Using Hierarchical Dynamic Scripting to Create Adaptive Adversaries

Jeremy Ludwig 1, 2  
Arthur Farley 2

1 Stottler Henke Associates  
San Mateo, CA 94404  
ludwig@stottlerhenke.com  
2 University of Oregon  
Eugene, OR 97403  
ludwig@cs.uoregon.edu, art@cs.uoregon.edu

Keywords:  
dynamic scripting; adaptive adversaries; behavior modeling; reinforcement learning

ABSTRACT: The dynamic scripting reinforcement learning algorithm, designed for modern computer games, can be extended to improve the speed, effectiveness, and accessibility of learning in modern computer games without sacrificing computational efficiency. This paper describes two 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 and (2) an integration of this library with an existing hierarchical behavior modeling architecture. The hierarchical dynamic scripting algorithm is tested in the context of the computer game Neverwinter Nights. The results are compared to those generated by standard dynamic scripting and Q-learning variants. Finally, we discuss work in progress that utilizes the developed library for simulating asymmetric adversaries.

1 Introduction
Dynamic scripting (DS) is one example of an online reinforcement learning algorithm developed specifically to control the behavior of adversaries in games and simulations [Spronck, Ponsen, Sprinkhuizen-Kuyper, & Postma, 2006]. This reinforcement learning algorithm has been tested in both role playing games (Neverwinter Nights [Bioware, 2002]) and real-time strategy games (Wargus [The Wargus Team, 2007]), with promising results. That is, adversaries using DS quickly learn how to beat the human opponent. In this paper we look at extending the dynamic scripting algorithm to improve the speed, effectiveness, and accessibility of learning in games and simulations, without sacrificing the computational efficiency of the dynamic scripting algorithm. The first enhancement is developing a flexible, stand alone, version of dynamic scripting that allows for hierarchical dynamic scripting and the second is integrating this library into an existing hierarchical behavior modeling architecture.

First, a definition of the dynamic scripting reinforcement learning algorithm is given. Following this, the extensions to dynamic scripting are described. The following sections detail experiments undertaken in the Neverwinter Nights computer game. The final sections of the paper discuss related work and describe current efforts to utilize this learning algorithm in order to create adaptive asymmetric adversaries for military training.

1.1 Dynamic Scripting
Dynamic Scripting is one instance of a reinforcement learning algorithm [see Sutton & Barto, 1998 for an introduction]. This section outlines the DS algorithm and then compares it to the more standard Q-Learning algorithms. There are three main components to the dynamic scripting algorithm: script selection, action policy, and action value updating [Spronck et al., 2006]. Let Foo be an agent in a role-playing game, where Foo’s behavior is controlled by the dynamic scripting algorithm. Foo will perform in a number of episodes (scenarios), in which it will need to choose an action to perform during each step (round) of the episode. Foo has a set of actions it can take in an episode called characterActions, where each action has an action value. The following compares the algorithms for Q-learning and for dynamic scripting:

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 selected, based on their Q values.
- At end of episode, actions are updated using a Q-learning update function (e.g. Sarsa).
Dynamic Scripting

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

1.1.1 Script Selection

The first component of the algorithm is script selection. Before each episode, Foo selects a subset of actions to use in the episode. A free parameter, $scriptSize$, determines the size of the subset. This component uses soft-max selection based on action values, a form of fitness proportionate selection, to create a script of actions from the set $characterActions$. Action $a$ is selected for inclusion in the script with probability

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

, where $V(a)$ is the current value of action $a$ and $\tau$ 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.

1.1.2 Action Policy

The second 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 is not applicable. Actions are ordered first by their priority. This is generally assigned by the behavior author (the human who writes the behavior model), though there has been some work on learning action priorities [Timuri, Spronck, & van den Herik, 2007]. 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.

1.1.3 Value Updating

Action value updating is the third component of dynamic scripting. Each action is assigned an initial value and the sum of all of the action values always remains constant.

The behavior author creates a reward function that provides feedback on the utility of the script. 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 Foo’s behavior and redistributes the action values accordingly. 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. The compensation mechanism is responsible for keeping the sum of all of the action values constant. Additionally, all actions have a maximum and minimum action value, and the result of any under/overflow is divided among the other actions.

2 Extending Dynamic Scripting

We add two specific enhancements to the dynamic scripting algorithm that improve learning behavior and flexibility while imposing a minimal cost. The enhancements are: developing a flexible, stand alone, version of dynamic scripting that allows for hierarchical dynamic scripting and performing an architectural integration where this library 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.

2.1 Hierarchical Dynamic Scripting

The first enhancement is to develop a stand-alone EDS library that supports hierarchical dynamic scripting. This involves creating a Java version of the dynamic scripting algorithm and adding three specific extensions: (1) adding choice points, to support hierarchical behavior modeling, (2) adding reward points to support immediate and episodic learning and (3) creating a scaled version of the weight update algorithm to improve learning performance in certain situations.

A choice point defines a distinct learning problem, where the agent chooses a single action from some number of actions at each choice point. Choice points used here are similar to the choice points found in the Hierarchy of Abstract Machine and ALisp architectures [Andre & Russell, 2002]. The value of a choice point is that an agent can learn separately how to accomplish specific subtasks. For example, an agent may have two choice points: responseChoice and attackChoice. The first choice...
point is used to choose a subtask at the top level of the hierarchy; the second is used to select a primitive action within a subtask. The top level choice point, responseChoice, has two subtasks to choose from, Attack and Defend. These two behaviors are distinct subtasks and are identified by starting with a capital letter. Within the Attack subtask there are two primitive actions to choose from, knockdown and melee, from the attackChoice point. The agent learns the values of the Attack and Defend subtasks separately from how to actually carry out an attack.

Each choice point is identified by a unique string and contains a number of attributes: the script size, the actions available in the choice point, and the selections made in the choice point. For the dynamic scripting subset, a value of 1…n indicates the number of actions in the script with zero indicating that all actions should be in the script. This allows the choice point to easily use some or all of the available actions. Each choice point has a set of Action objects associated with it that describes the possible action choices available in this choice point. An action contains both the index of the action, its priority, and the perceived value of the action. The initial priority and value of an action is determined initially by the user, either by loading an xml file or programmatically setting initial action values. Choice points also have a list of Selection objects. This list describes all of the actions that were actually selected by this choice point. Each selection contains that action that was chosen and the user-supplied state information that defines the game state at the time the action was selected.

There are situations in modern computer games where immediate learning can have a significant impact. For example, suppose an agent is choosing between two types of melee attacks. The first is to use the attack primitive action. The second is to use the knockdown primitive action followed by the attack primitive action. After performing one or the other, 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. Continuing 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. This update function makes use of a real-valued reward that is supplied by an external Java object that implements the I_EDSAdjustor interface. Each choice point contains its own I_EDSAdjustor object created by the behavior author.

For long episodes with a small number of possible actions, it is likely that all actions will be selected in the episode. If episodic learning is used, the resulting reward will then change all of the action values in the same way and no real learning will occur. To counter this, we introduced the idea of value scaling into the update mechanism. 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, let the reward be equal to 50. Under the non-scaling weight update, if action A was completed 9 times and action B completed 1 time, they would both receive the same value update of 50. However, with value scaling, A would receive 90 and B would receive 10.

2.2 Architecture Integration

With the EDS Library implemented as described previously, hierarchical dynamic scripting is ready for integration with a hierarchical task network agent architecture. Tasks networks are a standard, and straightforward, way of representing agent behavior in games [Fu & Houlette, 2002]. 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.

We added dynamic scripting choice points and reward points to the graphical behavior network tool SimBionic. A dynamic scripting choice point node 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. A fitness proportionate selection mechanism, described previously, uses the action values to choose an action. Condition nodes in between a choice point and an action (choice) can be used to determine if that action is available or appropriate in the current game state. The behavior is then executed until a reward node is reached,
at which time the action values for the given choice point is updated using the EDS library methods.

These choice points utilize the EDS library to allow for learning anywhere within the task network. Building on the previous example a two-level hierarchy is examined, with the top level behavior shown in Figure 2.1. In this example the behavior starts at the green node and ends at the red note. The name of the choice point is “Response”, and rewards are given (i.e. values updated) only after the battle is complete. That is, either choice 1 or 2 is made the flow of control transfers to the predicate isBattleOver. If yes, a transition moves control to the reward action, otherwise control goes back to the choose action. For any encounter, many choices will be made before the battle is complete. An example reward function for “Response” is to return 1 if the battle is won or 0 if the battle is lost.

As described earlier, Attack and Defend shown in Figure 2.1 are subtasks that also may contain choice points. Figure 2.2 shows the implementation of the Attack subtask. This subtask contains a choice point, named “Attack”, that is distinct from the top level “Battle” choice point. Learning for the “Attack” problem is applied immediately after performing the primitive action attack or the sequence of primitive actions knockdown, attack. An example reward function for the attack problem is (the amount of damage caused / the number of actions performed).

The “Response” learning task learns which high-level actions are successful in an encounter while the “Attack” learning task learns how to generate as much damage as possible if the agent is attacking. The reward functions only involve game state that applies directly to the choice point, so the reward function needs to be customized for each choice point.

---

**Figure 2.1** A top level task with a choice point and an episodic reward point. The action value of choice 1 is 0.5 and action value of choice 2 is 0.1.

**Figure 2.2** The Attack subtask, which features a choice point with an immediate reward. The action value of choice 1 is 5 and the action value of choice 2 is 12.

---

### 3 Methods

The NeverWinter Night (NWN) Arena module created by Spronck et al. [2006] is used to compare the behavior of a team controlled by EDS versus a team controlled by the static library included as part of the NWN game. A separate Java application is used to connect EDS with NWN, using a MySQL database as a communication conduit since the NWN API does not allow direct external connections.

The Arena module is used to set up a series of encounters (episodes) between two identical mid-level teams, where both parties consist of a Fighter, Rogue, Mage, and Priest. Each episode is divided into a number of decision points, at which every character gets to choose a single action. An episode ends when all of the team members of one party are eliminated, at which time a new episode is started. This continues until the learning team outperforms the static team as discussed in the Results.

At the beginning of each episode, the learning team characters generate a script. The Fighter and Rogue choose 5 of their 20 actions while the Mage and Priest choose 10 of their 50 actions. For the rest of the episode, they select one action out of this script at each decision point. Based on this setup, four particular experiments were carried out: DS (Dynamic Scripting), Q (Q-Learning), HDS (Hierarchical Dynamic Scripting), and HQ (Hierarchical Q Learning).

The first experiment, 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-learner that makes use of the dynamic scripting weight update function rather than a more standard update function such as Sarsa. As in the previous experiments, all four characters share the same set of action values.

HDS makes use of the flexible nature of the EDS library by 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.

Finally, the HQ experiment adds the manually constructed hierarchy to the original Q-learning experiment.

4 Results

There are two measurements of particular interest. The first is learning efficiency, i.e. how quickly the algorithm learns to beat its opponent. The second is the variety of actions an agent takes while playing an opponent.

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. In this experiment, learning efficiency is measured by determining the average number of episodes required to beat a particular adversary, known as the turning point as defined by Spronck et al. [2006]:

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

The mean turning point for the different experiments is shown in Table 4.1. 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 using the KS-test.

<table>
<thead>
<tr>
<th>Mean Turning Point</th>
<th>n=</th>
<th>Q</th>
<th>HDS</th>
<th>HQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>24.8</td>
<td>30</td>
<td>p&lt; .05</td>
<td>X</td>
</tr>
<tr>
<td>Q</td>
<td>39.65</td>
<td>20</td>
<td>p&lt; .05</td>
<td>X</td>
</tr>
<tr>
<td>HDS</td>
<td>20.9</td>
<td>30</td>
<td>p&lt; .05</td>
<td></td>
</tr>
<tr>
<td>HQ</td>
<td>30.8</td>
<td>30</td>
<td>p&lt; .05</td>
<td></td>
</tr>
</tbody>
</table>

The graph in Figure 4.1 compares the fitness of these four algorithms during the first 19 episodes. The overall fitness value is a weighted measure of the remaining group members and their health combined with the remaining health of an individual, as defined by Spronck et al. [2006]. A single fitness value is generated each episode by averaging over the team members individual fitness values. Each data point on the 10-window graph is the average fitness of the learning team members over the last ten rounds.

Variety of actions is another important measure of interest for an online learning algorithm. A simulation or game where the opposing agent always performs the same actions will not remain of interest to the player for very long. Three different variety measures were gathered for each of these experiments. Possible diversity measures the average number of actions, from the total of 62, that are reasonably likely to be selected. This is defined as having an action value > 50. The second measure of diversity,
selection diversity, is the proportion of distinct actions that are taken by an agent. An agent that does not repeat any actions in an episode would have a score of 1.0, while an agent that repeatedly performs a single action over and over would have a score of \((1 / \text{number of actions})\). The final measure of diversity is the mean squared error of the selected actions. This measure provides information on the distribution of the repeated actions, where a lower score indicates that the agent was not just repeating a single action but instead was choosing more equally among the already completed actions.

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 4.2 compares the mean possible diversity; Table 4.3 compares the selected diversity; and Table 4.4 the mean squared error of the selected actions.

Table 4.2 Mean Possible Diversity

<table>
<thead>
<tr>
<th></th>
<th>Mean Possible Diversity (n=10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>31</td>
</tr>
<tr>
<td>Q</td>
<td>32</td>
</tr>
<tr>
<td>HDS</td>
<td>49</td>
</tr>
<tr>
<td>HQ</td>
<td>50</td>
</tr>
</tbody>
</table>

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

Table 4.3 Mean Selection Diversity

<table>
<thead>
<tr>
<th></th>
<th>Mean Selection Diversity (n=10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>0.39</td>
</tr>
<tr>
<td>Q</td>
<td>0.33</td>
</tr>
<tr>
<td>HDS</td>
<td>0.43</td>
</tr>
<tr>
<td>HQ</td>
<td>0.51</td>
</tr>
</tbody>
</table>

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

Table 4.4 Mean MSE of Selected Actions

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Mean MSE Selected Actions (n = 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS</td>
<td>6.44</td>
</tr>
<tr>
<td>Q</td>
<td>21.46</td>
</tr>
<tr>
<td>HDS</td>
<td>4.37</td>
</tr>
<tr>
<td>HQ</td>
<td>3.17</td>
</tr>
</tbody>
</table>

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

5 Related Work

With respect to dynamic scripting, the weight update function by itself has been used in research on hierarchical reinforcement learning (Ponsen, Spronck, & Tuyis, 2006), but this did not include a complete hierarchical version of dynamic scripting. Additionally, in work on controlling behavior in a real time strategy game Ponsen and Spronck (2004) divided the overall learning problem into a number of sub-problems. Our work builds on this in creating a general purpose learning architecture.

There is significant related work on game behavior architectures that include hierarchical reinforcement learning. Hierarchical reinforcement learning [see Barto & Mahadevan, 2003; Dietterich, 2000] has been previously applied to the problem of online learning for agent behavior in a number different architectures: ALisp [Andre & Russell, 2002], Icarus [Shapiro, Langley, & Shachter, 2001], and Soar [Nason & Laird, 2004]. The main difference between the HRL implemented in these three modeling architectures and the work described in this paper is the type of reinforcement learning being used. ALisp is based on the MAXQ algorithm and both Soar and Icarus use Sarsa derivatives, while the work described in this dissertation is based on the dynamic scripting algorithm. There are also significant differences in the agent architectures themselves outside of the learning algorithms.

6 Discussion and Future Work

For both measures of learning efficiency, the general ordering from best to worst performance is: HDS > DS > HQ > Q. With respect to the measures of diversity, the HQ and HDS 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 > HDS > DS > Q, though HQ and HDS are very close. These results suggest that by integrating the EDS library with SimBionic we should expect greater learning efficiency than with the other tested learning algorithms. Additionally, we expect that the EDS library will also still perform well with respect to action diversity.

Based on these results there is an ongoing investigation into using the SimBionic behavior modeling architecture, combined with EDS, to model adaptive asymmetric adversaries. This general approach is used to control behavior at two specific levels: tactic and agent. At the tactic level, the basic pieces of a training scenario are put in place before the scenario begins. This includes placing snipers, IEDs, ambush forces, etc. As the scenario
unfolds, the behavior at the agent level controls the behavior of each agent within the scenario. All of this behavior takes place within the OLIVE Platform for multi-player mission simulation.

7 References

Author Biographies

JEREMY LUDWIG is an AI Researcher at Stottler Henke Associates and a PhD candidate in the Computer Science department at the University of Oregon. His areas of research include machine learning, behavior modeling, and expert systems.

ARTHUR FARLEY is a Professor of Computer and Information Science with interests in artificial intelligence and communication networks.