Examining Extended Dynamic Scripting in a Tactical Game Framework

Jeremy Ludwig[1] **and Arthur Farley**[2]

[1]Stottler Henke Associates
San Mateo, CA 94404
ludwig@stottlerhenke.com

[2]University of Oregon
Eugene, OR 97403
art@cs.uoregon.edu

Abstract

Dynamic scripting is a reinforcement learning algorithm designed specifically to learn appropriate tactics for an agent in a modern computer game, such as Neverwinter Nights. This reinforcement learning algorithm has previously been extended to support the automatic construction of new abstract states to improve its context sensitivity and integrated with a graphical behavior modeling architecture to allow for hierarchical dynamic scripting and task decomposition. In this paper, we describe a tactical abstract game representation language that was designed specifically to make it easier to define abstract games that include the large amount of uncertainty found in modern computer games. We then use this framework to examine the effectiveness of the extended version of the dynamic scripting algorithm, using Q-learning and the original dynamic scripting algorithms as benchmarks. Results and discussion are provided for three different abstract games: one based on combat in role-playing games and two based on different aspects of real-time strategy games.

Introduction

Dynamic scripting (DS) [Spronck et al., 2006] is one example of an online reinforcement learning algorithm developed specifically to control the behavior of adversaries in modern computer games and simulations as defined by Laird and van Lent [2001]. The DS algorithm was designed especially to support efficient learning based on a limited amount of experience and to maintain a diversity of selected behaviors, both of which are significant requirements for online learning in modern computer games [Spronck et al., 2006]. The DS algorithm has been tested in both role playing games (Nevewinter Nights) and real-time strategy games (Wargus), with promising results. That is, agents using DS quickly learn how to beat their opponent.

This reinforcement learning algorithm has previously been extended to support the automatic construction of new abstract states to improve its context sensitivity and integrated with a graphical behavior modeling architecture to allow for hierarchical dynamic scripting and task decomposition [Ludwig & Farley, 2007, 2008]. The aim of these extensions was to improve the learning performance, applicability, and flexibility of dynamic scripting-based learning in games and simulations. Learning performance is measured in terms of the relative score achieved by the agent and how quickly 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. While the results presented by Ludwig and Farley demonstrated improved learning performance in an abstract predator/prey game [2007] and improved learning performance in the role playing game NeverWinter Nights [2008], the remaining two claims were not addressed.

In this paper we describe a tactical abstract game framework used to further evaluate the extended version of the dynamic scripting algorithm. The results generated by the extended algorithm (EDS) are compared to results from the standard dynamic scripting algorithm in three different abstract games. The general framework, and the three specific games created with it, have all been designed to support efficient examination of all three criteria: improved learning performance, applicability, and flexibility.

The remaining introduction presents an overview of the DS algorithm and a description of the extended version of DS (EDS) built into a behavior modeling architecture. The tactical abstract framework is described in the next section. The third section contains the methods, games, and results for each of three experiments. The final section of this paper offer discussion of these results and concluding remarks.

Overview of Dynamic Scripting

This section outlines the more standard Q-learning [Sutton & Barto, 1998] followed by DS [Spronck et al., 2006] as a way to highlight the main elements of the DS algorithm. In both cases, an agent makes use of the learning algorithm to determine what action to take given a perceived game state. Note that DS makes relatively little use of game state information, which makes it more useful for higher-level, tactical decisions and less useful for decisions that rely highly on current context such as movement through a grid-world.

Q-Learning:

- Actions have a value Q(s,a), where the set of states, S, can contain actual or abstract game states. Abstract game states are collections of game states that are treated as a single state.

- Actions are selected during an episode with value proportionate selection, based on their Q values.
- When a reward is given, selected actions are updated using a Q-learning update function combined with a domain-specific reward function created by the behavior author.

Dynamic Scripting:

- Actions have (i) a value Q(s, a), where the set of states, S, contains only a single abstract state; (ii) an optional IF clause that describes when an action can be applied based on the perceived game state; and (iii) a user-defined priority (or learned, see [Timuri et al., 2007]) that captures domain knowledge about the relative importance of this action.
- Action values are used to create scripts of length *n* prior to an episode by selecting actions in a value-proportionate manner (softmax) from the complete set of actions available to the agent.
- During an episode, *applicable* actions are selected from the script in priority order first, the action value second. *Applicability* is determined by the perceived game state and the actions IF clause.
- At the end of an episode, action values are updated using the dynamic scripting updating function combined with a domain-specific reward function created by the behavior author. In contrast to Q-learning, each action has its value updated, not just the selected action(s). The reward is distributed primarily to the actions selected in the episode and then to actions in the script that were not selected, with a smaller negative reward given to actions not included in the script.

Extending Dynamic Scripting

While the dynamic scripting algorithm has shown significant promise in controlling agent behavior in modern computer games, there were a number of issues that were previously addressed in an extended version of dynamic scripting [Ludwig & Farley, 2007, 2008]. In their work, the authors create an extended version of dynamic scripting that is integrated with a graphical behavior modeling architecture to allow for hierarchical dynamic scripting and task decomposition and that supported the automatic construction of new abstract states to improve its context sensitivity.

The end result of this effort is a graphical behavior modeling tool [Fu & Houlette, 2002] that supports the use of dynamic-scripting based choice points. An example behavior is shown in Figure 1. This figure consists of actions (rectangles), conditions (ovals), and ordered, directed connectors. Within a graph (called a behavior), the actions and conditions can reference other behaviors (bold rectangles) to form hierarchical behaviors. Control flows from the top shaded node to the bottom shaded node.

The dynamic-scripting based choice point in this particular behavior is indicated by the *choose*(*choicePointName*) action. Choice points, as used in extended dynamic scripting, are based on the choice points found in the Hierarchy of Abstract Machine and ALisp architectures [Andre & Russell, 2002]. Each choice point has a corresponding reward point indicated by the *reward*(*choicePointName*) action. This reward may be immediate or episodic as determined by the behavior author in placing the reward point.

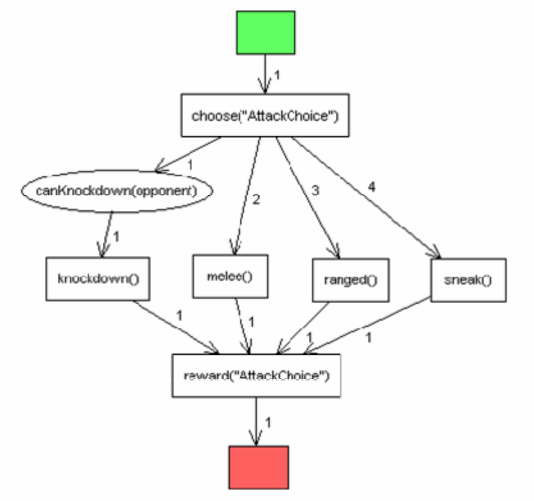

Figure 1. Example behavior with dynamic scripting-based choice point.

Table 1 illustrates the dynamic scripting specific data associated with this particular choice point. Control enters the top node and flows to the choice point. The choice point then generates a script of two actions from the four available actions based on the action values; one possibility is indicated by asterisks. When executing the script, If the condition *canKnockdown(opponent)* is true, action 1, with higher priority, will be selected; otherwise, action 2 will be selected. When the reward point is reached, the values associated with the actions will be updated and the script emptied.

Table 1. Action priority, value, and script selection data for the "AttackChoice" choice point.

Action	Priority	Value	Script
1	High	112	*
2	Low	88	*
3	Low	50	
4	Med	117	

The addition of choice and reward points to a hierarchical modeling architecture allows the behavior author to perform both task decomposition and manual state abstraction with dynamic scripting-based choice points. An example of manual state abstraction is deciding to have one script (choice point) for when a Fighter is present on the opposing team and another for when not present.

In adding DS-based choice points to a behavior modeling architecture, the authors build on and generalize previous research on hierarchical dynamic scripting [Dahlbom & Niklasson, 2006; Ponsen et al., 2006] and dynamic scripting with manual state abstraction [Ponsen & Spronck, 2004]. Additionally, there is significant related work on game behavior architectures that include hierarchical reinforcement learning such as ALisp [Andre & Russell, 2002] , Icarus [Shapiro et al., 2001], and Soar [Nason & Laird, 2004] to name a few. The main difference between the hierarchic learning implemented in these three architectures and the work described in this paper is the reinforcement learning algorithm being used.

The automatic state abstraction component of EDS uses data-mining techniques to automatically create classification trees that identify the most relevant variables in the perceived game state based on historic state and reward data. Once these trees are created, each leaf node in the tree is given its own set of dynamic scripting data (Table 1). The main idea behind this feature is to identify states where the current script appears to be working and states where the script is not working. Following this, separate scripts are created for the different state sets. This is known in machine learning as 'boosting', and in fact boosting algorithms such as Decision Stump [Witten & Frank, 2005] have proved effective when used in EDS [Ludwig, 2008]. This is the default used for automatic state abstraction in EDS. The tree-based automatic state abstraction utilized by extended dynamic scripting is based on previous research that showed promising results in the context of more standard reinforcement learning algorithms (e.g., see G-Algorithm [Chapman & Kaelbling, 1991] and U-Tree [McCallum, 1996]).

When a decision is made in a choice point with automatic state abstraction enabled, the correct set of dynamic scripting data is retrieved based on the current perceived game state. To continue our example, if the behavior author does not know that different scripts should be used depending on whether a Fighter is present on the other team, automatic state abstraction could learn that this is an important distinction and create distinct scripts for the two cases. This is an example of a state abstraction actually found by EDS when tested in Neverwinter Nights [Ludwig, 2008]. EDS with state abstraction also successfully improved performance of a character in a predator/prey abstract game [Ludwig & Farley, 2007]

Tactical Abstract Game Framework

In order to better evaluate the claims of EDS, we created an abstract game framework based upon high-level (tactical) decisions in games and simulations. 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 aspects commonly encountered in modern computer games. With these additions the TAG framework can be used to model some instances of the class of decision problems referred to as Markov Decision Processes (MDPs). An MDP is defined by the tuple $<S, A, T, R>$ [Kaelbling et al., 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 performed in state s, and R is the reward function, where $R(s, a)$ is the expected reward 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 states, the transition function, and the reward function. To be clear, TAG is an implemented framework that includes (i) an XML-based game specification language, (ii) a Java-based player specification, and (iii) a general computer program that instantiates a game based on the specifications, uses the given player to make decisions in the game, and reports the results generated by the player. 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. Note that in order to take up less space, we are using a formal representation rather than the actual XML representation.

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.

The second basic component of a game definition is the set of actions, A. Each action is defined by a set of parameter value tuples $< O$, 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. TAG is only capable of producing a reward function, *R(s,a)*, that generates a random reward distribution within the given bounds.

- **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, defined the same as r+ but used 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.* The applicability value is a significant simplification that determines the applicability of an action randomly without the need to specify information on the actual or observed game state.

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.

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.

TAG Player

A TAG Player is responsible for making decisions in a TAG Game, where each player implements a different action selection method. The player implementations examined in this paper are Q, DS, and EDS. The Q player implements the Q-learning algorithm as described by Sutton and Barto [1998]. The Q-player provides a baseline for informational purposes only – the performance of a standard reinforcement learning algorithm on the same problem. There are a number of standard ways the Q-learning algorithm could have be extended to improve performance that are not investigated, though it does take advantage of manually constructed abstract states when available. The DS player implements the standard dynamic scripting algorithm, without manual or automatic state abstraction and without task decomposition. The EDS player makes full use of the behavior architecture and choice points to support all three of these.

TAG Experiment

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*. Rewards can be delayed to require multiple actions per reward (an episodic reward).

Experiments

We define three distinct games in the TAG framework, with a number of different players for each game: Anwn, Resource Gathering, and Get the Ogre. The first is an abstract role-playing game, based in part on the NeverWinter Nights computer game. The Resource Gathering game builds on a real-time strategy subtask studied by Mehta et al. [2008]. Get the Ogre is derived from another real-time strategy game subtask, previously explored with ALisp [Marthi et al., 2005]. These three games represent a range of different problems encountered in modern computer games.

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, EDS and DS should perform very well. The other two games require a sequence of actions to complete the game. It is predicted that DS will perform poorly on the Resource Gathering and Get the Ogre games without the additional domain knowledge that can be encoded in EDS. For each of these games, we present a description of the game followed by the main results and discussion of their significance.

Anwn

Anwn is an abstract version of the combat portion of a role playing game. The tuple <F, A, R> is defined as:

- F: 10 features (Boolean and Integer)
- A: 40 actions - 10 good (low applicability, high reward); 20 medium (moderate applicability and reward); 10 poor (high applicability, low reward); Current observation state has little affect on rewards
- R: Randomly change 1 observation feature after each action

For the DS and EDS players, the priority assigned matches the rating of the actions (i.e. good = high priority). In all cases, the score is the average reward received in the episode. Rewards are immediate after each selection.

The results of this experiment are shown in Figure 2. Manual state abstraction indicates the construction of three abstract game states based on domain knowledge. The EDS and Q players use this to learn which actions perform best in each of three abstract game states. EDS with automatic state abstraction comes in second, performing nearly as well as the manually constructed state abstraction. With automatic state abstraction, the EDS player is learning to create distinct scripts for different sets of game states. The DS learner learns a single script that is used in all game states. This game demonstrates the utility of both manual and automatic state abstraction in EDS.

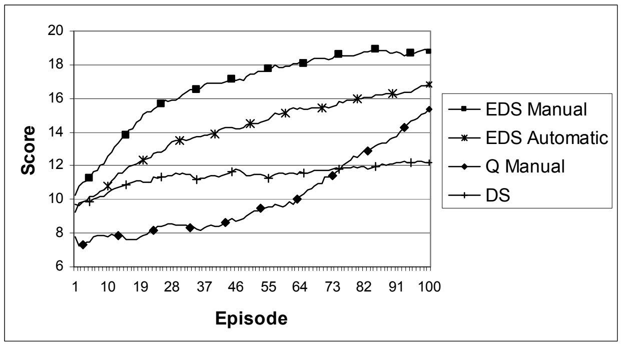

Figure 2. Performance of learners in the Anwn game. Higher scores indicate better performance (average reward received in an episode) (n=1000).

Resource Gathering

In the Resource Gathering game, the goal is to send out a peasant to collect 100 wood and 100 gold by making one trip to the forest and one trip to the gold mine. However, the locations of the gold, wood, and town hall (where resources are to be returned) are not known ahead of time.

- F: 10 features (Boolean and Integer)
- A: 13 actions - Move to 1 of 9 possible line-of-sight locations (9 distinct actions); Mine gold (if gold mine visible); Chop wood (if forest visible); Drop off wood (if carrying wood and town hall visible); Drop off gold (if carrying gold and town hall visible)
- R: Changes the observation states in predictable ways, such as moving the peasant to the location, gathering a resource into a peasants arm, and dropping off the resources to increase the amount of stored wood or gold.

The DS player assigns the highest priority to the drop off actions, medium priority to gathering actions, and lowest priority to moving actions. For all players, the reward given is -1 * the number of actions taken in the episode.

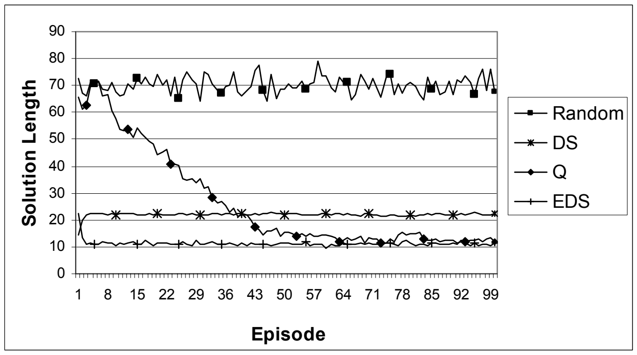

Figure 3. Results of the resource gathering experiment. Lower scores indicate better performance (shorter solution) (n=1000).

The results in show that while the Q player (no state abstraction) starts out performing the same as random, it starts to approach the optimal solution of eight actions relatively quickly. The DS player is able to perform better than random by making use of action priorities. For example, if the agent is over a forest and is not carrying anything it will chop wood (assuming that action is in the script) rather than move to a new location. However, the DS player demonstrates no learning since the script cannot take the context of the move actions into account. That is, if a peasant is holding gold it should be sent somewhere different than when it should be gathering gold.

The EDS player makes use of task decomposition, supplied by the behavior modeling architecture to break the problem into distinct learning subtasks as shown in the partial plan in Figure 4. In this figure, the *FindForest* and *FindTownHall* sub-behaviors (indicated by bold rectangles) contain distinct choice points that are quickly able to learn how to solve the subtask while the *ChopWood* and *DropOffWood* actions capture known action ordering that was specified as priorities in DS. This game demonstrates the utility of task decomposition as supported by EDS.

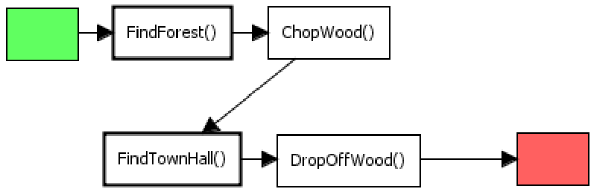

Figure 4. Task decomposition solution to Resource Gathering expressed with EDS.

Get the Ogre

In Get the Ogre, the goal is to perform a sequence of actions that creates a small squad of soldiers and attacks a nearby Ogre. The actions available to the player consist of building a farm, creating wood or gold gathering peasants, creating a solider, and attacking the Ogre with 3, 4, or 5 soldiers.

- F: 4 features (Integer)
- A: 13 actions - Build farm (if has wood); Gather wood-gathering peasant (if has food); Gather gold-gathering peasant (if has food); Create solider (if has wood and gold); Attack ogre (if has 3, 4, or 5 soldiers)
- R: Changes the observation states in predictable ways, such as spending resources, gathering resources, or attempting an attack on the ogre which is more likely to succeed with more soldiers.

The optimal sequence of actions includes creating two gold peasants, one wood peasant, four soldiers, and then attacking the Ogre. The DS / EDS assigned priorities to the actions as follows (from high to low): attack, soldier, gold, wood, farm. One difference with this game is that it is easy for the player to get into a state where no actions are applicable or to get into loops and not solve the problem within a limit of 50 actions. The episodic reward is based on defeating the Ogre with the least number of soldiers (4 is optimal), or a large negative reward if not solved.

The DS player was not able to solve this problem since there exists no single script that can perform this task without getting into an infinite loop or reaching a state where no action was available. The EDS player with automatic state abstraction is able to find a reasonable set of abstract states after a minimum of five episodes and is outperforming the Q player by episode 25. This game demonstrates the ability of EDS to solve games that could not be solved by dynamic scripting without the inclusion of additional domain knowledge.

Conclusion

In this paper, we developed an abstract tactical game framework and three abstract games based on particular aspects of modern computer games. Each game was tested using a number of different learning algorithms in order to examine the capabilities of the extended dynamic scripting algorithm.

Taken together, the results from the three abstract games provide evidence for the hypothesis that EDS improves upon the basic dynamic scripting algorithm. EDS demonstrated increased learning performance, shown through faster learning and improved scores in all three games. EDS also demonstrated 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, EDS demonstrated learning in games where dynamic scripting alone could not as shown in the Resource Gathering and Get the Ogre games.

These extensions are especially important for dynamic scripting as the DS algorithm is designed to learn very quickly when applied in modern computer games, in part by ignoring game state information. The extended dynamic scripting algorithm allows the behavior modeler to start with the speed of learning and diversity of behavior supplied by dynamic scripting and improves upon it by allowing for a number of ways in which to add (or learn) domain knowledge that can be used to improve learning performance.

References

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

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.

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.

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

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

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

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

Ludwig, J. (2008). *Extending Dynamic Scripting.* Department of Computer and Information Science, University of Oregon, Ann Arbor: ProQuest/UMI.

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., & Farley, A. (2008). *Using hierarchical dynamic scripting to create adaptive adversaries.* Paper presented at the 2008 Conference on Behavior Representation in Modeling and Simulation, Providence, RI.

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.

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.

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., & 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.

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.

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.

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.

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