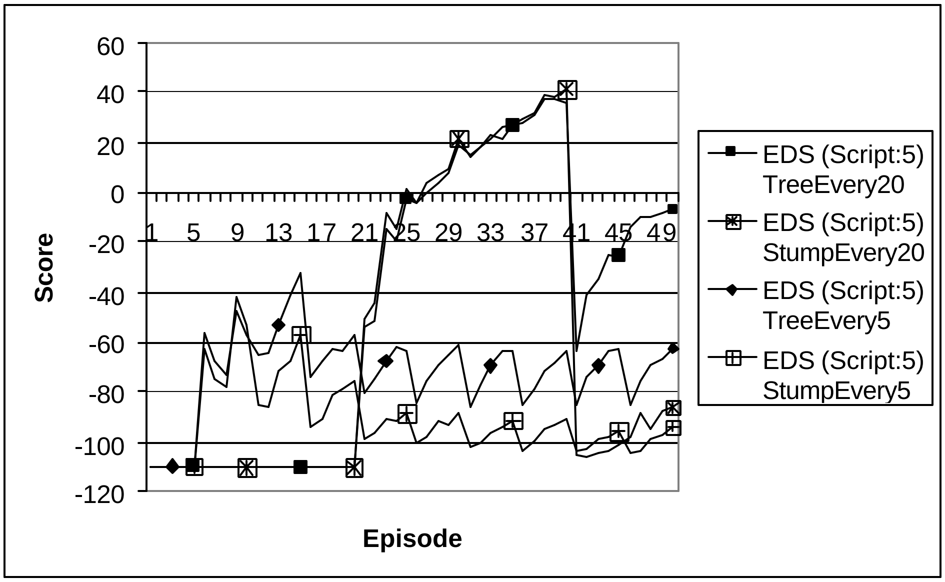


Figure 52 Best results of EDS with automatic state abstraction with trees built every 5 or every 20 episodes.

Figure 53 summarizes the best results for the Get the Ogre game. EDS, without state abstraction, is unable to learn on this problem at all. The best Q-learning solution is able to make quite a bit of headway, solving the problem a little more than ½ of the time. The best performance on this problem is found EDS with automatic state abstraction. While it may be that the performance of the Q player could be improved by the use of automatic state abstraction or by other techniques such as linear function approximation (Sutton & Barto, 1998), this critique does not detract from the main result: EDS can adequately solve a problem that could not be handled by the standard dynamic scripting algorithm.

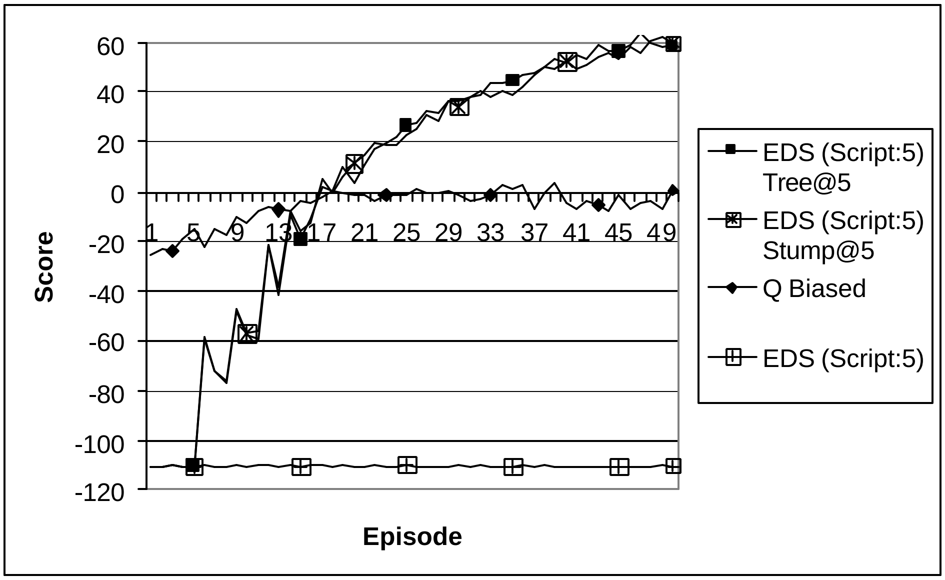


Figure 53 Overall comparison for Get the Ogre.

Modified Version of Get the Ogre

One common technique used when developing behavior models is to create new actions by adding constraints to existing actions. For example, a player can use the *heal* action at any time. However, this action is only really useful when a player is hurt. A behavior author can add domain knowledge by creating a new action *healWhenInjured* that is only available when the player is below 50% health. In a modified version of the GtO game, this is done by adding two new “emergency” actions:

- EBuildFarm (+4 Food, -100 Wood)
    - If the amount of food is low and there is only enough wood to build a farm, then build a farm to create more food.
- ECreatePeasantWood (+200 Wood, -1 Food)
    - If the amount of wood is low, then create a wood peasant to increase the amount of wood.

With the addition of these two actions, a player is less likely to get into a position where it gets stuck – if the food is getting low then a farm is created and if wood is getting low (which is required to build a farm) then wood is created. In the EDS learner, these two actions are given the Highest priority.

The addition of these two actions dramatically changes the results as shown in Figure 54. Now, EDS without state abstraction performs very well, with higher scores than either table-based Q-learners with biased initial values (Q Biased) or the single-state Q-learner (Q Abstract). This result demonstrates that EDS works better with scriptable actions, where actions are encoded in the form of IF-THEN rules, than when all actions are generally applicable. The IF portion of an action is way to intuitively add context information to actions. Generally, higher priority actions should have the most restrictive IF clauses while lower priority actions may have no IF clause at all.

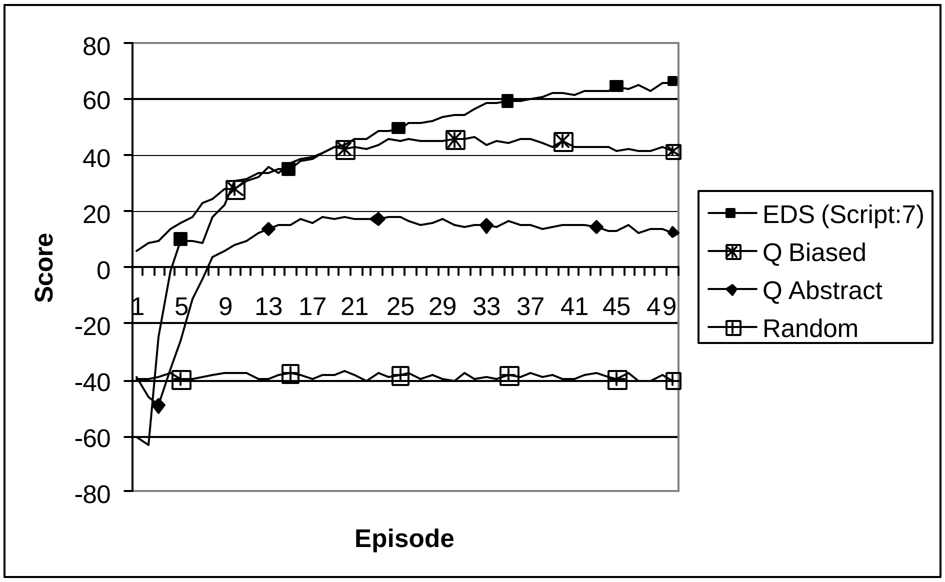


Figure 54 Overall results on the Get the Ogre game when additional actions are included.

Resource Gathering

The Resource Gathering game is an abstract version of another problem found in real-time strategy games, where the goal is to harvest in-game resources, such as Gold or Wood, and return them to the Town Hall. This game was previously examined by Mehta et al., (2008), where they focused on the automatic construction of task hierarchies. In this game, the goal is to move to a location with a line of sight to a gold mine or forest, mine gold or chop wood, and then return the resources to the town hall. The goal is to acquire 100 wood and 100 gold as quickly as possible – one trip to the forest and one trip to the gold mine.

Game Definition

The Resource Gathering game is defined by the sets *F, A,* and *R.* The game contains five visible observation features and five hidden state variables. There are thirteen actions available to the player, some of which depend upon the hidden state variables. State transitions are not made randomly in this game, but instead depend explicitly on the action selected.

- *F* =
    - Gold: Integer (Initial value = 0)
    - Wood: Integer (Initial value = 0)
    - Location: Integer (Initial value = 5)
    - HoldingGold: Boolean (Initial value = false)
    - HoldingWood: Boolean (Initial value = false)
    - GoldMine1: Integer (Initial value = 1) [Hidden]
    - GoldMine2: Integer (Initial value = 2) [Hidden]
    - Forest1: Integer (Initial value = 2) [Hidden]
    - Forest2: Integer (Initial value = 3) [Hidden]
    - TownHall: Integer (Initial value = 5) [Hidden]
- *A* =
    - MoveTo1
    - MoveTo2
    - MoveTo3
    - MoveTo4
    - MoveTo5
    - MoveTo6
    - MoveTo7
    - MoveTo8
    - MoveTo9
    - MineGold (Location = GoldMine1 || GoldMine2)
    - ChopWood (Location = Forest1 || Forest2)
    - DropOffWood (Location = TownHall; HoldingWood = true)
    - DropOffGold (Location = TownHall; HoldingGold = true)

- *R* =
  - MoveToX (Location = X)
  - MineGold (HoldingGold = true; HoldingWood = false)
  - ChopWood (HoldingWood = true; HoldingGold = false)
  - DropOffWood (HoldingWood = false; +100 Wood)
  - DropOffGold (HoldingGold = false; +100 Gold)

An episode is over when 100 Wood and 100 Gold have been gathered. The rewards in the original version of this problem were -1 for every completed action, given at the end of the episode. The learner is rewarded for solving problem as quickly as possible. So, one optimal solution is with a reward of -8 is:

1. MoveTo1
2. MineGold
3. MoveTo5
4. DropOffGold
5. MoveTo3
6. ChopWood
7. MoveTo5
8. DropOffWood

Two changes were made to the reward function for the abstract game. First, an immediate reward is supplied instead of an episodic one. This reward function is: –1 * number of actions completed. Additionally, the maximum number of action attempts allowed in an episode is 100.

Player Definition

Similar to the previous games, three different players are used to solve the game: EDS, Q, and Random. Based on the similarities of this game to the GtO game, we focus mainly on describing the EDS player.

First, for the same reason that EDS failed to learn an effective script in GtO, it will also fail here if episodic rewards are used. Using the following priority assignment, a minimal script would need to contain MoveTo2, MoveTo5, MineGold, ChopWood, DropOffGold, and DropOffWood.

- High {DropOffGold, DropOffWood}
- Medium {ChopWood, MineGold}
- Low {MoveTo1, MoveTo2, …, MoveTo9}

With episodic learning, EDS will be unable to effectively choose between actions in the script of the same priority such as MoveTo2/MoveTo5 and ChopWood/MineGold. This is because for the duration of the entire episode, the highest-valued action for a given priority would always be selected. Even if the action script contained only these actions, the agent would be able to move to location 2, never location 5 (or vice versa). This is a limitation of the dynamic scripting approach.

Results and Discussion

This section contains the results generated by running the three playrs in the context of the Resource Gathering game. The x-axis contains the episode number, from 1 to 100. The y-axis is the number of actions required by the learner in order to complete the problem. Fewer actions indicate a more efficient solution, so a lower number is better. In all cases, the results are the average of 100 learners.

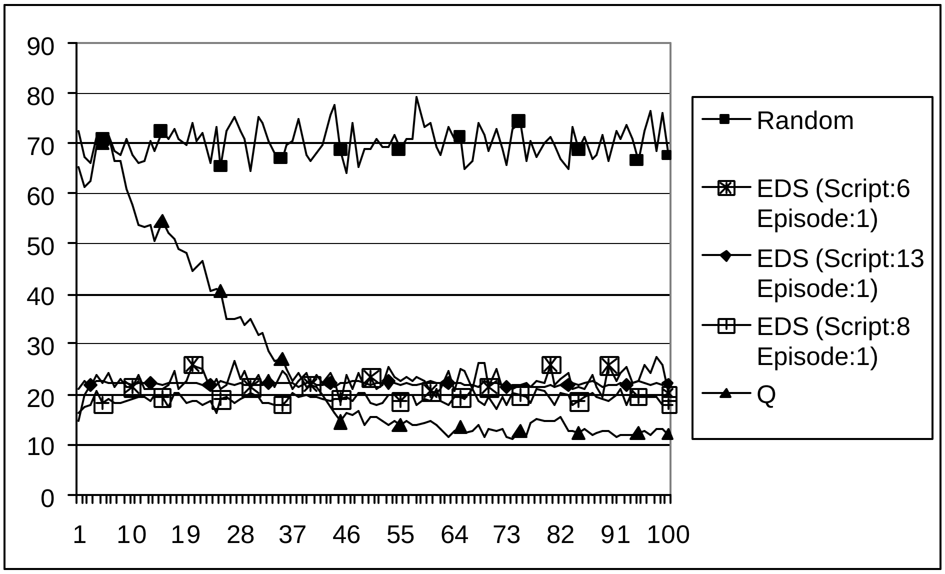


Figure 55 Performance on the Resource Gathering problem. The goal is to minimize the number of actions required to complete the problem.

The results in Figure 55 capture two main results. First, the standard table-based Q player does quite well on this problem, starting out with performance equal to that of the random player but ending up in the range of 12 actions – very close the optimal solution of 8 actions.

Second, the EDS player does much better than the random player, despite the fact that no learning occurs throughout the 100 episodes. EDS performs better than Random because of the combination of immediate rewards and priorities. The negative immediate rewards make it less likely that EDS will repeat the same action twice in a row. This results in a near random search through the action space, except that if the higher priority actions are available, they will be chosen before a lower priority action. While the negative rewards encourage the random selection of actions, the embedded priority information ensures better performance than found with the random player. Additional evidence for this hypothesis is the fact that script size has relatively little effect on player performance.

Modified Version of Resource Gathering

As seen previously, additional domain information can be authored in a number of forms: preconditions for the actions (IF statements), automatic and manual state abstractions, and immediate rewards to alter the script ordering in real time. In this section we introduce a fourth method of including domain knowledge through the creation of task hierarchies. A manually created task hierarchy seems a reasonable approach to this problem as it mirrors automatically created task hierarchies that have been found to perform well on this problem (Mehta et al., 2008). Figure 56 to Figure 61 shows the manually constructed task hierarchy used by the EDS player. First, the top level Main behavior is shown in Figure 56. It contains a single choice point that is expected to learn how to balance resource gathering tasks in order to achieve a specified quota.

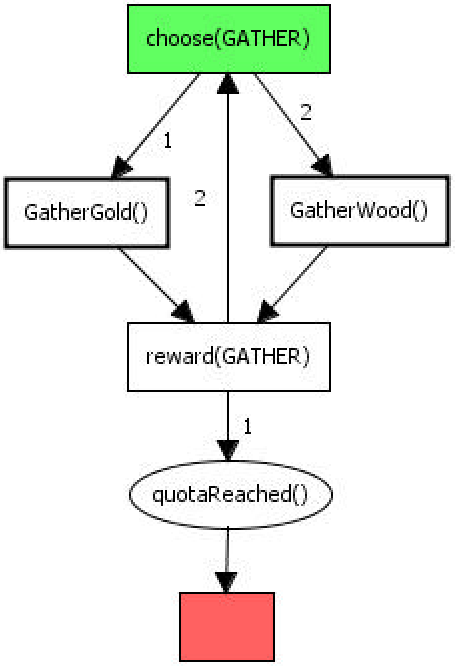


Figure 56 Top-level Main behavior.

Figure 57 and Figure 58 show the intermediate level behaviors for GatherWood and GatherGold. Both of these behaviors make use of a combination of sub-behaviors (FindForest) and primitive actions (ChopWood). No learning occurs in these behaviors, as the transitions have been completed specified as part of the behavior authoring process.

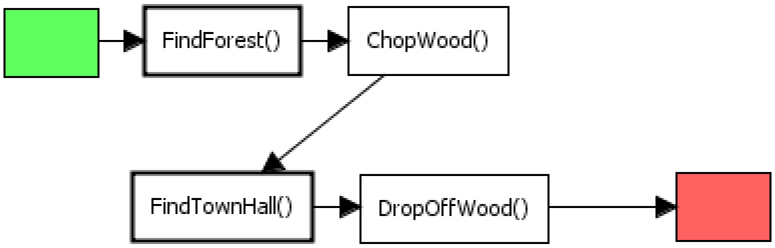


Figure 57 GatherWood sub-behavior called by Main behavior.

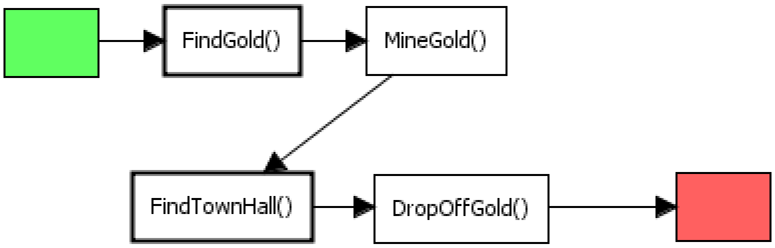


Figure 58 GatherGold sub-behavior called by Main behavior.

The next three figures all show behaviors with a single choice point. In each case, the behavior is attempting to learn a script which will allow it to find a hidden resource location: a forest, a gold mine, or the town hall.

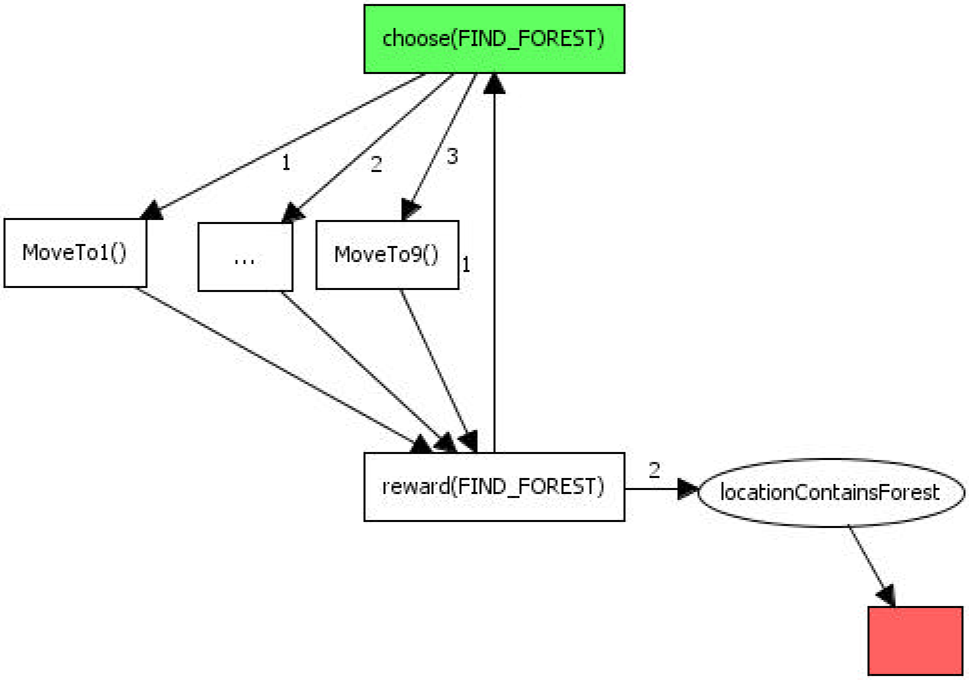


Figure 59 FindForest sub-behavior called by GatherWood.

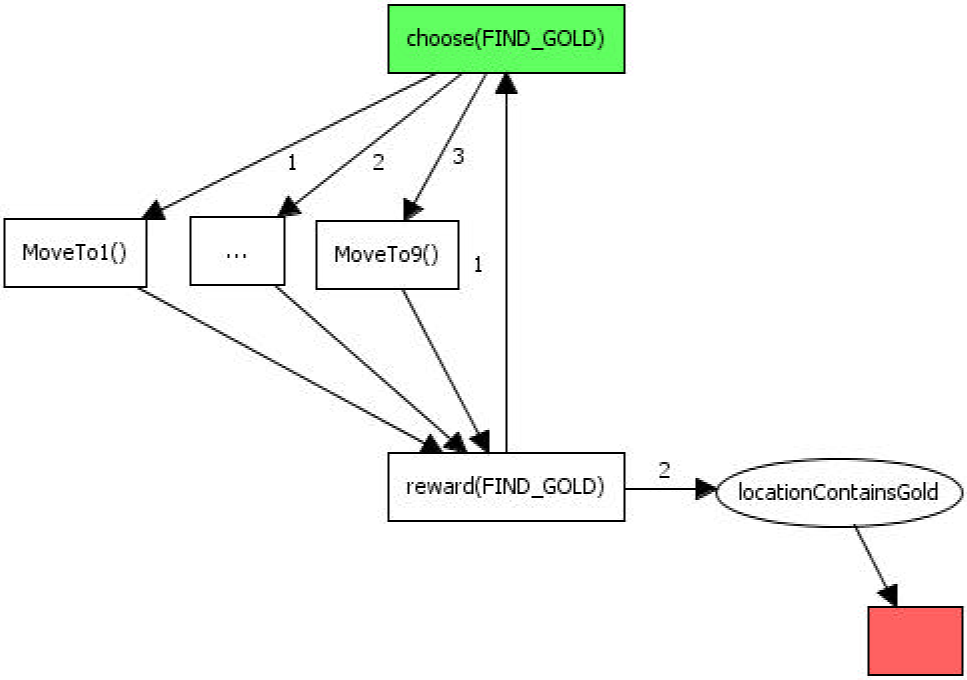


Figure 60 FindGold sub-behavior called by GatherGold.

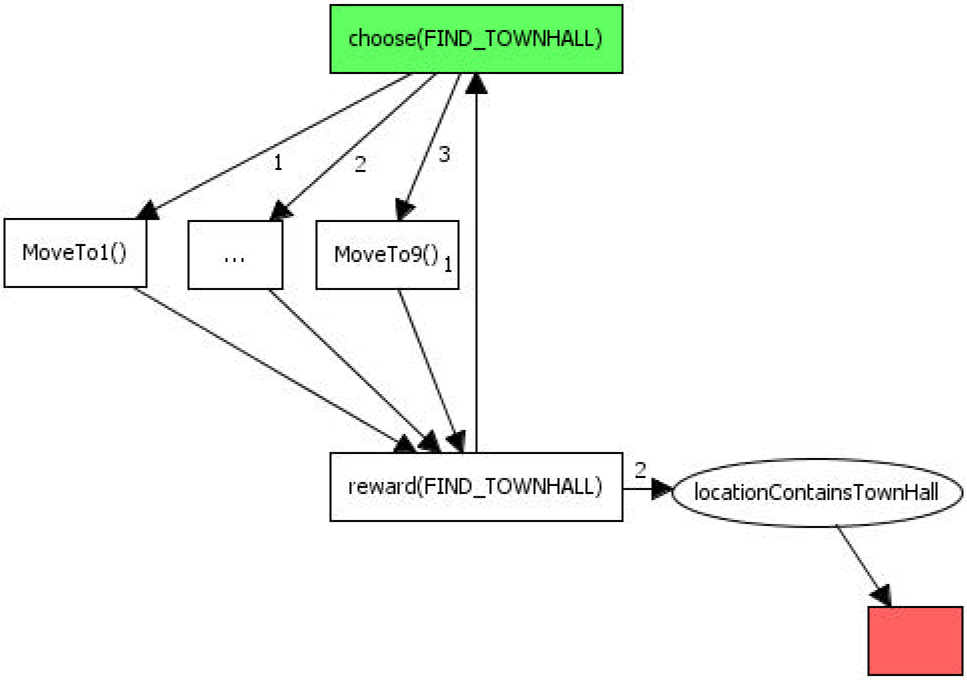


Figure 61 FindTownHall sub-behavior called by GatherGold and GatherWood.

The end result is that there are four sub-problems that the learner must solve: (1) balancing the gathering of wood vs. gold, (2) finding a hidden forest location, (3) finding a hidden gold location, and (4) finding the hidden town hall location. A significant amount of knowledge about how to complete the task has been encoded in the task hierarchy, such as that when gathering wood the character should ChopWood immediately after finding a forest.

Each action in a choice point has an initial value of 100, a maximum value of 150 and a minimum value of 50. The top level GATHER choice point always receives an immediate reward of -40. For the three FIND_ choice points, a reward of -40 is immediately provided for each incorrect selection and a reward of +40 for each correct selection.

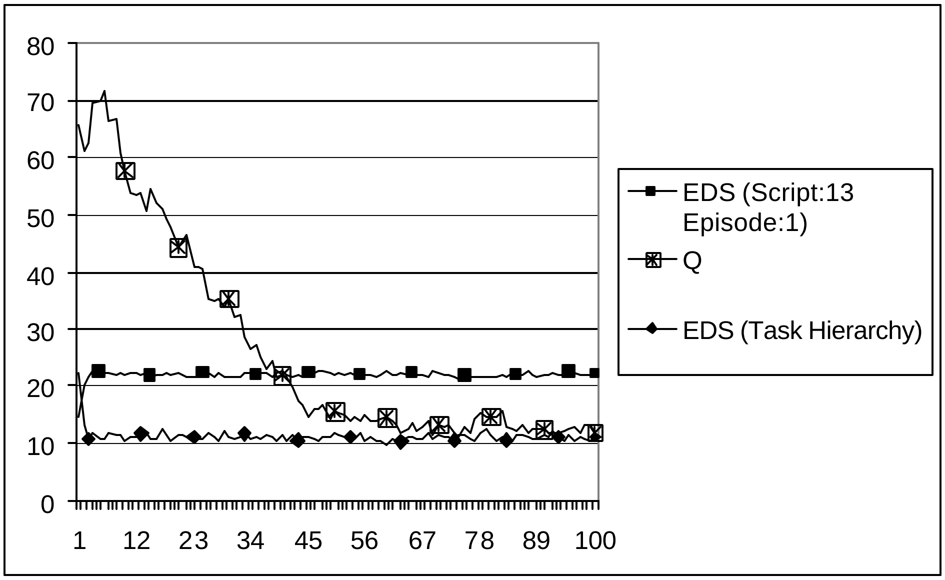


Figure 62 Number of actions per episode. EDS (Task Hierarchy) indicates the EDS algorithm with the manually constructed task hierarchy.

The manual task hierarchy greatly improves the performance of the EDS algorithm as shown in Figure 62, quickly learning to perform as well as the table-based Q player that does not make use of the task hierarchy[2]. This result demonstrates that EDS can learn to solve a problem that could not be addressed very well with standard dynamic scripting, while at the same time arriving at a maximum performance level similar to Q player.

[2] Of course, if the Q player made use of the task hierarchy it would be expected to more quickly reach maximum performance as well. The comparison made here is that they both arrived at nearly the same level of maximum performance given a fixed temperature in the soft-max selection algorithm, not the speed at which it is reached.

CHAPTER V
DEMONSTRATION

The game Neverwinter Nights (NWN) is used to demonstrate EDS in a commercial computer game, allowing comparisons in performance between the original and extended dynamic scripting algorithms. This chapter begins by describing the Neverwiner Nights computer game. Included in this initial section are details related to the integration of the NWN game and the extended dynamic scripting implementation. The following section discusses a wrapper application that is used to perform multiple experiments by varying the learning parameters. The methods section describes the common procedures used in all of these experiments and a list of the results to be gathered is found in the following section on learning performance measures. The final section of this chapter details the learning parameters used for each experiment and presents the generated results.

NeverWinter Nights

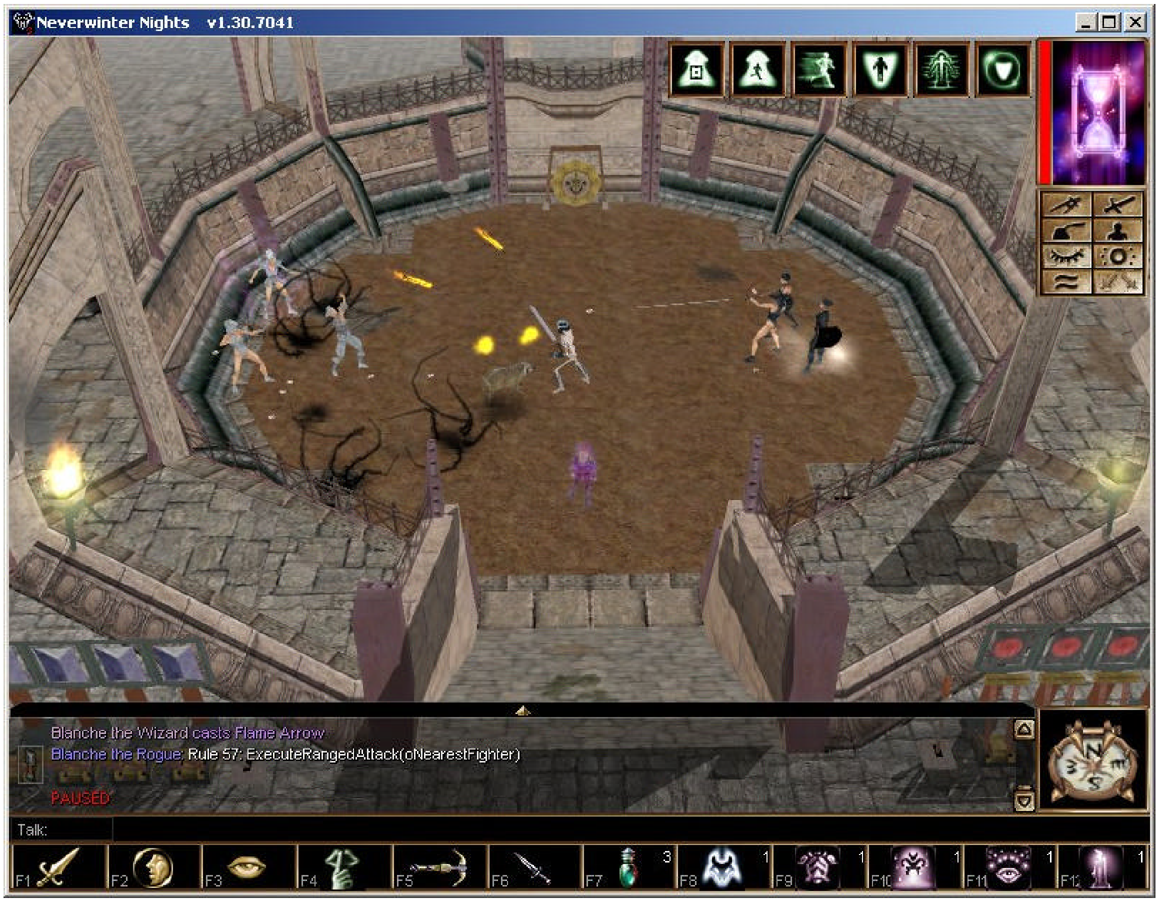
Neverwinter Nights (Bioware, 2002) is a role-playing game where a player controls one or more *characters* in a fantasy medieval setting. The characters are generally of two basic types: characters that use physical attacks such as swords and bows (fighters) and those that cast helpful spells at their friends and harmful spells at their enemies (magic users). These c haracters encounter and sometimes battle other *parties*, where a party consists of some number of characters and/or monsters. Figure 63 shows a NWN screenshot with a human party (controlled by a person) and a spider party (controlled by the computer). This particular game was selected to facilitate direct comparisons between the extended and standard dynamic scripting algorithms, as Neverwinter Nights was also a testbed for the dynamic scripting algorithm.

Figure 63 NWN screen shot of a human party and a spider party (www.bioware.com).

The NWN game performs well as a testbed for research into game AI as it has both a powerful toolset and a strong user base. The toolset allows programmers to create their own NWN worlds and contains a scripting language that can be used to change the behavior of objects and characters. The syntax of this scripting language is similar to the C computer programming language. The advantage of a strong user base is that there are online question and answer forums on how to use the scripting language – a valuable resource when programming in a non-mainstream language such as NWN Script.

Communicating with NWN

In order to test the dynamic scripting algorithm in NWN, Spronck et al. (2006) directly implemented their algorithm in the NWN Script language. This implementation was included in a NWN Arena module that provided an infrastructure for parties to battle each another in an arena-like setting. Each battle is called an *encounter* that ends when one team defeats all members of the opposing team. The module and scripting code are available for download at www.cs.unimaas.nl/p.spronck/NWN.htm. Figure 64 is a screenshot of two identical teams (one in white outfits, the other in black outfits) battling in the arena. The learning team (white uniform) is controlled by the DS algorithm while the static team (black uniform) is controlled by the standard NWN character scripts. The badger and skeleton, in the center of the arena, are creatures summoned by the white and black teams.



Blanche the Wizard casts Flame Arrow
Blanche the Rogue: Rule 57: ExecuteRangedAttack(oNearestFighter)
PAUSED

Figure 64 The NWN module for testing dynamic scripting (Spronck).

The goal for this dissertation is to demonstrate the performance of a general implementation of the extended dynamic scripting algorithm, so re-implementing EDS in NWN Script is not an option. Instead, the original arena scripts from Spronck et al. (2006) were modified to write out the character state to a MySQL database at particular points in the script and to read the action to be performed by a character from the MySQL database. A database conduit was used, rather than a normal socket connection, because NWN Script does not support any external connections. The MySQL functionality was provided by the free third-party tool Neverwinter Nights

Extender (www.nwnx.org), which is an NWN server for running persistent multi-player worlds.

State information is written whenever particular events happen to a character:

- When it is created
- On a its heartbeat (every six seconds)
- At the end of a combat round
- Damage is taken by it
- On its death
- An action is read from the database for it

Character state is written as a single XML string that is written to the database (see 0 for an example). The character state information contains the actions that are currently available to the character as well as some basic perceptions about the opposing team. For example `<BUFF_SELF>1</BUFF_SELF>` indicates that the BUFF_SELF action is currently available to this character and `<NEAREST_FIGHTER />` indicates that there is an opposing fighter nearby. Numerous modifications were made to the NWN Arena scripts to support writing out the state information at all of these points.

A character reads in an action from the database when it currently has no action to perform and an enemy is perceived. This functionality required a smaller change to the NWN Arena script, replacing the large code base that implements dynamic scripting with a function that reads a string representation of the action to be performed from the MySQL database. For example the following strings indicate that the highest possible summon spell should be cast at the nearest enemy:

```
“SPELL_HIGH_SUMMON”   “NEAREST”
```

While the script is processing an action, a notification string of PENDING is written to the database. After an action has been completed, this is changed to COMPLETED if the action was completed successfully or FAILED if the action failed for some reason. Note that if an action fails, the character will instead perform an action selected by the standard NWN character scripts (the same scripts controlling the non-learning team).

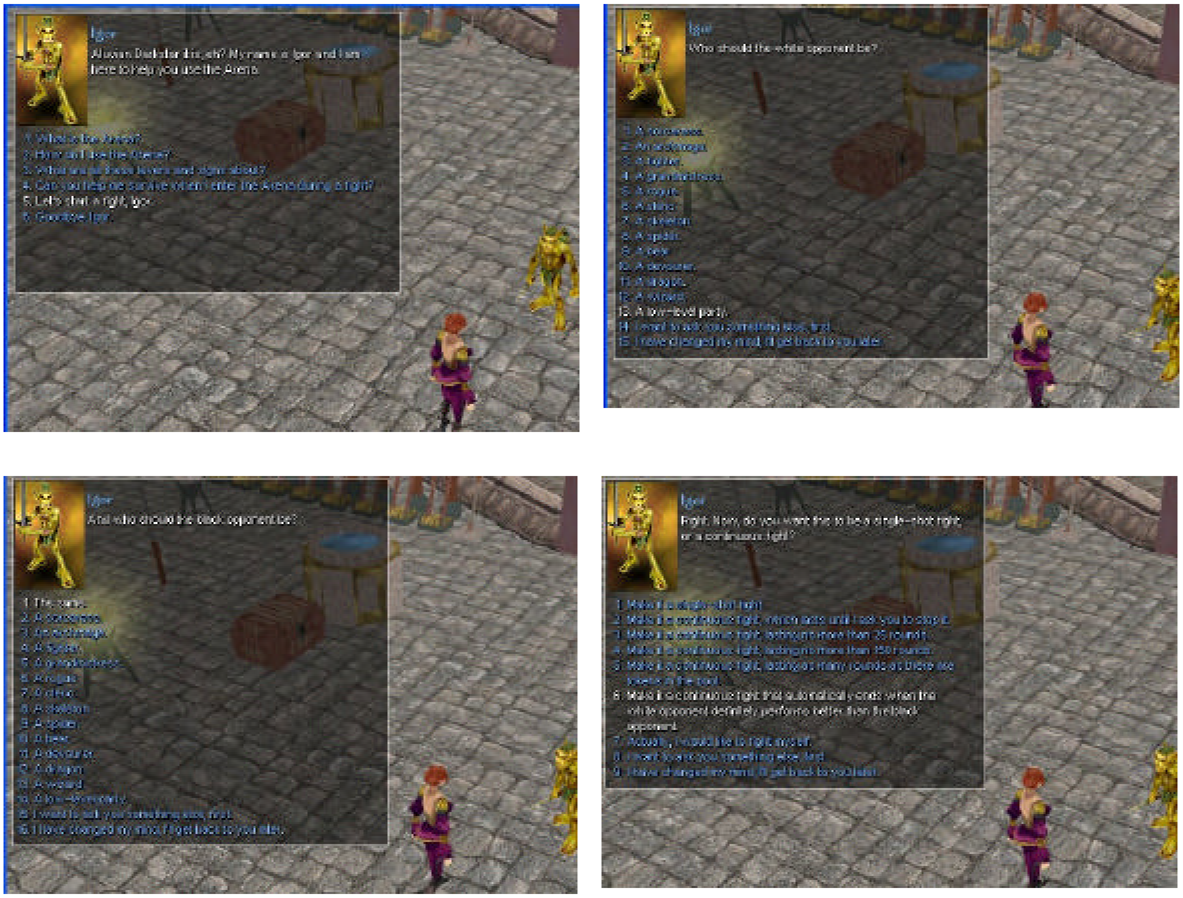
A separate Java application is used to connect the extended dynamic scripting implementation with NWN, using the database as a conduit. The wrapper is driven by a main loop which reads the current state information from the database, requests a decision from the extended dynamic scripting library, and then writes out the current action to be performed once every second. The wrapper also checks the database for information on the status of the current encounter, for notification of events such as an encounter ending.

Experiment Setup

A number of factors remain constant across the different experiments: the use of the NWN Arena module, the reward function, and the some of the learning algorithms settings. Each is covered below.

The NWN Arena module created by Spronck et al. (2006), shown in Figure 64, is used for all of the experiments described below. The learning team (white uniforms) is controlled by the extended dynamic scripting algorithm through the wrapper application. The static team (black uniforms) is controlled by the standard NWN character scripts that ship with the NWN game. The Arena dialog infrastructure is used to setup a series of encounters between two identical mid-level parties, where both parties consist of a Fighter, Rogue, Mage, and Priest. The dialog sequence is shown in Figure 65.

The result of this sequence is that a learning team and a static team are created. The two parties then battle in the Arena until all of the members of one team are killed. At this point, the encounter is over and all remaining characters and summoned creatures are removed from the Arena. Two new parties are then created to start a new encounter. This continues until a *turning point* is reached, a measurement encoded in the NWN arena scripts, which indicates that the learning team is consistently outperforming the static team. This measure is defined in detail in the next section.



Igor
[illegible] Gladiator [illegible], eh? My name is Igor and I am here to help you use the Arena.

1. What is the Arena?
2. How can I use the Arena?
3. What are all these levers and signs about?
4. Can you help me survive when I enter the Arena during a fight?
5. Let's start a fight, Igor.
6. Goodbye Igor.

Igor
Who should the white opponent be?

1. A sorceress.
2. An archmage.
3. A fighter.
4. A grandmistress.
5. A rogue.
6. A cleric.
7. A skeleton.
8. A spider.
9. A bear.
10. A devourer.
11. A dragon.
12. A wizard.
13. A low-level party.
14. I want to ask you something else, first.
15. I have changed my mind, I'll get back to you later.

Igor
And who should the black opponent be?

1. The same.
2. A sorceress.
3. An archmage.
4. A fighter.
5. A grandmistress.
6. A rogue.
7. A cleric.
8. A skeleton.
9. A spider.
10. A bear.
11. A devourer.
12. A dragon.
13. A wizard.
14. A low-level party.
15. I want to ask you something else, first.
16. I have changed my mind, I'll get back to you later.

Igor
Right. Now, do you want this to be a single-shot fight, or a continuous fight?

1. Make it a single-shot fight.
2. Make it a continuous fight, which lasts until I ask you to stop it.
3. Make it a continuous fight, lasting no more than 25 rounds.
4. Make it a continuous fight, lasting no more than 250 rounds.
5. Make it a continuous fight, lasting as many rounds as there are [illegible] in the pool.
6. Make it a continuous fight that automatically ends when the white opponent definitely performs better than the black opponent.
7. Actually, I would like to fight myself.
8. I want to ask you something else, first.
9. I have changed my mind, I'll get back to you later.

Figure 65 Dialogs used in the NWN Arena module to initiate a combat session that runs until the learning team reaches the turning point.

The episodic reward function created by Spronck (2006) and defined in the original NWN Arena scripts, remains constant across the different experiments. The reward function analyzes the fitness value of the character and returns a value between -50 and 100. The reward function is defined as:

1. (Initialize values.)
    a. Set *breakEven* ← 0.3
    b. Set *mxPenalty* ← 50
    c. Set *mxRward* ← 100
2. (Get the fitness for this character.) Set *fitness* ← getFitness()

3. (Determine the reward.) If *fitness* < *breakeven*
    a. Then *reward* = $-\dfrac{breakEven - fitness}{breakEven} * mxPenalty$
    b. Else *reward* = $\dfrac{fitness - breakEven}{1.0 - breakEven} * mxRward$

Fitness is defined as a character, and its teammates, being alive and healthy at the end of the encounter. The team component of the fitness function takes into account the number of group members remaining and the ratio of their current health to their maximum health.

1. (Initial value.) Set *groupFitness* ← 0.2
2. (Factor in remaining team members.) Set *groupFitness* ← *groupFitness* + $\dfrac{remainingTeamMembers}{4} * 0.4$
3. (Factor in remaining team health.) Set *groupFitness* ← *groupFitness* + $\dfrac{remainingTeamHealth}{mxTeamHealth} * 0.4$

The individual portion of the function combines the group fitness measure with the remaining health of the individual character if they are still alive. If this character is not alive, then its fitness is the same as the group fitness.

1. (Initial value.) Set *individualFitness* ← 0.5 * *groupFitness* + 0.3
2. (Factor in remaining health.) Set *individualFitness* ← $\dfrac{remainingHealth}{mxHealth} * 0.2$

Finally, the learning algorithm requires that values be set for a number of free parameters. These remain constant throughout all of the experiments. The temperature used in the soft-max selection algorithm is set to 5. The minimum action value is set at 0 and the maximum action value is set at 2000. For all actions, the initial action value is set at 100.

Learning Performance Measures

There are a number of measurements that can be compared across experiments to demonstrate the effects of different learning parameters. The measurements this paper is concerned with are learning efficiency and action variety.

Learning Efficiency

An important measurement for online learning algorithms is learning efficiency. The computer is expected to learn how to beat the human opponent given a relatively small number of encounters to learn from. Learning efficiency is measured by determining the average number of episodes required to beat a particular adversary, known as the *turning point*. Spronck et al. (2006) defined the turning point in NWN as:

> “The dynamic team is said to ‘outperform’ the static [controlled by standard NWN scripts] team at an encounter, if the average fitness over the last ten encounters is higher for the dynamic team [controlled by EDS] than for the static team. The turning point is the number of the first encounter after which the dynamic team outperforms the static team for at least ten consecutive encounters.”

As implemented in the original NWN Arena scripts, the minimum number of encounters required to reach a turning point is 19. We make use of the same definition in our duplication of their experiment.

A secondary measurement of learning efficiency introduced is provided by examining the average fitness of the two teams in size 10 windows. This provides information on the speed at which a learning algorithm is progressing to the turning point.

Variety

The second measure of experiment performance is action variety, based on the idea that characters in a game are more entertaining (for the human player) if they exhibit a variety of behaviors rather than performing the same actions over and over again. Spronck et al. (2006) write that requirement of variety is met by the script aspect of the dynamic scripting algorithm: “Dynamic scripting generates a new script for every agent, and thus provides variety in behaviour.” To allow us to analyze action variety quantitatively in this demonstration, we developed three diversity measures: *possible diversity, selection diversity,* and *repeat diversity*.

The first measure, *possible diversity*, captures the number of actions that a character has a reasonable possibility of including in a dynamically generated script. This measure examines the action values when the turning point is reached and counts the number of actions that have a reasonable chance of selection, where reasonable is defined as having an action value > 50. The idea is to count all actions with non-zero action values, where the threshold is set at half of the starting value. In the case where the actions are hierarchical, a combined action list is created that contains the maximum value for an action across the different characters. This means that if the same action is available to two characters, and the value reaches the threshold for both characters, it still only counts as a single action with respect to variety.

The second measure, *selection diversity*, examines the number of different actions that were actually used by each character during an encounter. During an episode, each character select some number of actions. At the end of the encounter, this measure counts the number of distinct actions that were selected across the four different characters, *s*, and then divides by the number of actions selected by all the characters, *n*. For example, the fighter selects action a twice and the mage selects a once, b once, and c twice. The action selections are (a, 3), (b, 1) and (c, 2), where (x, y) indicates action x was selected y times. The number of distinct selected actions is 3 (|a, b, c|) and the number of selected actions, *n*, is 6 (3 + 1 + 2). We now have:

```
Selection diversity = s / n = 3 / 6 = .5
```

This measure captures the ratio of distinct to repeated actions in a single episode, with a maximum of 1 indicating that no actions were repeated and a minimum of 1/n indicating the same action was selected every time. Longer sequences of actions create a lower minimum bound. If s = 1 and n = 2, the measure is .5 while if s = 1 and n = 10 the measure is .1. That is, performing the same action every time during a short encounter is better than performing the same action every time during a long encounter.

One difficulty with selection diversity is that it does not distinguish between the following two action selection sets: {(a,9), (b, 1)}, {(a,5), (b,5)}. In both cases, the selection diversity is 2/10, while from a player's standpoint the second set of action values has more variety.

The third measure, *repeat diversity*, describes the distribution of the repeated actions in order to capture variety of the repeated actions and address the shortcoming in the selection diversity measure. Repeat diversity is defined as the mean squared error (MSE) of the action distribution relative to a mean action distribution. MSE provides a quantitative measurement of the difference between the actual action distribution and the ideal (mean) action distribution. MSE is defined as: $\frac{1}{n}\sum_{j=1}^{n}\left(q_j - q\right)^2$, where θ_j is the number of times an action was used and θ is the number of selected actions / number of distinct actions. Looking at the problematic action selection sets, the repeat diversity of (a,9), (b, 1) is 16 while the repeat diversity of (a,5), (b,5) is 0. The second action set has greater repeat diversity relative to the first action set.

Note that if a small number of actions are much more effective than the rest of the actions, then variety is at odds with learning efficiency. This is a tradeoff that the behavior designer has some control over by adjusting the parameters of the EDS algorithm. However, for the experiments detailed below, all parameters are fixed. This means that while we measure diversity for comparison purposes, we to not actively seek to maximize diversity.

Extended Dynamic Scripting Learning Parameters

In addition to supporting the sense/act loop, the wrapper application is used to set values for a number of extended dynamic scripting learning parameters: script, selection, hierarchy, and reward modes. By varying the learning parameters, different experiments can be carried out that demonstrate the effects of the extensions to the dynamic scripting algorithm. The algorithms behind each of the learning parameters are discussed in the following sections.

Script Mode

There are two script modes available, SCRIPT and ALL. The script mode affects the script generation before an episode and how they are updated after an episode.

SCRIPT mode matches the standard dynamic scripting algorithm, where an action script is created before an episode. This uses a free-parameter, which determines the size of the script. The script sizes remain unchanged from the original dynamic scripting work, where the size of the script is 10 for magic users and 5 for fighters. Scripts are created by selecting actions, without replacement, from the set of all actions available to the character. Actions are chosen via the soft-max selection algorithm described previously. The SCRIPT mode also affects how rewards are applied to actions. In this mode, the full reward is given to each action in the script that was successfully completed during the encounter. A half reward is given to each action in the script that was either not selected or failed to complete successfully. Compensation is applied to all actions that are not part of the script.

In the ALL mode, action selection is much simpler – all actions available to a character class become part of its action script. In the case of value updating, the algorithm is the same for both ALL and SCRIPT mode. The main difference in this mode is the contents of the three lists. In the ALL mode, the completedList contains all of the successfully completed actions and the notInScriptList contains all of the actions that either were never completed successfully or never selected.

Selection Mode

Action selection during the episode is controlled by one of two selection modes: PRIORITY and VALUE. In both modes, only actions applicable to the current context are considered. This means both that the character can perform the action and that a valid target is available. For example, in order to carry out a ranged attack a character must have a ranged weapon (such as bow and arrows) and a target (such as nearest fighter). It is possible that the character has no applicable actions to perform. In this case, the DEFAULT action is written to the database and the character falls back on the standard NWN character scripts.

The difference between the two selection modes is in how they select an action when an agent is required to make a decision. The PRIORITY selection mode makes use of the author-assigned priority of an action: HIGHEST, HIGH, MEDIUM, LOW, or LOWEST. In PRIORITY mode, actions are sorted first on whether or not an action has already failed in this encounter, second on action priority, and third by action value. While this selection mode is mostly consistent with the definition of the dynamic scripting algorithm, sorting on action failure was not part of the original dynamic scripting implementation. This addition was required by the NWN communication implementation. Having an action fail generally means it is no longer available to the character (e.g. the last healing potion was used), so sorting on action failure keeps the agent from choosing the same high priority, but failing, action multiple times in the same encounter.

For the VALUE selection mode, the values assigned to the actions are used to select the action rather than its priority. VALUE selection makes use of the same soft-max algorithm used to select the action script in dynamic scripting. This fitness proportionate selection algorithm is a typical selection method for reinforcement learning implementations.

Hierarchy Mode

The hierarchy mode has two different settings, FLAT and CLASS, In the FLAT mode, all character classes (Fighter, Rogue, Mage, Priest) share the same global set of action values, in keeping with the original implementation (Spronck et al., 2006). That is, if a Fighter rewards the action HEAL_SELF and increases the action value, the action value will also increase for a Mage. Each character does have its own dynamically generated script.

In the CLASS mode, each character makes use of separate set of global action values in addition to its own script. This corresponds to separate choice point for each character class. In this mode, if the Fighter rewards the HEAL_SELF action, the action value will be unchanged for the Mage (and vice versa). The CLASS mode makes use of choice and reward points to realize manual state abstraction.

Reward Mode

The reward mode allows the wrapper to switch between the standard EPISODIC dynamic scripting reward function and the new SCALED version of the reward function. In both cases, actions are rewarded at the end of every encounter using the algorithms outlined in Chapter 0. The reward function itself is described in the Experiment section of the current chapter.

Automatic State Abstraction

There are two possible selections for this parameter: NONE, and STUMP. When this parameter is set to NONE, only a single abstract state is used by the learning algorithm. In the other case, the learning algorithm utilizes decision tree methods to automatically add new abstract states. In STUMP mode the Decision Stump algorithm is used. The J48 decision tree state abstraction algorithm was not tested in NeverWinter Nights based on its performance in the abstract version of NWN. Both of these algorithms are described in Chapter 0.

Methods

This section describes the different experiments carried out in the NeverWinter Nights domain. For each experiment, the motivation behind the experiment and the supporting learning parameters are discussed.

Experiment 1: Dynamic Scripting

The first experiment is designed to generate results using the standard dynamic scripting algorithm. These results form a baseline to which the various dynamic scripting enhancements can be compared. The learning parameter settings for this experiment are:

- ScriptMode.SCRIPT
- SelectionMode.PRIORITY
- HierarchyMode.FLAT
- RewardMode.EPISODIC
- AbstractionMode.NONE

As with the original dynamic scripting algorithm, action scripts are dynamically created for each episode, applicable actions are chosen in priority order from the script, action values are shared across all characters, and rewards are applied at the end of an encounter.

Experiment 2: Q Variant

This experiment examines the difference affect of replacing the dynamic scripting algorithm with something akin to standard Q-learning. The learning parameter settings for this experiment are:

- ScriptMode.ALL
- SelectionMode.VALUE
- HierarchyMode.FLAT
- RewardMode.EPISODIC
- AbstractionMode.NONE

There are two main differences from the baseline experiment. First, all actions are available to the character during an encounter rather than only those in the script. Second, actions are selected based on their value and not their priority. No hierarchy is used in this experiment and the reward function remains the same as in the original experiment. The resulting algorithm is a type of Q variant that makes use of the dynamic scripting value update function rather than a more standard update function such as Q-learning. We expect that this algorithm will have lower scores on both learning efficiency and action variety than dynamic scripting.

Experiment 3: Extended Dynamic Scripting

The third experiment examines the affect of hierarchy on the NWN character behaviors. The learning parameter settings for this experiment are:

- ScriptMode.SCRIPT
- SelectionMode.PRIORITY
- HierarchyMode.CLASS
- RewardMode.EPISODIC
- AbstractionMode.NONE

The only difference from Experiment 1 is the change in the hierarchy mode. We expect that this algorithm will have better scores on both learning efficiency and action variety than dynamic scripting.

Experiment 4: Hierarchical Q Variant

This experiment extends Experiment 2 by replacing the single abstract state with the hierarchy used in Experiment 3. Experiment 4 investigates the importance of script selection and action priority when using the hierarchal version of the behavior. The learning parameter settings for this experiment are:

- ScriptMode.ALL
- SelectionMode.VALUE
- HierarchyMode.CLASS
- RewardMode.EPISODIC
- AbstractionMode.NONE

We expect that this algorithm will have lower scores on both learning efficiency and action variety than dynamic scripting but better scores in both measures than the flat Q variant.

Experiment 5: Reward Scaling

This experiment demonstrates the effect of reward scaling by extending the basic dynamic scripting setup (Experiment 1). The learning parameter settings for this experiment are:

- ScriptMode.SCRIPT
- SelectionMode.PRIORITY
- HierarchyMode.FLAT
- RewardMode.SCALED
- AbstractionMode.NONE

We expect that reward scaling will have no effect on either measure in the NWN environment.

Experiment 6: Automatic State Specialization

This experiment demonstrates the affect of automatic state specialization, utilizing the Decision Stump classifier, on the performance of the standard dynamic scripting algorithm.

- ScriptMode.SCRIPT
- SelectionMode.PRIORITY
- HierarchyMode.FLAT
- RewardMode.SCALED
- AbstractionMode.STUMP

We expect that this algorithm will have better scores on both learning efficiency and action variety than dynamic scripting.

Results

The results are grouped to form a number of distinct sections. The first section covers experiments 1-4. This section compares Extended Dynamic Scripting, Q-Learning, and Hierarchical Q-Learning to the basic Dynamic Scripting algorithm. The second section demonstrates the effect of the scaled update algorithm in experiment 5. Finally, the third section provides results for experiment 6 on automatic state specialization. Within each section, the results are divided into two parts: learning efficiency and variety. In all cases, statistical significance is given using the KS Test.

Experiments 1-4

This section describes the results generated from experiments 1-4, where the different learning conditions are: DS, Q, EDS, and HQ. Experiment 1, DS, is designed to generate results using the standard dynamic scripting algorithm with the EDS wrapper. These results form a baseline to which the various dynamic scripting enhancements can be compared. This experiment makes use of a single choice point, where all the characters share a single set of action values. Each character has its own script. The second experiment, Q, examines the effect of replacing the dynamic scripting algorithm with something akin to standard Q-learning. There are two main differences from the baseline experiment. First, all actions are available to the character during an encounter rather than just a script. Second, actions are selected based on their value and not their priority. For comparison purposes, the resulting algorithm is a type of Q variant that makes use of the dynamic scripting weight update function rather than the standard Q-learning update function. As in the previous experiments, all four characters share the same set of action values. Experiment three, EDS makes use of the flexible nature of extended dynamic scripting, replacing the single choice point with four separate choice points, one for each character class. This is a manually constructed form of state abstraction. This experiment is designed to demonstrate the usefulness of choice and reward points that can be placed arbitrarily in a behavior. Experiment 4, HQ, adds the manually constructed hierarchy to the original Q-learning experiment.

Learning Efficiency

The mean turning point for the different experiments, as defined previously, is shown in Table 9. The statistical significance of each difference is given on the right hand side of table, where $p < .05$ indicates a statistically significant difference and an X represents no measurable difference.

Table 9 Mean turning point for experiments 1 to 4

	Mean Turning Point	**n=**	**Q**	**EDS**	**HQ**
DS	24.8	30	p< .05	p< .05	X
Q	39.65	20		p< .05	X
EDS	20.9	30			p< .05
HQ	30.8	30			

The graph in Figure 66 compares the fitness of the four learning algorithms during the first 25 episodes. Each data point on this graph represents the average fitness of the learning team members, for the last ten rounds.

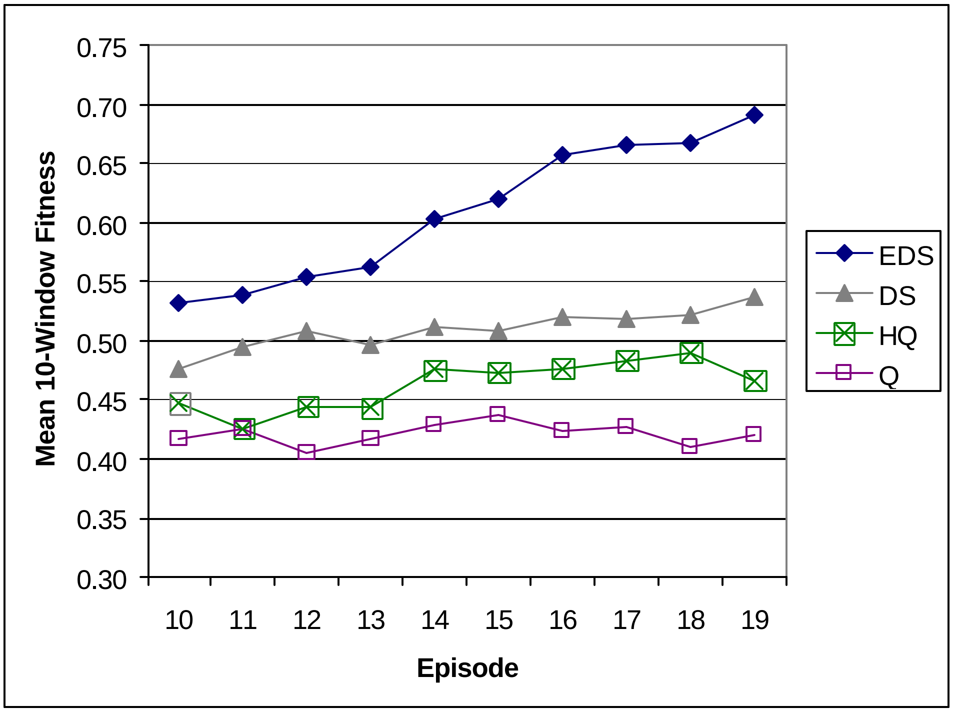


Figure 66 Mean 10-window fitness for experiments 1 to 4. Higher fitness indicates better performance.

Variety

The three different variety measures were gathered for each of these experiments. In all of the tables below, the mean is generated by taking the average value during a run that lasts until the turning point is reached. These values are then averaged across n runs. Table 10 compares the mean possible diversity; Table 11 compares the selected diversity; and Table 12 the mean squared error of the selected actions.

Table 10 Mean possible diversity for experiments 1-4.
Higher values equal greater diversity.

	Mean Possible Diversity (n = 10)
DS	31
Q	32
EDS	49
HQ	50

Both EDS and HQ are significantly different than DS and Q ($p < .05$).

Table 11 Mean selection diversity for experiments 1-4.
Higher values equal greater diversity.

	Mean Selection Diversity (n=10)
DS	0.39
Q	0.33
EDS	0.43
HQ	0.51

The differences between HQ and DS, Q, EDS are significant as well as the difference between EDS and Q ($p < .05$).

Table 12 Mean repeat diversity of selected actions for experiments 1-4.
Lower values equal greater diversity.

	Mean Repeat Diversity (n = 10)
DS	6.44
Q	21.46
EDS	4.37
HQ	3.17

EDS, HQ, DS are significantly different than Q; the difference between HQ and EDS, DS is also significant ($p < .05$).

Experiment 5

This section presents the results for experiment 5, which examines the effect of reward scaling. These results are compared to the basic dynamic scripting results. DS refers to the learning team that makes use of regular dynamic scripting, while Scaled refers to the learning team that makes use of dynamic scripting with the scaled update algorithm. As expected, scaling has no significant effect on the performance of dynamic scripting in NeverWinter Nights.

Learning Efficiency

Table 13 Mean turning point for experiment 5.

	Mean Turning Point	DS	Scaled
DS	24.8 (n = 30)		X
Scaled	27.6 (n = 20)		

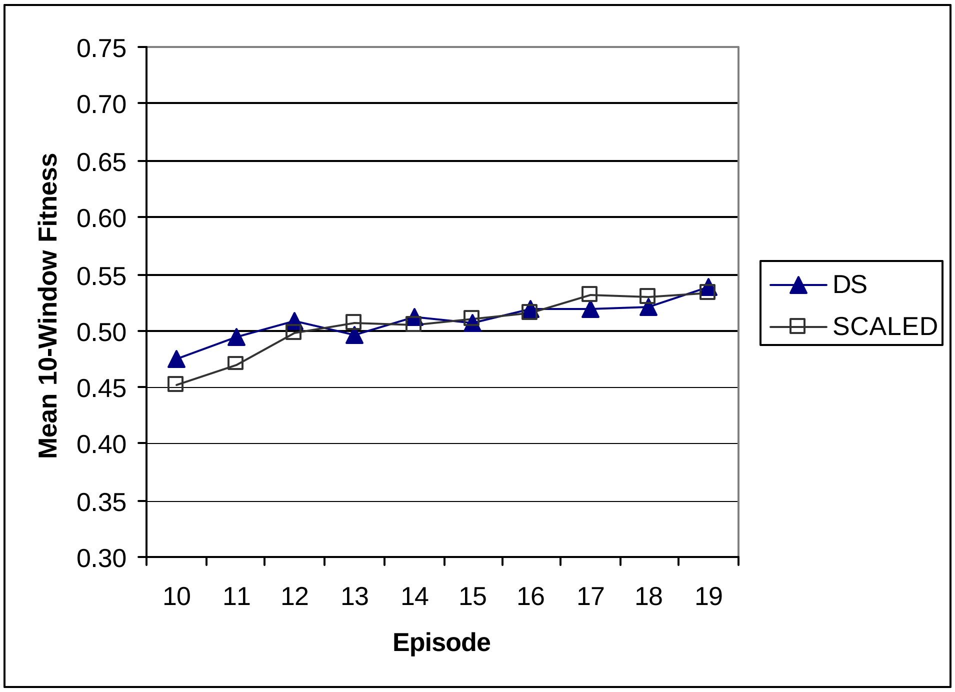


Figure 67 Mean 10-window fitness for experiment 5.

Variety

Table 14 Mean possible diversity for experiment 5.

	Mean Possible Diversity (n = 10)
DS	31
Scaled	28

None of these differences are significant at the p < .05 level.

Table 15 Mean selection diversity for experiment 5.

	Mean Selection Diversity (n=10)
DS	0.39
Scaled	0.35

None of these differences are significant at the p < .05 level.

Table 16 Mean repeat diversity of selected actions for experiment 5.

	Mean Repeat Diversity (n = 10)
DS	6.44
Scaled	7.37

None of these differences are significant at the p < .05 level.

Experiment 6

This section presents the results for experiment 6, which examines the effect of automatic state construction using the decision stump algorithm. The results from three different tests of the decision stump algorithm are compared to the basic dynamic scripting results. DS refers to the learning team that makes use of regular dynamic scripting. Stump@X refers to a learning team that makes use of the decision stump algorithm to create abstract states one time, after X episodes. StumpEveryX refers to a learning team that makes use of the Decision Stump algorithm to create states after every X episodes.

Learning Efficiency

Table 17 Mean turning point for experiment 6.

	Mean Turning Point		DS
DS	24.8	(n = 30)	
StumpEvery5	29.4	(n = 20)	X
StumpEvery10	27.8	(n = 20)	X
Stump@5	23.0	(n = 20)	p < .08

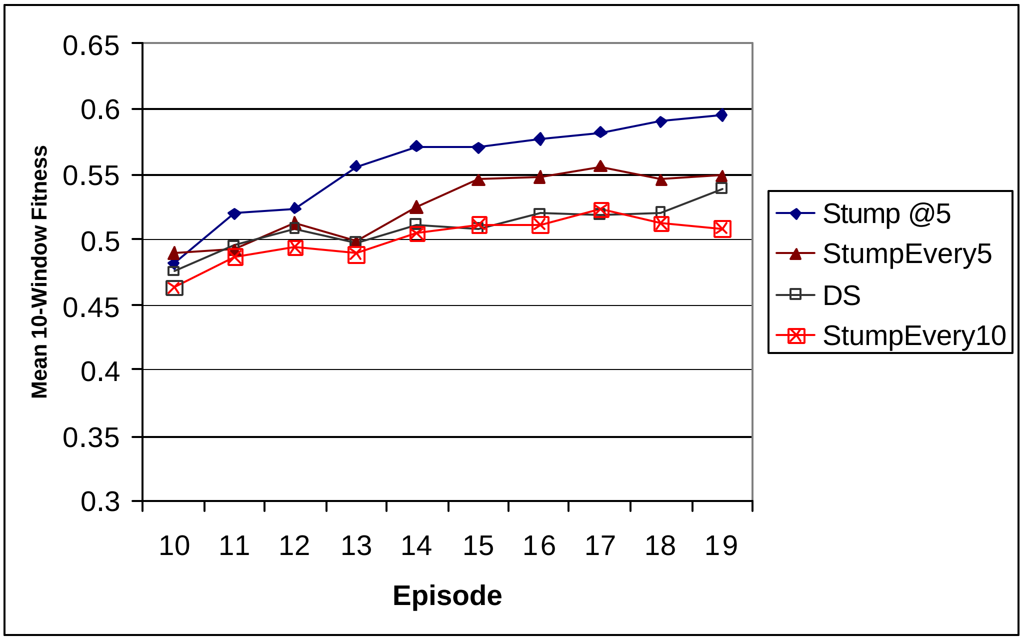


Figure 68 Mean 10-Windows fitness for experiment 6.

Variety

Table 18 Mean possible diversity for experiment 6.

	Mean Possible Diversity (n = 10)
DS	31
StumpEvery5	48
StumpEvery10	43
Stump@5	52

All three stump varieties are significantly different than DS at the p < .05 level.

Table 19 Mean selection diversity for experiment 6.

	Mean Selection Diversity (n=10)
DS	0.39
StumpEvery5	0.38
StumpEvery10	0.37
Stump@5	0.37

None of these differences are significant at the p < .05 level.

Table 20 Mean repeat diversity of selected actions for experiment 6.

	Mean Repeat Diversity (n = 10)
DS	6.44
StumpEvery5	6.85
StumpEvery10	6.78
Stump@5	6.59

None of these differences are significant at the p < .05 level.

Discussion

The results presented above validate the performance of the extended dynamic scripting algorithm. Each set of results will be discussed individually.

The first result set demonstrates the efficacy of the hierarchal extension of dynamic scripting. For both measures of learning efficiency, the general ordering from best to worst performance is: EDS > DS > HQ > Q. This met our expectations, demonstrating the superiority of the extended dynamic scripting algorithm in terms on learning efficiency.

With respect to the measures of diversity in the first set of results, the HQ and EDS algorithms had very similar results in the number of actions likely to be available for script selection (possible diversity). However, the HQ algorithm selected a significantly larger number of distinct actions in each episode and was more likely to choose evenly among the selected actions. Based on the combination of these results, the ordering from best to worst performance is: HQ > EDS > DS > Q, though HQ and EDS are very close. These results did not exactly match our expectations. Specifically, we predicted that action variety for HQ should be less than DS but found that HQ had the best action variety overall. The reason for this difference is likely due to the reduced number of learning opportunities encountered by the hierarchical learners, where the single learning problem was divided into four sub-problems (and therefore each sub-problem made only ¼ of the decision found in the overall problem).

The second set of results, Experiment 5, demonstrates that the value update scaling mechanism does not adversely affect the learning performance. That is, the scaling mechanism was designed to assist learning when there are few actions to choose from. This is not the case in NWN, so we did not expect the scaling mechanism to affect results in either direction. This was validated by the results.

The automatic state abstraction results in Experiment 6 were somewhat of a disappointment. While in one case, Stump@5, learning efficiency was significantly improved from standard dynamic scripting, the results where otherwise underwhelming. Similarly, possible diversity was significantly improved by automatic state abstraction, but the two measures of diversity in action usage did not show any significant differences. However, these results do suggest that the use of automatic state abstraction does not have a detrimental effect on either the learning efficiency or on the action variety.

Overall, the results from demonstrating the extended dynamic scripting in a commercial computer game suggest that by integrating extended dynamic scripting with SimBionic, we should expect greater learning efficiency than with the other tested learning algorithms and that the learning SimBionic architecture will also still perform well with respect to action variety. Additionally, the addition of a scaling mechanism to the value update mechanism should not adversely affect performance in the case where there are many actions to choose from. Finally, the results on automatic state abstraction are ambiguous. It is unclear whether the predicted slight improvement in performance would be worth the computational costs associated with this feature.

CHAPTER VI
CONCLUSION

A primary goal of the research in this dissertation has been to create a useful framework for game developers, where they can easily define adaptive behaviors that utilize machine learning techniques. This goal has been achieved, as described in this dissertation, by building on the dynamic scripting algorithm and embedding the result within a graphical behavior modeling environment. The first section in this chapter is devoted to summarizing the performance results found in this dissertation and discussing their importance in demonstrating the utility of this research. The second section of this chapter describes future work, aimed at improving the performance of EDS, further demonstrating its utility, and improving accessibility to the SimBionic behavior modeling environment and EDS.

Discussion

Dynamic scripting was designed specifically for modern computer games like Neverwinter Nights, where the set of available actions in a choice point meet the following requirements. First, there should exist some actions that intuitively can be assigned a high priority and that contain a more specialized IF clause. That is, there should be some rules that the behavior author can identify as important in particular situations, e.g. using a *heal* action when a character is low on health. Second, there also should be a number of actions, given lower priority, which are applicable in most situations. The standard dynamic scripting algorithm performed better than the standard Q-learning reinforcement learning algorithm in the two abstract games that met these conditions, Anwn and the modified version of Get the Ogre, as well as in the actual game Neverwinter Nights. These three games demonstrate the utility provided by the standard dynamic scripting algorithm, which allows the behavior author to apply domain knowledge in the form of priorities and specialized IF-clauses for actions.

Extended Dynamic Scripting implements two additional methods by which behavior authors can utilize domain knowledge to improve learning performance: manual state abstraction and task decomposition. The Weather and Anwn abstract games both provide evidence for the utility of manual state abstraction. In the Weather game, standard dynamic scripting is unable to perform better than randomly due to the following action constraints: there are only a small number of actions, there are no priorities to distinguish between the actions, and there are no specializing IF clauses. However, a small amount of domain knowledge in the form of manual state abstraction is used to quickly boost prediction performance from random (50%) to 70% correct. This game demonstrates a type of learning problem where, due to the nature of the available actions, EDS is applicable but standard dynamic scripting is not. Even in the Anwn game, where standard dynamic scripting outperforms Q-learning, the version of EDS that includes manual state abstraction greatly outperforms standard dynamic scripting. Additional evidence for this was provided by the Neverwinter Nights demonstration, where manual state abstraction resulted in better performance for EDS than standard dynamic scripting.

The Resource Gathering game demonstrates the usefulness of task decomposition, the second method for introducing domain knowledge in EDS. In this game, the standard dynamic scripting algorithm was unable to solve the problem with episodic rewards. While knowledge of action priorities could be successfully used in this domain to solve the problem much faster than the Random player when immediate rewards were introduced, the algorithm was still unable to demonstrate effective learning because the actions lacked specialized IF clauses that could be used to choose between actions of similar priority included in the same script. The EDS player that makes use of a task hierarchy, without action priorities or IF clauses, was quickly able to learn to complete the problem in fewer steps (on average) than the Q-learning player. Additionally, the graphical modeling language provided an intuitive representation of the task decomposition.

Extended Dynamic Scripting also introduces a way in which learning efficiency can be improved by learning about the domain through the automatic construction of abstract state trees, rather than entering domain knowledge manually. In all three games where automatic state abstraction was applied, the EDS algorithm outperformed the standard dynamic scripting algorithm. In both Weather and Get the Ogre, standard dynamic scripting was unable to learn to solve the problem, performing at the random level in Weather and worse than random in Get the Ogre. In both cases EDS was able to solve the problem utilizing automatic state abstraction, overcoming problems that could not be solved with the standard algorithm due to the nature of the available actions. In comparison with the standard Q-learning reinforcement learning algorithm, EDS performs slightly worse than Q-learning in the Weather game and better than Q-learning in Get the Ogre. EDS with automatic state abstraction even performs significantly better than the original dynamic scripting algorithm in the Anwn domain, where dynamic scripting alone does reasonably well.

Taken together, the results from the four abstract games and one commercial game clearly demonstrate that the dynamic scripting extensions described in this dissertation improve upon the basic algorithm in the following ways:

- **Increased learning performance:** This is shown through faster learning and improved scores for EDS in all four games. This was especially apparent in the ability of EDS to quickly take advantage of its early experience to improve performance in just a few trials.
- **Increased applicability and flexibility:** By allowing additional means for including domain knowledge in the form of manual state abstractions and task hierarchies and through automatic state abstraction in cases where domain knowledge does not exist, EDS can solve types of problem than dynamic scripting alone could not.
- **Acceptable computational cost:** The state abstraction feature of EDS has significant computational cost, but this cost was acceptable for the abstract and commercial games used in this research. Given the typical number of agents, how quickly they need to make decisions, the size of the state space, the number of historical decision instances, and the known optimizations, we expect that this cost will be acceptable for most commercial computer games.

Future Work

There are three particular areas of interest for future work on EDS, the first two of which are discussed at length below. The first is aimed at improving the performance automatic state specialization in EDS. The second area of future work involves demonstrating the utility of EDS in a simulation-based training environment. The third area involves improving accessibility to the EDS environment by investigating the possibility of changing SimBionic from a commercial product into an open-source project (in progress).

Improving the Performance of Automatic State Specialization

One issue to consider in the future is how to determine when to build the initial abstract state tree. A classification tree is constructed with the idea of maximizing the prediction of the reward, which does not necessarily mean that the dynamically generated scripts will differ among the leaf nodes once the state tree is created and retrained. That is, we can create a tree at any time but it may not be effective in producing the desired outcome where the scripts differ among the leaf nodes. We need a relative measure for testing the effectiveness of a classification tree in the context of the dynamic scripting algorithm. This should allow us to compare the existing action values (when no abstract state tree exists) to a newly created tree to determine if the new tree is likely to produce distinct action scripts. This algorithm could then be used to automatically determine when to create the initial abstract state true.

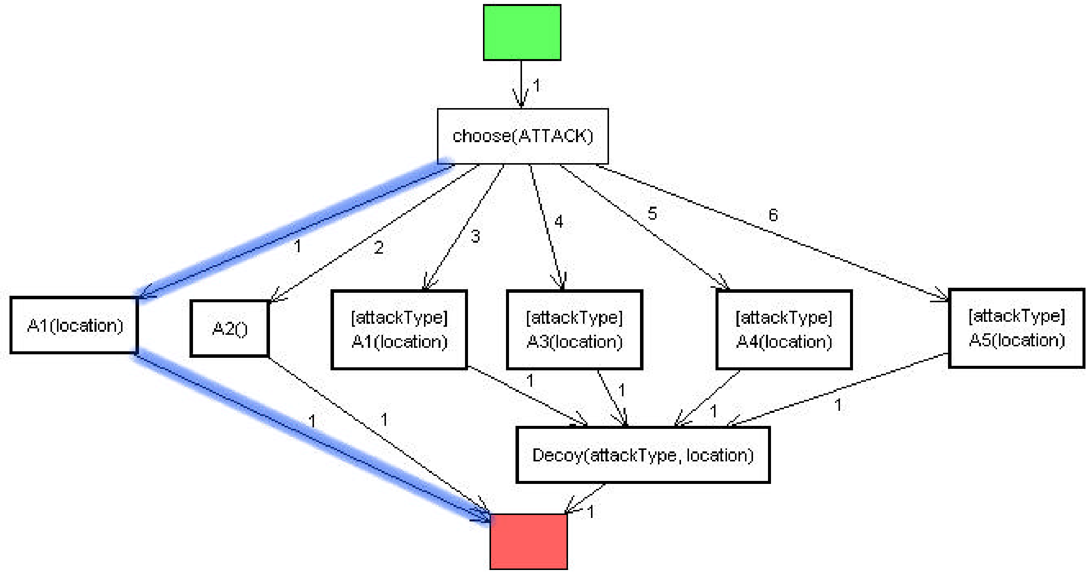
Another issue to consider is the re-creation of abstract state trees. In Weather and Anwn, creating the trees every *n* episodes resulted in better performance than creating a tree at a particular location. However, as seen in Get the Ogre, creating the tree when rewards are the exact same across leaf nodes is detrimental to performance. It would be interesting to compare the re-creation of trees as performed in this dissertation to the continuous improvement of trees as implemented in the U-Tree algorithm (McCallum, 1996). Given the utility of the Decision Stump algorithm in these games, one simple way to implement this would be to “stack” Decision Stumps. The research question here is: would extending existing trees match the performance of creating trees in Anwn and Weather while at the same time improving performance in the Get the Ogre game?

Finally, as demonstrated in the Anwn abstract games, there are situations were the construction of abstract state trees (and possibly extending them) could be detrimental to the agent in terms of computational performance. An algorithm that determines when to build (or extend) trees should also consider the computational costs, weighed against the predicted performance boost. For example, one simple way of limiting the computation time spent on creating and training trees would be to consider only a limited amount of the historic knowledge (e.g. 100 instances). While a tuneable parameter like this may be acceptable, it would be more useful for the behavior author to specify the amount of time available to spend on state abstraction (e.g. 10ms / update) and then for the state abstraction algorithm to automatically tune the parameter to meet this requirement.

Simulation-Based Training Domain

We have begun to explore the use of EDS to create adaptive adversaries as part of a simulation-based training system (Jensen, Ludwig, Proctor, Patrick, & Wong, 2008). In this work, SimBionic would be used to control the behavior of avatars in a massively multi-player online game (MMOG), where the overall training objective is to realistically challenge and surprise the human trainees in a quick and efficient manner. Extended dynamic scripting would be used to create behaviors that adaptively determine the best training scenario configuration to challenge the team of human players. The possible training scenarios would be defined as a hierarchical set of choice points, where the objective is to learn to select the scenario configurations most likely to succeed against the current tream. The following describes the design of the SimBionic behaviors for this training problem.

An abstracted version of the top level tactical behavior is shown in Figure 69, giving example behavior structure without the domain-specific content developed during this research project. The ATTACK choice point chooses the A1 adversarial tactic, as shown by the bold highlighting. The choice point selection is highlighted, where the A1 tactic is selected.



1
choose(ATTACK)
1
2
3
4
5
6
A1(location)
A2()
[attackType] A1(location)
[attackType] A3(location)
[attackType] A4(location)
[attackType] A5(location)
1
1
1
1
1
1
Decoy(attackType, location)
1

Figure 69 Main adaptive behavior.

After the main behavior chooses the A1 attack type, control is transferred to the A1 sub-behavior, seen in Figure 70. The choice point for the A1 tactic must choose between a number of different ways to carry out the A1 tactic. In this instance, A1_2 is chosen and control transfers to the A1_2 sub-behavior.

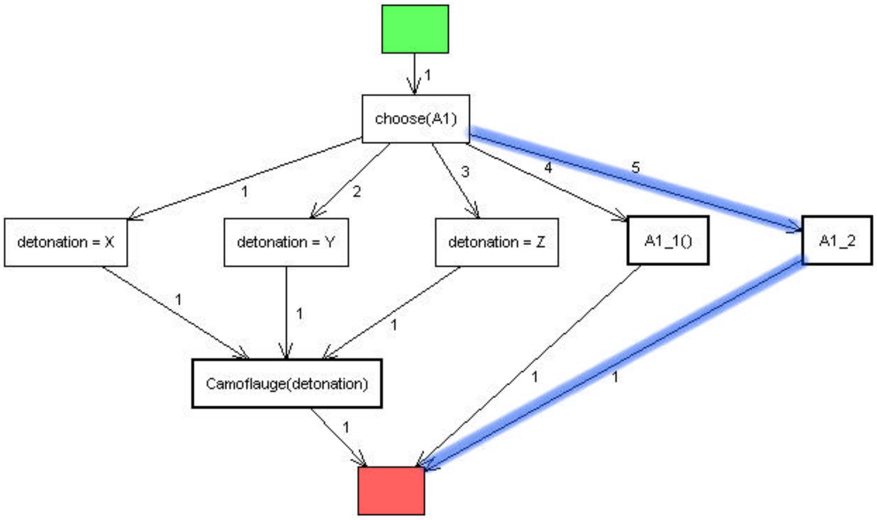


Figure 70 A1 sub-behavior.

The behavior in Figure 70 chooses the specific elements of the A1 tactic. The A1_2 behavior (Figure 71) is responsible for choosing the specific location of the A1_2 tactic in the simulated world from a number of pre-determined locations. A primitive action, selectA1_2ActiveFile, is then called to load all of the selections into the MMOG.

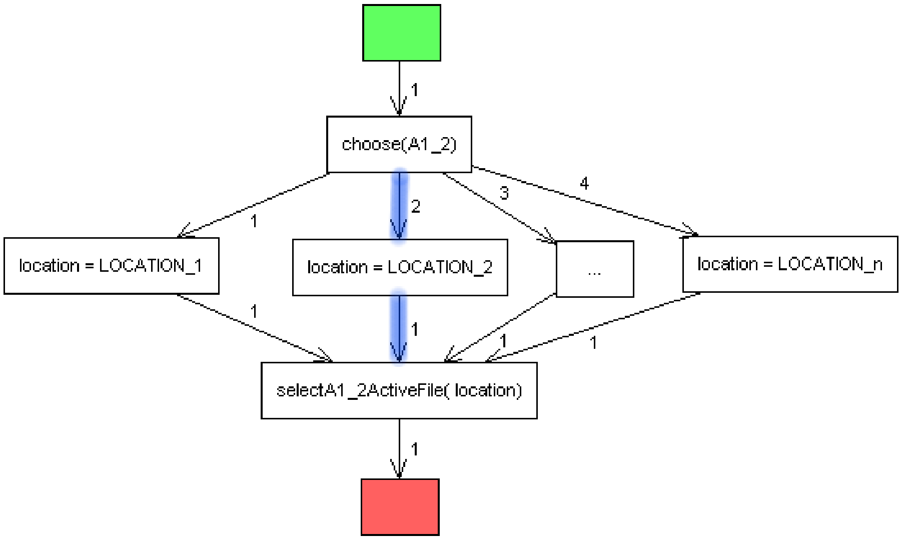


Figure 71 A1_2 sub-behavior.

The behavior in Figure 71 determines the location of the tactic elements in the simulation. After the A1_2 sub-behavior is executed, the simulation is ready for the team of trainees to begin using the simulation.

Once the simulation has ended, the results from the scenario are used to update the values associated with the actions in the choice points. This is performed with the help of a number of reward functions that supply a quantitative value that represents the success of the adversary's tactics in the scenario Reward points, encoded as part of the behavior model, use one or more of these reward functions to update a choice point's action values. They may also be integrated with specific incremental or decremental reward factors that take into account considerations such as unpredictability.

This research group was able to extract realistic domain knowledge, design the adaptive behaviors, and perform a number of demonstrations of the adaptive tactics (via role-players) against a group of six human players under a grant from the Office of Naval

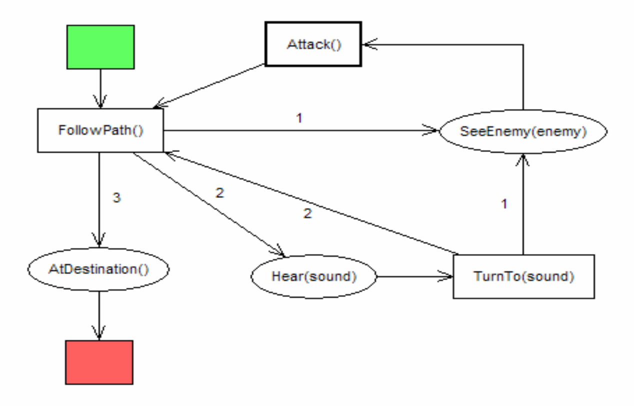
Research. Funding options to complete this project by implementing the adaptive adversary design are currently under investigation.

Coda

To summarize, this dissertation has described three specific extensions to the dynamic scripting algorithm that improved learning behavior and flexibility while imposing minimal computational cost: developing a flexible, stand alone, version of dynamic scripting that allows for hierarchical dynamic scripting; extending the context sensitivity of hierarchical dynamic scripting through automatic state construction; and performing an architectural integration where this algorithm was incorporated into an existing hierarchical behavior modeling architecture. To evaluate the effectiveness of this new dynamic scripting architecture, a tactical abstract game framework was created that allows efficient experimentation with abstract versions of modern computer games. Four different abstract games were used to measure the performance of the dynamic scripting extensions with respect to learning efficiency and computational cost. These experiments confirmed the value of our extensions to dynamic scripting and pointed out avenues for future improvements. In closing, we provided a demonstration of our dynamic scripting extensions applied to a commercially available, modern computer game.

APPENDIX A
SIMBIONIC GLOSSARY

The core of SimBionic is a visual authoring tool that allows users to draw flow chart-like diagrams that specify sequences of predicates, actions, connectors, and behaviors. Action nodes represent an action that the agent can perform in the simulation and are represented by rectangular nodes. Predicates are represented by ovals; these nodes are true or false based on the agent's perception or its internal variables. Control flows from one action to another by directed connectors. Behaviors are compositions of conditions, actions, connectors, and behaviors.



- Rectangles
  - choose(*xxx*): A unique adaptive choice point
  - Thin Rectangle: Primitive action
    - Something the agent does in the game
  - Bold Rectangle: Behavior
  - Green: Starting state
  - Red: Ending state
    - Once an ending state is reached, the behavior is terminated and control flows back up the behavior stack
  - Variable Bindings

        - Indicated by [], e.g. [attackType] binds a particular value to the variable attack type
        - A variable may also be bound by passing it into a Behavior, e.g. Attack(location) can bind a particular value to the variable location

- Ovals
    - Predicate (also referred to as Conditions): Transition only occurs when a predicate is true
        - Something the agent senses in the game
        - Can be used to limit the choices available to a choose node
- Connectors
    - Arrows that direct the state transitions through the behaviors
        - In most rectangles/ovals, transitions are attempted in order (1…n)
        - In choose(*xxx*) nodes, transition order is ignored

APPENDIX B
EDS API

This appendix contains some illustrative implementation details from the EDS Library.

Utility Methods

There are a number of utility methods that allow access to the library or perform basic functions:

```
 /**  @return the EDS singleton */
static public EDSWrapper getInstance()

/** Set the logger to use for detailed DS information. If null,
logging will not be performed.  */
public void setLogger(Logger logger)

/** Log the given msg. */
public void log(Level level, String msg)
public void log(Exception ex)
```

Choice Point Methods

A number of methods exist that allow the user to create, load, store, and access the choice points.

```
/** Load the choice point from an XML file. */
public void load(String fileName)

/** Save the DS choice point database to an XML file. */
public void save(String filename)

/** Add a choice point with an initial set of actions */
public void addChoicePoint(String choice point,
                              ArrayList<Action> actions)

/** @return the Actions associated with this choice point*/
public ArrayList<Action> getChoicePoint(String choicePoint)
```

Action Selection

The following method is used to order the actions by their assigned value. This is done by using the *softMaxSelection* method to choose the first action, a, from the set A of all available actions. Then the second action is chosen, again using the *softMaxSelection* method, from the set A’ = A – a. This continues until all of the actions are ordered.

```
/**
* Lookup the values in the choice point table and return the
* order in which to try the given transitions.
* @return the indexes to try, in order
*/
public int[] orderActions(String choice point)
```

This method uses a soft max (Boltzmann) method to choose one action from the list of actions based on its associated value, a form of fitness proportionate selection. This algorithm includes a temperature variable that can be adjusted. As the temperature increases the probability of selecting an action with a lower value increases. As the temperature decreases, it is more likely that the action with the highest value is selected.

```
/**
 * Select a value from the list using a soft max distribution.
 * @return the selected index, null if no decision is made
 */
protected Action softMaxSelection(ArrayList<Action>  actions)
```

The next method is used by a client to indicate that it selected one of the actions ordered by the *orderActions* method.

```
/** Called when an ordered action is selected */
public void actionSelected(String choicePoint, int index)
```

The EDS library uses this to create a list of *selection* objects that have occurred in each choice point. Each selection contains the choice point for which the selection was made, the *action* chosen, and a property map that contains the relevant game state for the selection. The property map is created by calling the *getSelectionState* method on the *I_EDSAdjustor* object created by the user for a particular game.

I_EDSAdjustor

In order to adjust the values associated with each action, a game specific component must gather the game state at appropriate times, supply a reward when the value adjustment is initiated for a choice point, and provide information on the min/max values associated with a choice point. This minimum amount of functionality is provided to the DS library by developing a Java class that implements this interface.

```
/**
 * When selection occurs in a choice point, get any state
 * information that would be needed for the reward function.
 *
 * @return a serialzable Map of state information
 */
public Map getGameStateInformationOnSelect(String choicePoint);

/**
 * When a reward is requested, this method returns the reward
 * @param actionSelection a list of the action selections that
 * have occurred
 * @return the reward for the given choicePoint
 */
public double getReward(String choicePoint, ArrayList<Selection>
                                          actionSelections);

/** @return maximum action value for this choice point */
public double getMaxChoicePointValue(String choicePoint);

/** @return minimum action value for this choice point*/
public double getMinChoicePointValue(String choicePoint);
```

Adjusting Action Values

The *adjustValues* method examines the list of actions that have occurred for the given choice point. If only one action has occurred, an immediate reward is applied. If more than one action has occurred then an episodic award is applied. In either case, the reward function is the same. The value of action a in choice point c, V(c,a), must always be <= the maximum reward and >= the minimum reward as defined by the *I_EDSAdjustor* for the particular game and choice point.

```
/** Use the DS algorithm to adjust the values based on the
 * actions selected for the current choice point */
public void adjustValues(String choicePoint)
```

For immediate and episodic rewards, the value update function methods is similar to the standard DS update function (Spronck et al., 2006). In the case of an action that was selected, the update function is V(c,a) = V(c,a) + reward. Actions that were not selected in the episode receive a reverse compensation, where compensation = - ((#selectedActions / #unselectedActions) * reward) and the update function is V(c,a)

= V(c,a) + compensation. Below is the pseudo code for the adjusting the value of a single action.

```
private double adjustValue(Action a, double adjustment,
                        double MAX_VALUE, double MIN_VALUE)
{
      double value = a.getValue() + adjustment;
      double remainder = 0;
      if(value > MAX_VALUE)
      {
            remainder = value - MAX_VALUE;
            value = MAX_VALUE;
      }
      else
      if(value < MIN_VALUE)
      {
            remainder = MIN_VALUE - value;
            value = MIN_VALUE;
      }
      a.setValue( value );
      return remainder;
}
```

The following algorithm shows how the reward and compensation is applied to the list of actions performed in this choice point (completedList) and the list of action not performed (notCompletedList). These two sets of actions are distinct, and their union is equal to the set of all action available to the character (characterActions). After updated, the list of completed actions is emptied and the list of unused actions is set to equal the list of actions available to the character.

```
double reward = myDynamicScriptingAdjustor.getReward(choicePoint,
                                                  selectionList);
double compensation =  -( completedList.size()*reward )
                        / (notCompletedList.size());
for(Action a : completedList)
      remainder += adjustValue(a, reward);
for(Action a : notCompletedList)
      remainder += adjustValue(a, compensation);
distributeRemainder(completedList, notCompletedList, remainder);
completedList.clear();
notCompletedList.addAll(characterActions);
```

The remainder caused by either going below the minimum value of an action or above the maximum value in the *adjustValue* method is redistributed among the other actions in the *distributeRemainder* method. The net result of reward, compensation, and remainder distribution is that the sum of the action values is constant, it is only the distribution of values that changes.

APPENDIX C
EXAMPLE NWN CHARACTER STATE

```
<root>
  <HEARTBEAT>0</HEARTBEAT>
  <NEAREST />
  <NEAREST_MAGE />
  <NEAREST_PRIEST />
  <NEAREST_FIGHTER />
  <NUMBER_ENEMIES>0</NUMBER_ENEMIES>
  <NUMBER_MELEE_ENEMIES>0</NUMBER_MELEE_ENEMIES>
  <MAXHITPOINTS>64</MAXHITPOINTS>
  <HITPOINTS>64</HITPOINTS>
  <GOOD_EVIL>1</GOOD_EVIL>
  <FIGHTER>1</FIGHTER>
  <ROGUE>0</ROGUE>
  <MAGE>0</MAGE>
  <PRIEST>0</PRIEST>
  <HEAL_SELF>1</HEAL_SELF>
  <BUFF_SELF>1</BUFF_SELF>
  <SPELL_DEATH_WARD>0</SPELL_DEATH_WARD>
  <SPELL_FREEDOM_OF_MOVEMENT>0</SPELL_FREEDOM_OF_MOVEMENT>
  <SPELL_REGENERATE>0</SPELL_REGENERATE>
  <SPELL_HASTE>0</SPELL_HASTE>
  <SPELL_TIME_STOP>0</SPELL_TIME_STOP>
  <SPELL_HIGH_ABSORPTION>0</SPELL_HIGH_ABSORPTION>
  <SPELL_HIGH_ANTIINVIS>0</SPELL_HIGH_ANTIINVIS>
  <SPELL_HIGH_ANTIMIND>0</SPELL_HIGH_ANTIMIND>
  <SPELL_HIGH_ELEMENTABSORPTION>0</SPELL_HIGH_ELEMENTABSORPTION>
  <SPELL_HIGH_DAMAGEABSORPTION>0</SPELL_HIGH_DAMAGEABSORPTION>
  <SPELL_HIGH_DAMAGEAREA__FIGHTER>0</SPELL_HIGH_DAMAGEAREA__FIGHTER>
  <SPELL_HIGH_DAMAGEAREA__MAGE>0</SPELL_HIGH_DAMAGEAREA__MAGE>
  <SPELL_HIGH_DAMAGEAREA__PRIEST>0</SPELL_HIGH_DAMAGEAREA__PRIEST>
  <SPELL_HIGH_DAMAGEAREA__NEAREST>0</SPELL_HIGH_DAMAGEAREA__NEAREST>
  <SPELL_HIGH_SUMMON__FIGHTER>0</SPELL_HIGH_SUMMON__FIGHTER>
  <SPELL_HIGH_SUMMON__MAGE>0</SPELL_HIGH_SUMMON__MAGE>
  <SPELL_HIGH_SUMMON__PRIEST>0</SPELL_HIGH_SUMMON__PRIEST>
  <SPELL_HIGH_SUMMON__NEAREST>0</SPELL_HIGH_SUMMON__NEAREST>
  <SPELL_HIGH_CURSEAREA__FIGHTER>0</SPELL_HIGH_CURSEAREA__FIGHTER>
  <SPELL_HIGH_CURSEAREA__MAGE>0</SPELL_HIGH_CURSEAREA__MAGE>
  <SPELL_HIGH_CURSEAREA__PRIEST>0</SPELL_HIGH_CURSEAREA__PRIEST>
  <SPELL_HIGH_CURSEAREA__NEAREST>0</SPELL_HIGH_CURSEAREA__NEAREST>
  <SPELL_HIGH_CLOUDDAMAGE__FIGHTER>0</SPELL_HIGH_CLOUDDAMAGE__FIGHTER>
  <SPELL_HIGH_CLOUDDAMAGE__MAGE>0</SPELL_HIGH_CLOUDDAMAGE__MAGE>
  <SPELL_HIGH_CLOUDDAMAGE__PRIEST>0</SPELL_HIGH_CLOUDDAMAGE__PRIEST>
```

```
<SPELL_HIGH_CLOUDDAMAGE__NEAREST>0</SPELL_HIGH_CLOUDDAMAGE__NEAREST>
  <SPELL_HIGH_CLOUDCURSE__FIGHTER>0</SPELL_HIGH_CLOUDCURSE__FIGHTER>
  <SPELL_HIGH_CLOUDCURSE__MAGE>0</SPELL_HIGH_CLOUDCURSE__MAGE>
  <SPELL_HIGH_CLOUDCURSE__PRIEST>0</SPELL_HIGH_CLOUDCURSE__PRIEST>
  <SPELL_HIGH_CLOUDCURSE__NEAREST>0</SPELL_HIGH_CLOUDCURSE__NEAREST>
  <SPELL_HIGH_BREACH__FIGHTER>0</SPELL_HIGH_BREACH__FIGHTER>
  <SPELL_HIGH_BREACH__MAGE>0</SPELL_HIGH_BREACH__MAGE>
  <SPELL_HIGH_BREACH__PRIEST>0</SPELL_HIGH_BREACH__PRIEST>
  <SPELL_HIGH_BREACH__NEAREST>0</SPELL_HIGH_BREACH__NEAREST>
  <SPELL_HIGH_CONE__FIGHTER>0</SPELL_HIGH_CONE__FIGHTER>
  <SPELL_HIGH_CONE__MAGE>0</SPELL_HIGH_CONE__MAGE>
  <SPELL_HIGH_CONE__PRIEST>0</SPELL_HIGH_CONE__PRIEST>
  <SPELL_HIGH_CONE__NEAREST>0</SPELL_HIGH_CONE__NEAREST>
  <SPELL_HIGH_DAMAGE__FIGHTER>0</SPELL_HIGH_DAMAGE__FIGHTER>
  <SPELL_HIGH_DAMAGE__MAGE>0</SPELL_HIGH_DAMAGE__MAGE>
  <SPELL_HIGH_DAMAGE__PRIEST>0</SPELL_HIGH_DAMAGE__PRIEST>
  <SPELL_HIGH_DAMAGE__NEAREST>0</SPELL_HIGH_DAMAGE__NEAREST>
  <SPELL_HIGH_CURSE__FIGHTER>0</SPELL_HIGH_CURSE__FIGHTER>
  <SPELL_HIGH_CURSE__MAGE>0</SPELL_HIGH_CURSE__MAGE>
  <SPELL_HIGH_CURSE__PRIEST>0</SPELL_HIGH_CURSE__PRIEST>
  <SPELL_HIGH_CURSE__NEAREST>0</SPELL_HIGH_CURSE__NEAREST>
  <TALENT_HEAL>0</TALENT_HEAL>
  <TALENT_HEALING_SELF>1</TALENT_HEALING_SELF>
  <TALENT_ADVANCED_PROTECT_SELF>0</TALENT_ADVANCED_PROTECT_SELF>
  <TALENT_USE_PROTECTION_SELF>0</TALENT_USE_PROTECTION_SELF>
  <TALENT_USE_PROTECTION_OTHERS>0</TALENT_USE_PROTECTION_OTHERS>
  <TALENT_USE_ENHANCEMENT_SELF>0</TALENT_USE_ENHANCEMENT_SELF>
  <TALENT_ENHANCE_OTHERS>0</TALENT_ENHANCE_OTHERS>
  <TALENT_MELEE_ATTACKED>0</TALENT_MELEE_ATTACKED>
  <TALENT_RANGED_ATTACKERS>0</TALENT_RANGED_ATTACKERS>
  <TALENT_RANGED_ENEMIES>0</TALENT_RANGED_ENEMIES>
  <TALENT_SUMMON_ALLIES>0</TALENT_SUMMON_ALLIES>
  <TALENT_SPELL_ATTACK>0</TALENT_SPELL_ATTACK>
  <TALENT_MELEE_ATTACK>1</TALENT_MELEE_ATTACK>
  <TALENT_CURE_CONDITION>0</TALENT_CURE_CONDITION>
  <TALENT_USE_TURNING>0</TALENT_USE_TURNING>
  <TALENT_SNEAK_ATTACK>0</TALENT_SNEAK_ATTACK>
</root>
```

APPENDIX D
NWN ACTIONS BY CHARACTER CLASS

This appendix contains a list of actions for each of the four character classes. Actions in the list below are sorted first by priority (HIGHEST, HIGH, MEDIUM, LOW, LOWEST) and secondarily by action id number (0…n). The FILLER actions are never valid and serve only to allow a script to contain empty actions.

PRIEST ACTIONS

```
HEAL_SELF (0)
SPELL_DEATH_WARD (2)
SPELL_FREEDOM_OF_MOVEMENT (3)
SPELL_REGENERATE (4)
SPELL_HASTE (5)
HEAL_SELF (7)
SPELL_HIGH_SUMMON__NEAREST (1)
SPELL_HIGH_SUMMON__MAGE (2)
SPELL_HIGH_DAMAGEAREA__NEAREST (3)
SPELL_HIGH_DAMAGEAREA__MAGE (4)
SPELL_HIGH_CLOUDDAMAGE__NEAREST (5)
SPELL_HIGH_CLOUDDAMAGE__MAGE (6)
SPELL_HIGH_CLOUDCURSE__NEAREST (7)
SPELL_HIGH_CLOUDCURSE__MAGE (8)
SPELL_HIGH_CURSEAREA__NEAREST (9)
SPELL_HIGH_CURSEAREA__MAGE (10)
SPELL_HIGH_CONE__NEAREST (11)
SPELL_HIGH_CONE__MAGE (12)
SPELL_HIGH_DAMAGE__NEAREST (13)
SPELL_HIGH_DAMAGE__MAGE (14)
SPELL_HIGH_CURSE__NEAREST (15)
SPELL_HIGH_CURSE__MAGE (16)
SPELL_HIGH_ANTIINVIS (17)
SPELL_HIGH_ANTIMIND (18)
SPELL_HIGH_ELEMENTABSORPTION (19)
SPELL_HIGH_DAMAGEABSORPTION (29)
FILLER (30)
TALENT_HEAL (0)
TALENT_HEALING_SELF (2)
TALENT_USE_ENHANCEMENT_SELF (6)
TALENT_ENHANCE_OTHERS (7)
TALENT_MELEE_ATTACKED (8)
TALENT_RANGED_ENEMIES (10)
```

```
TALENT_SUMMON_ALLIES (11)
TALENT_SPELL_ATTACK (12)
TALENT_MELEE_ATTACK (13)
TALENT_CURE_CONDITION (14)
TALENT_USE_TURNING (15)
TALENT_MELEE_ATTACKED (16)
TALENT_SPELL_ATTACK (24)
FILLER (30)
TALENT_RANGED_ENEMIES (2)
TALENT_SPELL_ATTACK (3)
TALENT_MELEE_ATTACK (4)
TALENT_MELEE_ATTACKED (5)
FILLER (30)
TALENT_MELEE_ATTACK (0)
FILLER (30)
```

MAGE ACTIONS

```
HEAL_SELF (0)
SPELL_HASTE (5)
SPELL_TIME_STOP (6)
HEAL_SELF (7)
SPELL_HIGH_ABSORPTION (0)
SPELL_HIGH_SUMMON__NEAREST (1)
SPELL_HIGH_SUMMON__MAGE (2)
SPELL_HIGH_DAMAGEAREA__NEAREST (3)
SPELL_HIGH_DAMAGEAREA__MAGE (4)
SPELL_HIGH_CLOUDDAMAGE__NEAREST (5)
SPELL_HIGH_CLOUDDAMAGE__MAGE (6)
SPELL_HIGH_CLOUDCURSE__NEAREST (7)
SPELL_HIGH_CLOUDCURSE__MAGE (8)
SPELL_HIGH_CURSEAREA__NEAREST (9)
SPELL_HIGH_CURSEAREA__MAGE (10)
SPELL_HIGH_CONE__NEAREST (11)
SPELL_HIGH_CONE__MAGE (12)
SPELL_HIGH_DAMAGE__NEAREST (13)
SPELL_HIGH_DAMAGE__MAGE (14)
SPELL_HIGH_CURSE__NEAREST (15)
SPELL_HIGH_CURSE__MAGE (16)
SPELL_HIGH_ANTIINVIS (17)
SPELL_HIGH_ANTIMIND (18)
SPELL_HIGH_ELEMENTABSORPTION (19)
SPELL_HIGH_BREACH__NEAREST (20)
SPELL_HIGH_BREACH__MAGE (21)
SPELL_HIGH_DAMAGEABSORPTION (29)
FILLER (30)
TALENT_HEALING_SELF (2)
TALENT_USE_ENHANCEMENT_SELF (6)
TALENT_ENHANCE_OTHERS (7)
```

```
TALENT_MELEE_ATTACKED (8)
TALENT_RANGED_ENEMIES (10)
TALENT_SUMMON_ALLIES (11)
TALENT_SPELL_ATTACK (12)
TALENT_MELEE_ATTACK (13)
TALENT_SPELL_ATTACK (24)
FILLER (30)
TALENT_RANGED_ENEMIES (2)
TALENT_SPELL_ATTACK (3)
TALENT_MELEE_ATTACK (4)
FILLER (30)
TALENT_MELEE_ATTACK (0)
FILLER (30)
```

ROGUE ACTIONS

```
HEAL_SELF (0)
BUFF_SELF (1)
HEAL_SELF (7)
BUFF_SELF (17)
MELEE (22)
MELEE (23)
RANGED (24)
RANGED (25)
MELEE (26)
RANGED (27)
FILLER (30)
TALENT_HEALING_SELF (2)
TALENT_USE_ENHANCEMENT_SELF (6)
TALENT_MELEE_ATTACK (13)
TALENT_SNEAK_ATTACK (18)
TALENT_MELEE_ATTACK (23)
FILLER (30)
TALENT_MELEE_ATTACK (4)
FILLER (30)
TALENT_MELEE_ATTACK (0)
FILLER (30)
```

FIGHTER ACTIONS

```
HEAL_SELF (0)
BUFF_SELF (1)
HEAL_SELF (7)
BUFF_SELF (17)
MELEE (22)
MELEE (23)
RANGED (24)
RANGED (25)
MELEE (26)
```

```
RANGED (27)
FILLER (30)
TALENT_HEALING_SELF (2)
TALENT_USE_ENHANCEMENT_SELF (6)
TALENT_MELEE_ATTACK (13)
TALENT_SNEAK_ATTACK (18)
TALENT_MELEE_ATTACK (23)
FILLER (30)
TALENT_MELEE_ATTACK (4)
FILLER (30)
TALENT_MELEE_ATTACK (0)
FILLER (30)
```

BIBLIOGRAPHY

Aha, D. W., Molineaux, M., & Ponsen, M. (2005). Learning to win: Case-based plan selection in a real-time strategy game. In *Proceedings of the Sixth International Conference on Case-Based Reasoning (ICCBR-05)* (pp. 15-20). Chicago, IL: Springer.

Andrade, G., Ramalho, G., Santana, H., & Corruble, V. (2005). Extending reinforcement learning to provide dynamic game balancing. In D. W. Aha, M.-A. H.M. & M. van Lent (Eds.), *Reasoning, Representation, and Learning in Computer Games: Proceedings of the IJCAI Workshop (Technical Report AIC-05-127)*. Washington, DC: Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence.

Andre, D., & Russell, S. (2002). *State abstraction for programmable reinforcement learning agents.* Paper presented at the AAAI-02, Edmonton, Alberta.

Asadpour, M., Ahmadabadi, M. N., & Siegwart, R. (2006). *Reduction of learning time for robots using automatic state abstraction.* Paper presented at the First European Symposium on Robotics, Palermo, Italy.

Au, M., & Maire, F. (2004). *Automatic State Construction using Decision Tree for Reinforcement Learning Agents.* Paper presented at the International Conference on Computational Intelligence for Modelling, Control and Automation, Gold Coast, Australia.

Bakkes, S., Spronck, P., & Postma, E. (2005). Best-response learning of team behaavior in Quake III. In D. W. Aha, M.-A. H.M. & M. van Lent (Eds.), *Reasoning, Representation, and Learning in Computer Games: Proceedings of the IJCAI Workshop (Technical Report AIC-05-127)*. Washington, DC: Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence.

Barto, A., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Application, 13*(1-2), 41-77.

Best, B., Lebiere, C., & Scarpinatto, C. (2002). *A model of synthetic opponents in MOUT training simulations using the ACT-R cognitive architecture.* Paper presented at the Eleventh Conference on Computer Generated Forces and Behavior Representation, Orlando, FL.

Bioware. (2002). Neverwinter Nights v1.68 [Computer Program].

Bowling, M., Fürnkranz, J., Graepel, T., & Musick, R. (2006). Machine learning and games. *Machine Learning, 63*(3), 211-215.

Chapman, D., & Kaelbling, L. P. (1991). *Learning from delayed reinforcement in a complex domain.* Paper presented at the Twelfth International Joint Conference on Artificial Intelligence, Sydney, Australia.

CS: Source beta. (2004). Retrieved 2008, October 15, from www.cosmicbuddha.com/blog/archives/2004/08/

Dahlbom, A. (2006). *An adaptive AI for real-time strategy games.* Unpublished MS Thesis, Högskolan i Skövde.

Dahlbom, A., & Niklasson, L. (2006). *Goal-directed hierarchical dynamic scripting for RTS games.* Paper presented at the Second Artificial Intelligence in Interactive Digital Entertainment, Marina del Rey, California.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research, 13*, 227-303.

Flemming, J. (2008). Game brains. *Game Developer Magazine, August, 2008*.

Fu, D., & Houlette, R. (2002). Putting AI in Entertainment: An AI Authoring Tool for Simulation and Games. *IEEE Intelligent Systems*(July-August), 81-84.

Fu, D., & Houlette, R. (2003a). Conctructing a decision tree based on past experiences. In *AI Game Programming Wisdom 2*: Charles River Media.

Fu, D., & Houlette, R. (2003b). The ultimate guide to FSMs in games. In S. Rabin (Ed.), *AI Game Programming Wisdom 2*. Boston, MA: Charles River Media.

Fu, D., Houlette, R., & Ludwig, J. (2007). *An AI Modeling Tool for Designers and Developers.* Paper presented at the IEEE Aerospace Conference, Big Sky, MT.

Genesereth, M., & Love, N. (2005). Game Description Language [Electronic Version]. Retrieved October 15, 2008 from http://games.stanford.edu/competition/misc/aaai.pdf.

Hoang, H., Lee-Urban, S., & Muñoz-Avila, H. (2005). *Hierarchical Plan Representations for Encoding Strategic Game AI.* Paper presented at the Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05), Marina del Rey, CA.

Jensen, R., Ludwig, J., Proctor, M., Patrick, J., & Wong, W. (2008). *Adaptive behavior model for asymmetric adversaries.* Paper presented at the Interservice/Industry Training, Simulation and Education Conference 2008, Orlando, FL.

Jones, R. M., Laird, J. E., Nielsen, P. E., Coulter, K. J., Kenny, P., & Koss, F. V. (1999). Automated intelligent pilots for combat flight simulation. *AI Magazine, 20*(1), 27-41.

Jonsson, A., & Barto, A. (2000). Automated state abstractions for options using the U-tree algorithm. *Advances in Nueral Information Processing Systems, 13*, 1054-1060.

Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence, 101*, 99-134.

Khardon, R. (1999). Learning action strategies for planning domains. *Artificial Intelligence, 113*(1-2), 125-148.

Laird, J. E., & Congden, C. B. (2005). The Soar User's Manual Version 8.6 Edition 1. Retrieved June 1, 2005, from http://sitemaker.umich.edu/soar/soar_software_downloads

Laird, J. E., & van Lent, M. (2001). Human-level AI's killer application: Interactive computer games. *AI Magazine, 22*(2), 15-26.

Laird, J. E., & van Lent, M. (2005). *Tutorial on Machine Learning for Computer Games.* Paper presented at the Game Developer Conference 2005, San Francisco, CA.

Langley, P., & Choi, D. (2006). *A unified cognitive architecture for physical agents.* Paper presented at the Twenty-First National Conference on Artificial Intelligence, Boston, MA.

Ludwig, J., & Farley, A. (2007). A learning infrastructure for improving agent performance and game balance. In *Optimizing Player Satisfaction: Papers from the 2007 AIIDE Workshop*. Stanford, CA: AAAI Press.

Ludwig, J., & Houlette, R. (2006). *Intelligent behaviors for simulated entities.* Paper presented at the Interservice/Industry Training, Simulation and Education Conference 2006, Orlando, FL.

Maclin, R., Shavlik, J., Walker, T., & Torrey, L. (2005). Knowledge-based support-vector regression for reinforcement learning. In D. W. Aha, M.-A. H.M. & M. van Lent (Eds.), *Reasoning, Representation, and Learning in Computer Games: Proceedings of the IJCAI Workshop (Technical Report AIC-05-127)*. Washington, DC: Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence.

Manslow, J. (2002). Learning and adaptation. In S. Rabin (Ed.), *AI Programming Wisdom*. Boston, MA: Charles River Media.

Marthi, B. ALisp tutorial. Retrieved July 09, 2008, from http://www.cs.berkeley.edu/~bhaskara/alisp/alisp-system/doc/rl.html

Marthi, B., Russell, S., & Latham, D. (2005). Writing stratagus-playing agents in concurrent ALisp. In D. W. Aha, M.-A. H.M. & M. van Lent (Eds.), *Reasoning, Representation, and Learning in Computer Games: Proceedings of the IJCAI Workshop (Technical Report AIC-05-127)*. Washington, DC: Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence.

McCallum, A. R. (1996). Learning to use selective attention and short-term memory in sequential tasks. In *Fourth International Conference on Simulation of Adaptive Behavior* (pp. 315-324). Cape Cod, MA: The MIT Press.

Mehta, N., Ray, S., Tadepalli, P., & Dietterich, T. G. (2008). *Automatic discovery and transfer of MAXQ hierarchies.* Paper presented at the 25th International Conference on Machine Learning, Helsinki, Finland.

Millington, I. (2006). *Artificial Intelligence for Games.* San Francisco, CA: Morgan Kaufmann.

Minsky, M. (1961). Steps toward artificial intelligence. *Proceedings of the Institute of Radio Engineers, 49*(1), 8-30.

Moore, A. W., & Atkeson, C. G. (1995). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning, 21*(3), 199-233.

Nason, S., & Laird, J. E. (2004). *Soar-RL: Integrating reinforcement learning with Soar.* Paper presented at the Sixth International Conference on Cognitive Modeling, Pittsburgh, PA.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, Mass.: Harvard University Press.

Nogueira, N. (2006). Retrieved July 01, 2008, from http://en.wikipedia.org/wiki/Image:Minimax.svg

Orkin, J. (2004). *Symbolic representation of game world state: Toward real-time planning in games.* Paper presented at the Challenges in Game AI: Papers of the AAAI'04 Workshop, San Jose, CA.

Orkin, J. (2006). *Three States and a Plan: The A.I. of F.E.A.R.* Paper presented at the Game Developers Conference 2006, San Francisco, CA.

Ponsen, M., & Spronck, P. (2004). *Improving adaptive game AI with evolutionary learning.* Paper presented at the CGAIDE 2004 International Conference on Computer Games, Reading, UK.

Ponsen, M., Spronck, P., Muñoz-Avila, H., & Aha, D. W. (2006). Knowldge Acquisition for Adaptive Game AI. *Submitted*.

Ponsen, M., Spronck, P., & Tuyls, K. (2006). *Hierarchical reinforcement learning in computer games.* Paper presented at the ALAMAS'06 Adaptive Learning and Multi-Agent Systems, Vrije Universiteit, Brussels, Belgium.

Russell, S., & Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ: Prentice-Hall.

Santamaria, A., & Warwick, W. (2008). *Modeling probabilistic category learning in a task network model.* Paper presented at the 2008 Conference on Behavior Representation in Modeling and Simulation, Providence, RI.

Shapiro, D., Langley, P., & Shachter, R. (2001). *Using background knowledge to speed reinforcement learning in physical agents.* Paper presented at the Fifth International Conference on Autonomous Agents, Montreal, CA.

Smart, D. Retrieved September 4, 2008, from http://www.gameai.com/

Spronck, P. Neverwinter Nights. Retrieved October 15, 2008, from www.spronck.net

Spronck, P., Ponsen, M., Sprinkhuizen-Kuyper, I., & Postma, E. (2006). Adaptive game AI with dynamic scripting. *Machine Learning, 63*(3), 217-248.

Sutton, R., & Barto, A. (1998). *Reinforcement Learning: An Introduction*: MIT Press.

Sutton, R., McAllester, D., Singh, S., & Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12* (pp. 1057-1063): MIT Press.

Tadepalli, P., Givan, R., & Driessens, K. (2004). *Relational reinforcement learning: An overview.* Paper presented at the ICML'04 Workshop on Relational Reinforcement Learning, Banff, Canada.

Timuri, T., Spronck, P., & van den Herik, J. (2007). *Automatic rule ordering for dynamic scripting.* Paper presented at the Artificial Intelligence in Interactive Digital Entertainment, Stanford, CA.

Ulam, P., Goel, A., Jones, J., & Murdock, W. (2005). Using model-based reflection to guide reinforcement learning. In D. W. Aha, M.-A. H.M. & M. van Lent (Eds.), *Reasoning, Representation, and Learning in Computer Games: Proceedings of the IJCAI Workshop (Technical Report AIC-05-127)*. Washington, DC: Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence.

Uther, W. T. B., & Veloso, M. M. (1998). *Tree based discretization for continious state space reinforcement learning.* Paper presented at the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, Madison, WI.

Wang, Y., & Laird, J. E. (2007). *The importance of action history in decision making and reinforcement learning.* Paper presented at the Eighth International Conference on Cognitive Modeling, Ann Arbor, MI.

Warwick, W., & Santamaria, A. (2006). *Giving up vindication in favor of application: Developing cogntively-inspired widgets for human performance modeling tools.* Paper presented at the International Conference on Cognitive Modeling, Trieste, Italy.

Witten, I. H., & Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques, 2nd Edition*. San Francisco: Morgan Kaufmann.

Wray, R. E., Laird, J. E., Nuxoll, A., Stokes, D., & Kerfoot, A. (2005). Synthetic adversaries for urban combat training. *AI Magazine, 26*(3), 82-92.

Yannakakis, G. N., & Hallam, J. (2005). A scheme for creating digital entertainment with substance. In D. W. Aha, M.-A. H.M. & M. van Lent (Eds.), *Reasoning, Representation, and Learning in Computer Games: Proceedings of the IJCAI Workshop (Technical Report AIC-05-127)*. Washington, DC: Naval Research Laboratory, Navy Center for Applied Research in Artificial Intelligence.