A Learning Infrastructure for Improving Agent Performance and Game Balance

Jeremy Ludwig and Art Farley

Computer Science Department, University of Oregon
120 Deschutes Hall, 1202 University of Oregon
Eugene, OR 97403
ludwig@cs.uoregon.edu, art@cs.uoregon.edu

Abstract

This paper describes a number of extensions to the dynamic scripting reinforcement learning algorithm which was designed for modern computer games. These enhancements include integration with an AI tool and automatic state construction. A subset of a real-time strategy game is used to demonstrate the learning algorithm both improving the performance of agents in the game and acting as a game balancing mechanism.

Introduction

One of the primary uses of artificial intelligence algorithms in modern games [Laird & van Lent, 2001] is to control the behavior of simulated entities (agents) within a game. Agent behavior is developed using a diverse set of architectures that run from scripted behaviors to task networks to behavior planning systems. Once created, agents can continue to adapt by using on-line learning (OLL) algorithms to change their behavior [Yannakakis & Hallam, 2005]. There are two related goals that OLL can help acheive. The first is to alter behaviors such that the computer can outplay the human, creating the highest-performing behavior [Aha, Molineaux, & Ponsen, 2005]. The second is game balancing, to adapt the behavior of an agent, or team of agents, to the level of play of the human. The goal of game balancing is to provide a level of play that is a challenge to the player but still gives her a chance to win [Andrade, Ramalho, Santana, & Corruble, 2005].

Dynamic scripting (DS) is a reinforcement learning algorithm that has been designed to quickly adapt agent behavior in modern computer games [Spronck, Ponsen, Sprinkhuizen-Kuyper, & Postma, 2006]. It differs from reinforcement learning algorithms such as q-learning in that it learns the value of an action in a game, not the value of an action in a particular state of a game. This means that DS is a form of an action-value method as defined by Sutton and Barto [1998]. The DS learning algorithm has been tested in both the role playing game Nevewinter Nights and the real-time strategy game Stratagus [Spronck et al., 2006]. The Neverwinter Nights work also includes difficulty scaling mechanisms that works in conjunction with dynamic scripting to achieve game balancing. The authors have examined hierarchical reinforcement learning (HRL) in games using a dynamic scripting-based reward function [Ponsen, Spronck, & Tuyls, 2006] and compared its performance to a q-learning algorithm. The DS algorithm has also been extended to work with case-based selection mechanism [Aha et al., 2005]. Dynamic scripting has also been extended with a hierarchical goal structure intended for real-time strategy games [Dahlbom & Niklasson, 2006].

This paper describes a set of dynamic scripting enhancements that are used to create agent behavior and perform automatic game balancing. The first extension is integrating DS with a hierarchical task network framework and the second involves extending dynamic scripting with automatic state construction to improve learning performance. The rest of the introduction will describe these extensions in more detail. The Experiments and Results section describes two separate instances of applying this infrastructure to a sub-problem of a real-time strategy game. The focus of the first experiment is developing learning agents to play the game; the focus of the second is the performance of the game balancing behavior.

Dynamic Scripting Enhancements

A Java version of a dynamic scripting library was constructed based on the work described by Spronck, et al. [2006]. This library was implemented with the ability to learn both after an episode is completed (existing behavior) and immediately after an action is completed (new behavior). The library also contains methods for specifying the reward function as a separate object, so that a user only needs to write a reward object that implements a particular interface in order for the library to be used on a different game. This library differs from the original algorithm in two main ways. The original DS algorithm selects a subset of available actions for each episode based on the value associated with each action. During the game, actions are

then selected from this subset by determining the highest-priority action applicable in the current game state. The enhanced DS library chooses an action from the complete set of actions based on its associated value. This makes it possible for agent behavior to change during an episode and demonstrate immediate learning.

Despite this change, the weight adjustment algorithm was not changed. That is, when a reward is applied the value for action *a* is updated: V(a) = V(a) + reward. This applies for all actions selected in the episode (which may be only one). All unselected action receive compensation: V(a) = V(a) + compensation, where compensation = - ((#selectedActions/#unselectedActions) * reward). The SoftMax action selection mechanism also remained unchanged and had a fixed temperature of 5 for all experiments.

The DS library was then integrated with an existing task network behavior modeling tool [Fu, Houlette, & Ludwig, 2007]. This tool allows users to graphically draw flowchart-like diagrams to specify connected sequences of perceptions and actions to describe agent behavior (e.g., see Figure 2). *Choose* nodes were added to the tool to indicate that the DS library should be used to choose between the actions connected to the node. *Reward* nodes were also added to indicate the points at which rewards should be applied for a specific *choose* node. Each *choose* node learns separately, based only on the actions it selects and the rewards it receives, allowing for multiple choice and reward nodes in the agent behavior.

The behavior modeling tool supports hierarchical dynamic scripting by including additional choice points in sub-behaviors. It also supports a combination of immediate and episodic learning. For example, a top-level behavior can choose to perform a melee attack (and be rewarded only after the episode is over) and the melee attack sub-behavior can choose to attempt to trip the enemy (and be rewarded immediately). The resulting system is somewhat similar to that described by Marthi, Russell, and Latham [2005], especially the idea of "choice points" though the type of reinforcement learning is quite different.

The enhanced version of dynamic scripting also makes use of automatic state construction to improve learning. The standard DS algorithm considers only a single game state when learning action values. However, it is easy to see that at some points game state information would be useful. For example, in Neverwinter Nights, area-effect spells are more effective when facing multiple weaker enemies than when facing a single powerful enemy. So when deciding what spell to cast, having one set of weights (action values) for <= 1 enemy and another set of action values for >1 enemy could improve action value learning. The difficulty lies in extending the algorithm such that agent behavior is automatically improved while maintaining the efficiency and adaptability of the original algorithm.

To achieve automatic state construction, each *choose* node was designed to contain a variable number of policies and a classifier that partitions the game state into one of the available policies. The DS library was integrated with the Weka data mining system [Witten & Frank, 2005] to perform automatic construction of game state classifiers. Based on previous work in automatic state construction in reinforcement learning [Au & Maire, 2004], the Decision Stump algorithm was selected as an initial data mining algorithm. It examines the game state feature vector, the action taken, and the reward received to classify the game state into one of two policies. While this does move DS closer to standard reinforcement learning, by limiting the number of states in a complex game to a relatively small number (2-5) it is expected that the generated behavior can be improved with a negligible impact on the speed of the algorithm both in terms of number of games required and computational efficiency.

Experiments and Results

Two games, based on a real-time strategy sub-problem, are used to demonstrate the dynamic scripting infrastructure. The first game examines the ability of the infrastructure 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.

Experiment One: Agent Behavior

Figure 1 Worker To Goal Game

The *Worker To Goal* 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 paper uses dynamic scripting to make higher level decisions. The simple game shown in Figure 1 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.

The flat behavior of the worker uses a *choose* node 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 DS 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.

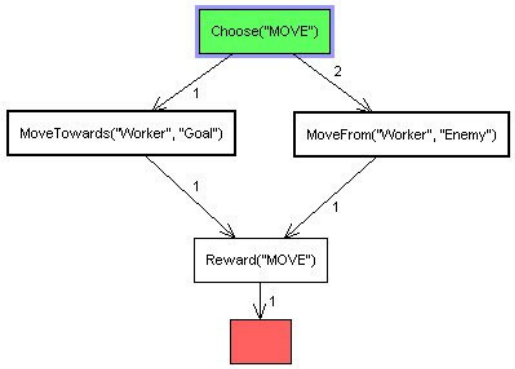

Figure 2 Worker Behavior

The behavior for this worker is shown in Figure 2. 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.

The hierarchical version of the worker behavior, H_Worker, replaces the primitive action MoveTowards with a sub-behavior and introduces another choice point in the new MoveTowards sub-behavior as seen in Figure 3. 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.

The Worker and H_Worker behaviors were each used to control the worker agent in 200 games with the described choice points, where all of the agents were positioned randomly at the start of each game and the weights 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.

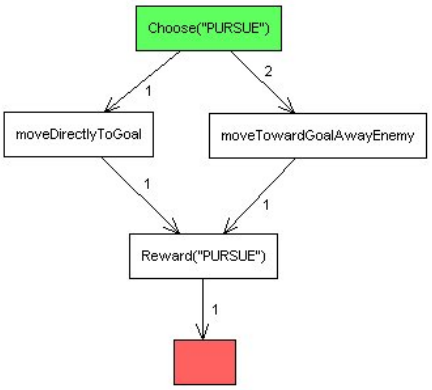

Figure 3 H_Worker ToGoal Behavior

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]).

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 et al., 2006]: relative distance to goal, direction to goal, distance to enemy, direction to enemy, and the received reward.

The Decision Stump algorithm was used after 1, 2, 5, 10, 20, and 100 games to partition the game state with varying amounts of data. 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 weights 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.

Experiment Two: Game Balancing Behavior

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 To Goal game shown in Figure 4. These two behaviors were chosen as the low and high performers of the previous experiment. For both workers, automatic state construction is turned off.

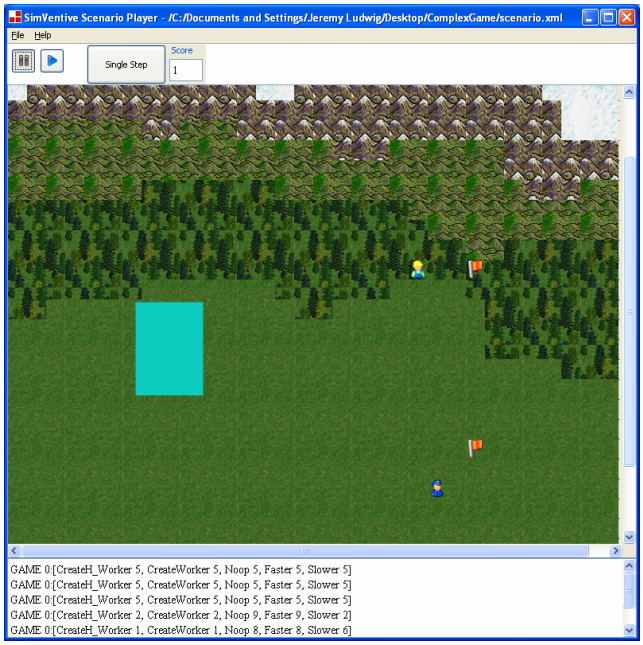

Figure 4 Worker To Goal 2

In Worker To Goal 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 demonstrates different levels of 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 computer scores one point. Every time the soldier captures a worker, the player 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.

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 5.

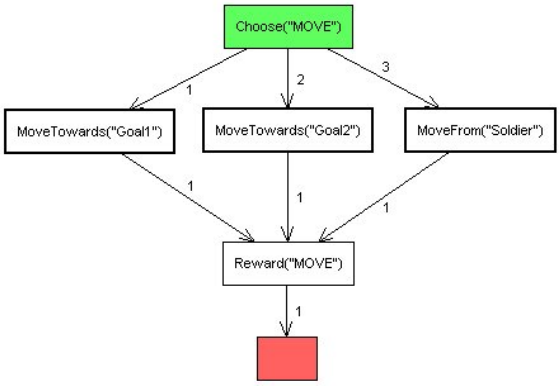

Figure 5 Worker 2 Behavior

Figure 6 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.

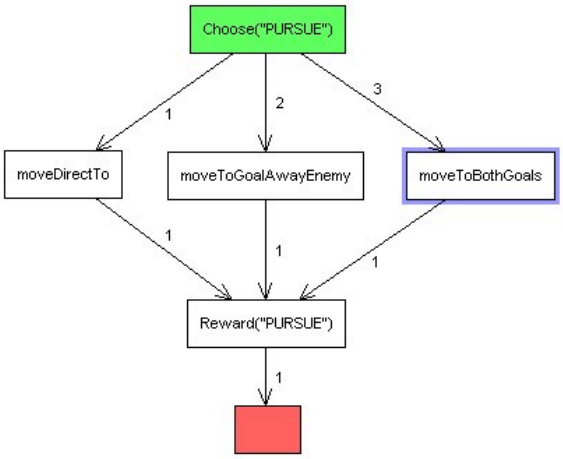

Figure 6 H_Worker 2 ToGoal Behavior

The game balancing behavior shown in Figure 7 attempts to keep the score as close to zero as possible by performing a number of possible actions each episode (30 game ticks). Keeping the score close to zero balances the number of workers that make it to the goal and the number of workers captured. Admittedly, this is an arbitrary function where the main purpose is to demonstrate the learning capabilities of the system and the actual function learned is of secondary importance.

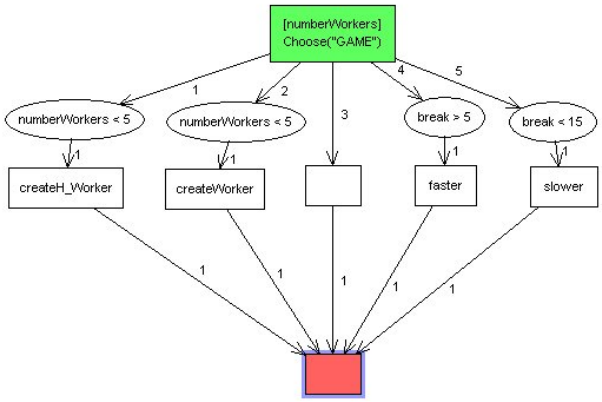

Figure 7 Game Balancing Behavior

The available actions are creating a Worker, creating an H_Worker, doing nothing (noop), speeding up (performing up to five actions/episode) and slowing down (performing down to one action/episode). This meta-level behavior was created using the same choice point infrastructure used in the agent behaviors.

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 |s| - |s'|, where s is the score at the beginning of the episode and s' is the score at the end. This rewards actions that bring the score closer to zero. Unlike the worker agents which are rewarded immediately, this behavior only receives a reward at the end of an episode.

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. For comparison, two other reward functions were also tested. The first doubles the reward, which causes them to have a bigger impact. The second halves the reward so the learning impact of a reward is halved. To provide an upper level bound on behavior, a fourth algorithm was created where the GAME choice point was replaced by a random decision.

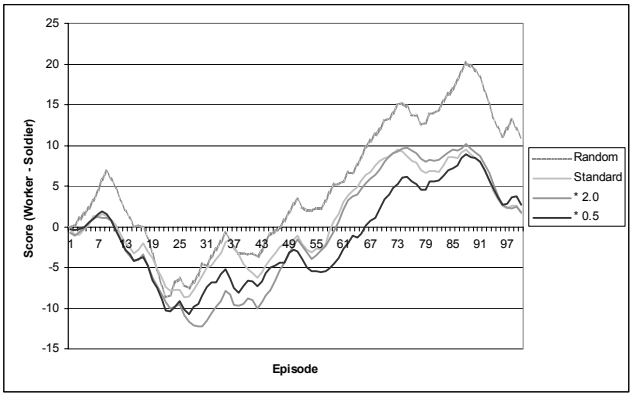

Figure 8 Game Balance Results

The average score (#worker goals - #workers captured) after each episode for the four different cases is presented visually in Figure 8. This graph also indicates the position of the soldier at various times during the game (100 episodes). 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.

Table 1 Mean Absolute Deviation

Random	12.0
Standard Reward	8.3
Double Reward	9.8
Half Reward	6.9

To capture a quantitative measurement, the mean absolute deviation (MAD) [Schunn, 2001] was also computed

across the eight games for each type and is shown in Table 1. The standard, double, and half reward cases are all statistically significant improvements ($p < .01$).

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 lowering the score by having more workers captured). Another interesting finding is that halving the reward (slowing down learning) resulted in the lowest MAD. Since actions are spread out evenly over an episode, a worker created at the end of an episode may not reach the goal or get captured until the next episode. Ignoring half of the reward each episode effectively takes this into account, while demonstrating the sensitivity of RL algorithms to different reward functions.

The game balancing performance -- while an improvement over a random controller -- was not as good as expected. We are still investigating ways to improve the game balancing behavior and the effects of applying automatic state construction techniques to this choice point.

Conclusion and Future Work

This paper demonstrates a single learning mechanism capable of both learning how to balance the game and how to play the game, representing different game aspects as a single type of learning problem. While promising, the two experiments discussed are only an initial test of the dynamic scripting infrastructure. The results show that automatic state construction can be used to improve agent behavior at a minimal cost, but more research is required to determine how to make this a generally useful feature. Additionally, while the game balancing behavior works to some extent, there is a lot of room for improvement. We are currently investigating applying automatic state construction and introducing a hierarchy of game balancing behaviors to improve performance.

It remains to be seen if these extensions can improve upon the behavior of the original DS algorithm in a large scale modern computer game. Our future work will focus on such an integration that will demonstrate the effectiveness and efficiency of the infrastructure as a whole. This will allow the enhanced DS algorithm to be compared to both the original DS algorithm as well as other standard RL algorithms such as q-learning.

References

Aha, D. W., Molineaux, M., & Ponsen, M. (2005). *Learning to win: Case-based plan selection in a real-time strategy game.* Paper presented at the Sixth International Conference on Case-Based Reasoning, Chicago, IL.

Andrade, G., Ramalho, G., Santana, H., & Corruble, V. (2005). *Extending reinforcement learning to provide dynamic game balancing.* Paper presented at the Reasoning, Representation, and Learning in Computer Games: Proceedings of the IJCAI workshop, Edinburgh, Scotland.

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.

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., & Ludwig, J. (2007). *An AI Modeling Tool for Designers and Developers*. Paper presented at the IEEE Aerospace Conference.

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

Marthi, B., Russell, S., & Latham, D. (2005). *Writing stratagus-playing agents in concurrent ALisp.* Paper presented at the Reasoning, representation, and learning in computer games: Proceedings of the IJCAI workshop, Edinburgh, Scotland.

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

Schunn, C. D. (2001). Goodness of Fit Metrics in Comparing Models to Data. Retrieved 06/06, 2004, from http://www.lrdc.pitt.edu/schunn/gof/index.html

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.

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

Yannakakis, G. N., & Hallam, J. (2005). *A scheme for creating digital entertainment with substance.* Paper presented at the Reasoning, representation, and learning in computer games: Proceedings of the IJCAI workshop, Edinburgh, Scotland.