RETALIATE: Learning Winning Policies in First-Person Shooter Games

Megan Smith, Stephen Lee-Urban, Héctor Muñoz-Avila

Department of Computer Science & Engineering, Lehigh University, Bethlehem, PA 18015-3084 USA

Abstract

In this paper we present RETALIATE, an online reinforcement learning algorithm for developing winning policies in team first-person shooter games. RETALIATE has three crucial characteristics: (1) individual BOT behavior is fixed although not known in advance, therefore individual BOTS work as "plug-ins", (2) RETALIATE models the problem of learning team tactics through a simple state formulation, (3) discount rates commonly used in Q-learning are not used. As a result of these characteristics, the application of the Q-learning algorithm results in the rapid exploration towards a winning policy against an opponent team. In our empirical evaluation we demonstrate that RETALIATE adapts well when the environment changes.[1]

Introduction

Reinforcement learning (RL) has been successfully applied to multiagent systems in many instances such as game theory (Bowling & Veloso, 2002), and RoboCup soccer (Matsubara *et al*., 1996; Salustowicz *et al*., 1998). However, RL has seen little use in digital games aside from static, turn-based, fully observable games such as Backgammon. On the other hand, off-line techniques such as decision tree learning have been successfully applied in digital games (Fu & Houlette, 2003). This lack of application of RL in complex digital games is surprising, given that game practitioners have pointed out the positive impact that on-line learning could have in the creation of more enjoyable games (Rabin, 2003). Commercial games currently use predefined difficulty levels, which are recognized as very limited. As an alternative approach, RL could be used to dynamically adjust the difficulty level of AI opponents (Spronk *et al*., 2004).

We conjecture that part of the reason why RL has not been broadly used for computer games is its well-known low convergence speed towards an optimal policy. The task is particularly daunting if we consider team-based first-person shooter (FPS) games. These are very popular kinds of games where teams consisting of two or more players compete to achieve some objectives. In team-based FPS games, individual players must have good reflexes and be able to make decisions on the fly. Also, players must work together to achieve the winning conditions of the game. The game is dynamic in the traditional sense of AI: world state changes occur while players deliberate about what to do. Moreover, FPS games are well-known for changes in the world state occurring rapidly. Frequently, FPS games are zero-sum games, where one team's gains are the other team's loses.

As a result of these complexities in team-based FPS games, a RL algorithm used in this domain needs to explore both individual player behavior and team behavior as well as possible interactions between the two. Formulating a winning policy depends on the environment which includes our own players, the opponent team (e.g., tactics), or the world state (e.g., the map where the game is played). The combination of these factors makes the problem of obtaining adequate team strategies even more challenging. A winning policy against one opponent might fail on another. Furthermore, an opponent's team may adjust its tactics to counter a winning policy.

In this paper we present RETALIATE (for: Reinforced Tactic Learning in Agent-Team Environments), an online reinforcement learning algorithm for team-based FPS games. RETALIATE has the following characteristics:

- In RETALIATE the behavior of the individual team members is fixed, although this behavior is not explicitly known. This means individual players work as "plug-ins" that can be easily exchanged.
- RETALIATE concentrates on coordinating the team rather than controlling individual player's reactive behavior. Specifically the problem model describes game map locations of interest. This serves to reduce the possible state space that RETALIATE needs to explore, resulting in a faster formulation of a policy.
- RETALIATE eliminates discount rates commonly used in Q-learning (i.e., discount rates that lead to convergence). If the environment (e.g., the opponent, the map) changes, then the algorithm can quickly adapt to these changes. As a consequence RETALIATE may not converge, or if it does converge, the resulting policy might not be optimal. We argue that optimality is not needed and rather speed in formulating a winning policy is desired.

As a result of these characteristics, RETALIATE performs rapid exploration towards a winning policy in our problem model against an opponent team.

We demonstrated RETALIATE's speed in learning a winning policy using the Unreal Tournament™ (UT) game engine, published by Epic Games Inc. Against a static opponent, RETALIATE proves capable of developing a winning policy within the first game most of the time and always within two games. We also conducted an

experiment showing that RETALIATE can adapt to a changing environment.

The paper continues as follows. We discuss related work in the next section, including a brief overview of reinforcement learning. Next, we discuss domination games in Unreal Tournament™ (UT), which is the kind of team-based FPS game we used in our experiments. Following that comes the main section of this paper, which explains our model of the learning problem. Then we discuss our empirical evaluation. We conclude this paper with final remarks.

Related Work

Computer games are considered by some to be the "killer app" for human-level AI, and coordinating behavior is one of many research problems they present (Laird et al., 2001). FPS games are particularly interesting because they present real-time requirements and provide a rich, complex environment. Another advantage is that, unlike the RoboCup domain where an agent's sensors can fail or provide inaccurate information, FPS games have perfect sensors that never fail. Also FPS games can avoid the need to consider team member communication, as is necessary in work described by Sen *et al*. (1994) and Lauer & Riedmiller (2000). This reduced complexity allows experiments to focus on coordination issues in a way not complicated by noise or communication protocols. Outside of game environments, controlling team agents is an important research topic (e.g., (Nair & Tambe, 2005)).

One example of coordinating bots was by Hoang et al. (2005). In that study, hierarchical task networks (HTNs) were used to encode strategic behavior that created cohesive behavior in a team of bots. The results demonstrated a clear dominance by the HTN team. In spite of the success, the strategy was encoded *a priori* specifically for the scenarios. No machine learning was involved.

Agents in RL operate under uncertainty. How can the agent, solely by trial-and-error interaction with the environment, determine the best action for a given state? One of the key aspects in RL is balancing between trying new actions that have unknown rewards – *exploring* – and those actions already tried with predictable rewards – *exploiting*. The agent's goal in a RL problem is to maximize its total reward. (Sutton & Barto, 1998).

RL has been used with success in certain classes of computer games. One of the most well-known examples is Gerry Tesauro's implementation of a RL agent, called TD-Gammon, that plays backgammon at a skill-level equal to the world's best human players. In addition to proving the power of the RL approach, TD-Gammon revolutionized backgammon when it arrived at game strategies previously unexplored by grandmasters, yet found to be equally effective (Tesauro, 1995). However, the backgammon domain is starkly different from the FPS domain in that backgammon is turn-based and the world is static and fully observable. In contrast, FPS games are real-time and the world is dynamic and only partially observable. Both backgammon and FPS games are nondeterministic.

Spronck & Ponsen (2004) use reinforcement learning to generate AI opponent scripts which can adapt to a player's behavior in a real-time strategy game. A similar technique is used in a role-playing game (Ponsen et al, 2004). This work differs with ours in the granularity of the learning problem. In Spronck's work the units are learning individual policies; for example, which opponent to attack and which weapon or spell to use based upon the individual unit's capabilities. In our work, RETALIATE is learning a team policy. Another major difference is the pace of the game. FPS games are well-known for having a faster pace than role-playing and real-time strategy games.

Domination Games in UT

UT is a FPS game in which the usual objective is to shoot and kill an opponent. Players track their health and their weapon's ammunition, as well as attempt to pick up various items strewn about the map. Opponents may be other human players via online multiplayer action or computer-controlled bots. An interesting feature of UT is the ability to play several different game variants. One of these variants is a domination game, a feature offered by almost every team-based multiplayer game.

A domination game map has two or more domination locations, each of which can be owned by any team. A team scores points by controlling one or more domination locations. The last team that touches a domination location owns that domination location. For example, in a domination game with two teams and three domination locations, the team that owns two domination locations scores points twice as fast as the team that only owns one domination location. However, if the second team captures one of the first team's domination locations, the scoring reverses and the second team scores points twice as fast as the first team. A domination game ends when one team scores a predetermined number of points, in which case that team is declared the winner. In the experiments, games were played to 50 points.

RL for Coordinating Teams of Bots

We used reinforcement learning techniques to learn a winning policy to play a domination game in UT. A policy indicates for every possible state which action to take. A winning policy is one where the action taken at every state will lead to a victorious outcome.

Problem Model

A set of states and associated actions model the reinforcement learning problem domain, that we call

henceforth the ***problem model***. For an UT domination game, the states consist of the Cartesian product $\prod_i o_i$ of the owner o_i of the domination location i. For instance, if there are 3 domination locations, the state (E,F,F) notates the state where the first domination location is owned by the enemy and the other two domination locations are owned by our friendly team. Other parameters were considered to increase the information contained in each state, however, through experiments, we found not only did this simpler definition greatly reduce the state space, but contained sufficient information to develop a winning policy. For three domination locations and two teams, there are twenty-seven unique states of the game, taking into account that domination locations are initially not owned by either team. The initial state is denoted by (N,N,N); indicating that the locations are not owned by any team.

States have an associated set of ***team actions***. A team action is defined as the Cartesian product $\prod_i a_i$ of the ***individual action*** a_i that bot i takes. For example, for a team consisting of three bots, a team action would consist of the Cartesian product of three individual actions. An individual action specifies to which domination location the bot should move. For example, in one team action, the three individual actions could send bot_1 to domination location 1, bot_2 to domination location 2, and bot_3 to domination location 3. A second team action might have individual actions sending all three bots to domination location 1. For a state with three domination locations there are twenty-seven unique team actions because each bot can be sent to three different locations. If the bot is already in that location, the action is interpreted as to stay in its current location. Individual actions by different bots are executed in parallel.

Despite the simplicity in the representation of our problem model, it not only proves effective but it actually mimics how human teams play domination games. The most common error of novice players in this kind of game is to fight opponents in locations other than the domination ones. This typically leads to a severe warning by more experienced players as these fights generally do not contribute to win these kinds of games. Part of the reason is if a player is killed, it simply reappears in a random location and has no effect in the score. Also players communicate to which domination points other player should go. This is precisely the kind of behavior that our problem model represents.

Before continuing, we refer to ***game instance*** as one episode where both teams start with 0 points and the initial state is (N,N,N) and ends with one of the teams reaching the predefined number of points.

The RETALIATE Algorithm

Below we show the top level of the RETALIATE online learning algorithm. RETALIATE is designed to run continuously for multiple game instances. This is needed as there is never a final winning policy. Rather, the RETALIATE-controlled team adapts continuously to changes in the environment. These changes include: changes in our own players, changes in the opponent team (e.g., change of tactics), or changes in the world state (e.g., game instance is played in a new map). RETALIATE receives as parameters the number of games *numGameInstances* that the algorithm will be run (used in the experiments), ε: rate of random exploration, α, the step-size parameter, which influences the rate of learning. RETALIATE starts by initializing the Q-values with a default value (lines 1-3). Then runRETALIATE is called subsequently for each new game instance, passing as parameter the Q-values from the previous run (lines 4-5).

```
    RETALIATE(numGameInstances, ε, α)
    input: ε: rate of random exploration,
    α: step-size parameter
    output: none
1   For every state S ∈ {EEE, …, FFF}
2       For every action A ∈ {<Go(x), Go(y), Go(z)>… }
3           Q[S, A] ← 100
4   For i ← 1 to numGameInstances
5       Q-table[] ← runRETALIATE(ε, α, Q-table[])
```

```
    runRETALIATE(ε, α, Q-table[1..#states,1..#actions])
    input: ε: rate of random exploration, α: step-size
    parameter
    output: updated Q-table[1..#states,1..#actions]
1   BeginGame()
2   k ← 0;  Initialize state s
3   While not( EndGame() )
4       A_k ← actionsInState(s)
        Choose
            i) with probability 1- ε:
5               a ← action A satisfying max_{A∈A_k} Q[s, A]
            ii) OR with probability ε:
                a ← random action A in A_k
6       executeAction(a)          //blocking call
7       s' ← currentState()
8       R = U(s') – U(s)
9       Q(s, a) ←
            Q(s, a) + α ( R + γ max_a' Q(s', a') – Q(s, a))
10      s ← s'
11      k ← k + 1
12  Return Q[]
```

Above we show the runRETALIATE algorithm, which runs during one game instance. First the game is started (line 1), the iterator k is initialized (line 2). The following loop (lines 3-11) continues iterating while the game instance is not finished. The set of possible actions A_k for the state *s* is assigned (line 4). Line 5 picks the team action *a* to be executed; the team action with the highest Q-value for the given state is chosen with a probability 1- ε, or a random team action from all those available is chosen with a probability ε. RETALIATE executes the selected team action *a* (Line 6) and observes the resulting state, s' (Line 7). Each bot can either succeed in accomplishing its

individual action or fail in doing so (e.g., the bot is killed before it could accomplish its action). Either way, executing a team action takes only a few seconds because the individual actions are executed in parallel. Line 8 computes the reward as the difference between the utilities in states *s* and *s'*. Line 9 performs an update on the Q-value. The Q-value for the pair (state,action) from the previous iteration (k-1) is updated (we discuss this formula in detail later on). In the last instructions of the loop, the state *s* is updated and the iterator *k* is incremented (Lines 10-11).

There are three aspects of the algorithm that we want to discuss in some detail:

- **Utility**. The utility u of state s is defined by the function U(s) = F(s) – E(s), where F(s) is the number of friendly domination locations and E(s) is the number of enemy-controlled domination locations. For example, relative to team A, a state in which team A owns two domination locations and team B owns one domination location has a higher utility than a state in which team A owns only one domination location and team B owns two.
- **Action Selection**. Step 5 of the runRETALIATE algorithm shows the ε-greedy policy used by RETALIATE. Most of the time (with probability 1- ε) RETALIATE chooses the action for the current state *s* that has the maximum Q-value. But with probability ε RETALIATE chooses a random action.
- **Policy update**. In Step 9 of the runRETALIATE algorithm, the Q-value for the pair (state,action) from the previous iteration (k-1) is updated with the step-size parameter α multiplied by the difference in utility between the current state and the previous state. If the utility increases, the Q-value is increased, otherwise it is decreased. This is the standard formula for computing Q-values (Sutton & Barto, 1998).

One point to note is the diversion from the traditional discounting of rewards by setting γ = 1 instead of a value less than 1. Non-discounted rewards normally means that there would be no convergence of the algorithm to an optimal policy.[1] This means that while the Q-function may be effective for a given opponent on a given map, a change in opponent strategy or the map will result in an again unstable Q-function and the exploration for a better strategy begins again. The common practice in Q-learning of setting γ to something less than one was not effective in this highly dynamic domain because a stable convergence was undesirable.[2] The strategy needs the ability to adjust to changing opponents and environments quickly enough to be effective.

Team Name	Description
HTNBot	The HTNbot is described in (Hoang *et al*., 2005); it uses HTN planning techniques trying to maintain control of half plus one of the locations.
OpportunisticBot	Does not coordinate among members whatsoever but each bot actively looks for a domination location to capture it.
PossesiveBot	Each bot is assigned a single domination location that it attempts to capture and hold during the whole game
GreedyBot	Attempts to recapture any location that is taken by the opponent

Table 1: Description of the 4 teams used in testing and training of RETALIATE.

Empirical Evaluation

We ran two experiments. For these experiments we used the Gamebots distribution (Gamebot, 2005) and the UT game. All bots use the same state machine to control their reactive behavior. For example, they fire at an enemy on sight, or pick up an inventory item if near one. This ensures a fair comparison of team strategies as no individual bot is better than the others. Pathfinding is determined by the UT game. While moving, the bots continue to react to game events as determined by their reactive state machine. Initially the bots do not know the location of the domination location. Therefore they are programmed to find those locations first. The information about locations found is shared among teammates.

We used the DOM-Stalwart map that comes with the Gamebots distribution. This map contains three domination locations. For every game instance of each experiment, we recorded the map used, the time taken, the score history, and the number of iterations of runRETALIATE.

Because of the nature of the game and the real-time updates and actions of the bots, an average of 150 actions are taken during the course of a single game. This allows RETALIATE to develop a strategy during the course of single game as opposed to many games.

RETALIATE was called with the following values for the parameters. ε: 0.1, α: 0.2, and numGameInstances : 10. These parameters were established by doing statistical sampling based on some trial runs.

[1] **Errata from printed version**: Non-discounted rewards will converge if there are no cycles in the state space. In our domain, there are cycles.

[2] **Errata from printed version**: regardless of the value of γ, Q-learning will adapt to changes in the environment. Setting γ to something less than one, resulted in a speedup in the development of a winning strategy in this domain.

Changing Environments

In the first experiment, we ran 5 trials of 10 game instances each. A trial was a single run of RETAILIATE with numGameInstances set to 10 (i.e. the q-table was retained across the 10 games). Within each trial, we ran RETALIATE against 3 opponent teams: opportunistic, possessive, and greedy as described in Table 1. In order to observe how RETALIATE adapts to a changing environment, we iterated through the three opponents. On the first game of each trial, the opponent was opportunistic. On the second game of each trial, the opponent was possessive (and the third greedy). This pattern was continued thereafter.

The results of this experiment are shown in Figure 1. The x-axis indicate the game instances and the y-axis indicate the final score. The score indicates the average of 5 runs of 10 game instances each. So for example, in game instance number 5 RETALIATE lost some games and as a result the average is less than 50. In total, RETALIATE won 47 out of the 50 game instances. The results clearly show that within one game RETALIATE is able to obtain a winning policy an overwhelming majority of the times.

Dynamic Opponent

In the second experiment, we ran 5 trials of 2 game instances each (i.e. 5 trials of RETALIATE with numGameInstances equal to 2). Within each trial, we ran RETALIATE against HTNbots, a system that dynamically changes its strategies. This opponent is considered to be difficult. HTNbots (Muñoz-Avila & Hoang, 2006) uses Hierarchical Task Network (HTN) planning techniques to encode strategies that coordinate teams of bots. The HTN planner SHOP (Nau *et al*., 1999) was used to generate the strategy that is pursued by the bots. HTNbots keeps track of the game and when the situation changes dramatically SHOP is called to generate a new strategy "on the fly". HTNbots was able to beat all other opponents from Table 1 (Hoang et al., 2005). The interesting point of this experiment is that we took a knowledge-intensive opponent to compete against a learning opponent that begins with no knowledge about winning strategies in the game.

Figure 2 shows the average score over time of the first game across the 5 trials (i.e. untrained RETALIATE). We observe that the first game within each trial was very even. This is noteworthy in that in the previous experiment, RETALIATE was able to beat each of the opponents within one game. At the beginning of the match bots controlled by RETALIATE and HTNbots will explore the map to find the domination locations. However, whereas the HTNbots will only start executing the dynamic strategies once all domination locations have been found, RETALIATE will be punished/rewarded based on the state and consequently immediately start developing a policy. As a result the game is competitive from the outset.

The differences in score can be used to explain how RETALIATE is working. A positive slope in this line represents a point during which RETALIATE was exploiting a previously learned effective strategy against its opponent. The numerous dips in the slope of this line in Figure 2 depict periods of exploration for RETALIATE. Either a previously unvisited state in the game was encountered, or the previously learned strategy for that state was not effective against that opponent (and consequently a new policy was explored). In the final game, there are fewer periods of exploration by RETALIATE due to already having obtained an effective strategy against their opponent.

Figure 3 shows the average score over time of the second game across the 5 trials. We observe that early within the second game, RETALIATE begins to develop a winning strategy.

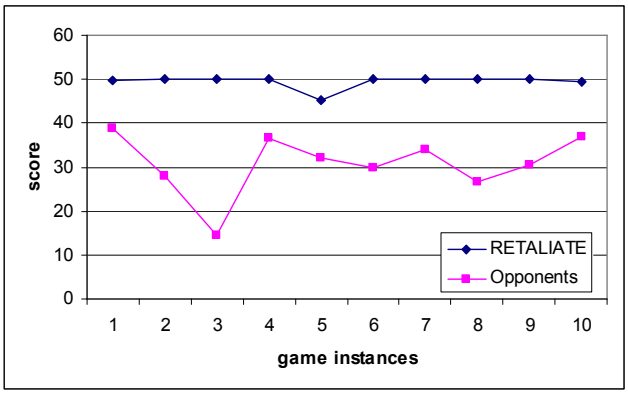

Figure 1: Results in a changing environment

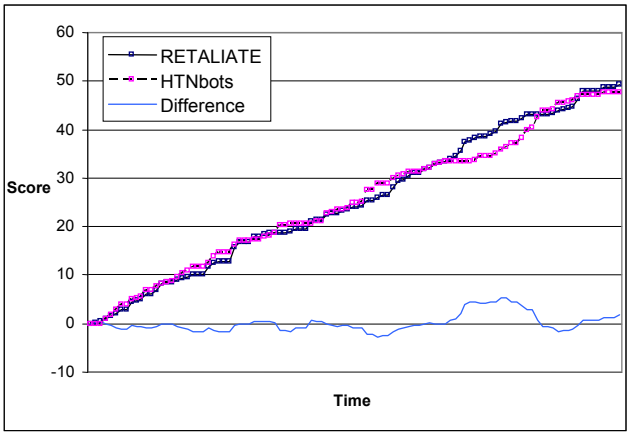

Figure 2: Results versus HTNBots, first game.

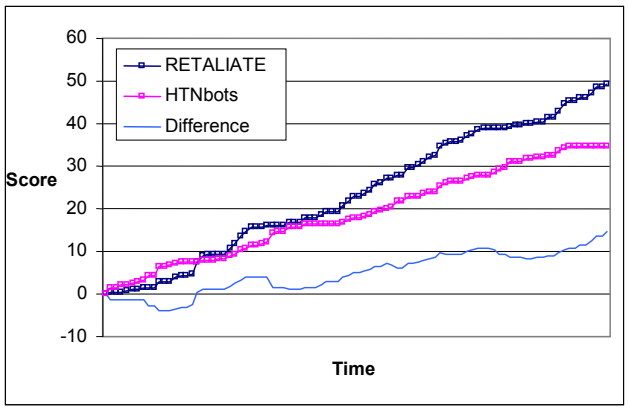

Figure 3: Results versus HTNBots, second game.

Final Remarks

The potential commercial applicability of the capability to generate adaptive team tactics is significant (van Lent et al., 1999). FPS games typically provide a toolset allowing players to design their own maps. Therefore the potential number of different maps that can be generated can be considered, from a practical perspective, unbounded. As such, encoding adequate team tactics in advance and the conditions under which they are applicable is an extremely difficult problem. This problem is compounded by the fact that in online games, teams are formed by combining players of varying capabilities; as a consequence, making a profile of team capabilities, which would allow the selection of adequate tactics *a priori*, is almost impossible.

To overcome the difficulty posed to pre-encoded strategies, we presented RETALIATE, an online reinforcement learning algorithm for team FPS games. RETALIATE voids discounted rewards in order to quickly adapt to changes in the environment. This is particularly crucial for FPS games where the world state can change very rapidly. RETALIATE's problem model is simple yet effective. Our empirical evaluation confirms the effectiveness of RETALIATE to find winning policies given our problem model.

Acknowledgements

This research is funded in part by the Naval Research Laboratory (subcontract to DARPA's Transfer Learning program), and the National Science Foundation. The opinions in this paper are those of the authors and do not necessarily reflect the opinions of the funders.

References

Bowling, M. & Veloso, M. Multiagent learning using a variable learning rate. *Artificial Intelligence*, vol. 136, no. 2, pp. 215--250, 2002.

Fu & Houlette. Constructing a Decision Tree Based on Past Experience. *AI Game Programming Wisdom 2*. Charles River Media, 2003.

Gamebot. http://www.planetunreal.com/gamebots/. Last viewed: January 24, 2005.

Hoang, H., Lee-Urban, S., and Muñoz-Avila, H. Hierarchical Plan Representations for Encoding Strategic Game AI. *Proceedings of Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*. AAAI Press. 2005

Kent, T. Multi-Tiered AI Layers and Terrain Analysis for RTS Games. In: *AI Game Programming Wisdom 2*. Charles River Media, 2003.

Laird, J.E., & van Lent, M. Interactive computer games: Human-level AI's killer application. *AI Magazine*, 22(2), 15-25. 2001

Lauer, M. & Riedmiller, M. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pp 535-542, 2000.

Matsubara, H., Noda, I., & Hiraki, K. Learning of cooperative actions in multiagent systems: a case study of pass play in soccer. In *Adaptation, Coevolution and Learning in Multiagent Systems: AAAI Spring Symposium*, pgs. 63--67, 1996.

Muñoz-Avila, H. & Hoang, H. (2006) Coordinating Teams of Bots with Hierarchical Task Network Planning. In: *AI Game Programing Wisdom 3*. Charles River Media.

Nau, D., Cao, Y., Lotem, A., & Muñoz-Avila, H. SHOP: Simple hierarchical ordered planner. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*. Stockholm: AAAI Press, 1999.

Ponsen, M. and Spronck, P. (2004). Improving Adaptive Game AI with Evolutionary Learning. *Computer Games: Artificial Intelligence, Design and Education* (CGAIDE 2004). pp. 389-396. University of Wolverhampton.

Rabin, S. Promising Game AI Techniques. *In: AI Game Programming Wisdom 2*. Charles River Media, 2003.

Reynolds, J. Team Member AI in an FPS. *In: AI Game Programming Wisdom 2*. Charles River Media, 2003.

Sen, S., Mahendra, S., Hale, J. Learning to Coordinate Without Sharing Information. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pp. 426-431. 1994

Spronck, P., Sprinkhuizen-Kuyper, I. and Postma. E. 2004. Online Adaptation of Game Opponent AI with Dynamic Scripting. *International Journal of Intelligent Games and Simulation*, Vol. 3, No. 1, University of Wolverhampton and EUROSIS, pp. 45–53.

Sutton, R. S. & Barto, A. G. *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998.

Nair, R. & Tambe, M. Journal of AI Research (JAIR), 23:367–420, 2005.

Tesauro, G. Temporal Difference Learning and TD-Gammon. In *Communications of the ACM*, March 1995 / Vol. 38, No. 3, 1995.

van Lent, M., Laird, J. E., Buckman, J., Hartford, J., Houchard, S., Steinkraus, K. and Tedrake, R., Intelligent

Agents in Computer Games. *Proceedings of the National Conference on Artificial Intelligence*, pp. 929-930, 1999.

Watkins, C. J. *Models of Delayed Reinforcement Learning*. Ph.D. thesis, Psychology Department, Cambridge University, Cambridge, UK, 1989.

Yiskis, E. A Subsumption Architecture for Character-Based Games. In: *AI Game Programming Wisdom 2*. Charles River Media, 2003.