Machine Learning 33, 201–233 (1998)

Colearning in Differential Games

JOHN W. SHEPPARD jsheppar@arinc.com
ARINC, 2551 Riva Road, Annapolis, MD 21401

Editors: Michael Huhns and Gerhard Weiss

Abstract. Game playing has been a popular problem area for research in artificial intelligence and machine learning for many years. In almost every study of game playing and machine learning, the focus has been on games with a finite set of states and a finite set of actions. Further, most of this research has focused on a single player or team learning how to play against another player or team that is applying a fixed strategy for playing the game. In this paper, we explore multiagent learning in the context of game playing and develop algorithms for "co-learning" in which all players attempt to learn their optimal strategies simultaneously. Specifically, we address two approaches to colearning, demonstrating strong performance by a memory-based reinforcement learner and comparable but faster performance with a tree-based reinforcement learner.

Keywords: Markov games, differential games, pursuit games, multiagent learning, reinforcement learning, *Q*-learning

1. Introduction

Since the genesis of the study of artificial intelligence (AI), AI researchers have found game playing to be a fertile area for exploring and expanding the capabilities of machines in problem solving. Games offer the human many challenges and opportunities for exploring his or her own abilities in finding strategies for personal advance—generally at the expense of the opponent. Attempting to install game playing abilities in the computer has opened new avenues for studying approaches to efficient search, pattern recognition, classification, and knowledge representation.

Initially, research in computer game playing was limited to constructing fixed strategies for the computer to apply against a human opponent. The worth of the strategy was determined based on how well the computer fared against the human. How many times did the computer win? How long did the game last? Was the computer, at least, an interesting opponent to play?

Until Arthur Samuel developed his checkers player (Samuel, 1959), the thought of constructing a machine that could "learn" to play a game capable of competing with a human was just a dream. With Samuel's checkers player, artificial intelligence took a step forward, demonstrating that a mere machine could not only be programmed to solve complex problems but could actually *learn* how to solve these problems by applying knowledge gained from previous experience. Since Samuel built his learning checkers player, the field of machine learning and the study of game playing have come together to yield several significant advances.

To date, research in multiple agent planning and control has been limited largely to the area of distributed artificial intelligence (Rosenschein & Genesereth, 1985; Stone & Veloso, 1996a; Suguwara & Lesser, 1993; Tan, 1993) and artificial life (Collins, 1992; Huberman & Glance, 1995; Sandholm & Crites, 1995; Stanley, Ashlock, & Tesfastsion, 1993). In distributed AI (DAI), several agents cooperate to achieve some goal or accomplish some task. The task is usually one of sufficient complexity that no single agent can accomplish the task alone. Because the agents cooperate, research in distributed AI has focused primarily on developing efficient procedures for communicating between the agents to enable the agents to develop the cooperative plans. Artificial life research, on the other hand, does explore issues related to both cooperation and competition, but its primary focus is on the emergence of intelligent behavior in a population of agents. For example, work on the iterated prisoner's dilemma has been useful to characterize mating habits (Sandholm & Crites, 1995; Stanley, Ashlock, & Tesfatsion, 1993).

Recently, work has begun to appear that focuses on learning in multiagent systems (Grefenstette, 1991; Schmidhuber, 1996; Tan, 1993). Problems in multiagent systems are distinct from problems in DAI and distributed computing, from which the field was derived, in that DAI and distributed computing focus on information processing and multiagent systems focus on behavior development and behavior management (Stone & Veloso, 1995). In addition, problems in multiagent systems are distinct from problems in artificial life in that multiagent systems focus on individual behaviors and artificial life focuses on population dynamics (Collins, 1992). So far, most work in learning and multiagent systems has emphasized multiple agents' learning complementary behaviors in a coordinated environment to accomplish some task, such as team game playing (Stone & Veloso, 1996b; Tambe, 1996a), combinatorial optimization (Dorigo, Maniezzo, & Colorni, 1996), and obstacle avoidance (Grefenstette, 1991).

The research discussed in this paper focuses on exploring methods of learning in the context of competitive multiagent systems. In particular, we focus on exploring methods for the on-line learning of optimal strategies for playing differential games and developing approaches to learning approximate optimal solutions to discrete Markov versions of these games. Differential games are related to control problems in that the objective is to determine values of a set of control variables that optimize some objective function (namely, payoff) while satisfying the constraints of the game. The games are related to planning problems in that each player, independently, attempts to make a decision to force the state of the game into the best state for that player.

The major contributions of this paper include the development and evaluation of two novel algorithms for colearning approximate optimal strategies to several differential games studied in the literature. The first algorithm is a memory-based algorithm in which the players share a common memory base, and the second is a tree-based algorithm in which the players share a common decision tree. We decided to use a common data structure for both agents to speed the convergence to optimal since the objective of the research was learning optimal strategies rather than simulating biological learning.

Because differential games assume simultaneous actions by the players, these games are more general than traditional games in extensive form. Consequently, the results of the research presented in this paper can be applied to the general class of two-player games in extensive form, with and without behavioral strategies.

2. Learning and Markov decision processes

For purposes of this paper, we restrict our attention to two-person zero-sum games of imperfect information. Further, we limit the scope to positional games. A *positional game* is a game in which the sequence of moves leading up to the current state is irrelevant in deciding the optimal strategy to apply. Thus, it is also a game of perfect recall because history does not matter. This property is called the *Markov* property and is derived from the study of Markov decision processes.

A Markov decision process (MDP) is defined by a set of states $\boldsymbol{S}$, a set of actions $\boldsymbol{A}$, a set of transitions between states $\boldsymbol{T}$, associated with a particular action, and a set of discrete probability distributions $\boldsymbol{P}$, over the set $\boldsymbol{S}$. Thus, $\boldsymbol{T} : \boldsymbol{S} \times \boldsymbol{A} \rightarrow \boldsymbol{P}$. Associated with each action while in a given state is a cost (or reward), $c(s, a)$. Given a Markov decision process, the goal is to determine a policy, $\pi(s)$, (i.e., a set of actions to be applied from a given state) to minimize total expected discounted cost.

Research in learning and MDPs has focused on developing approaches to finding optimal policies in MDPs when the state or action spaces becomes large. Many of the algorithms are derivatives of value and policy iteration but focus on sweeping only parts of the space. Other algorithms apply ideas from reinforcement learning to function approximators to learn the policies.

Barto et al. (Barto, Bradtke, & Singh, 1993; Barto, Sutton & Watkins, 1991) describe two alternative approaches to solving an MDP that are both forms of *asynchronous* dynamic programming. The first approach, which they refer to as “asynchronous dynamic programming,” is a derivative of Gauss-Seidel dynamic programming (Bertsekas, 1987); however, where Gauss-Seidel dynamic programming still performs a systematic “sweep” of all states, asynchronous dynamic programming allows states to be updated at arbitrary points in time. When a state is updated, it uses the current values of successive states.

The second approach is called real-time dynamic programming (RTDP) because it provides an on-line learning strategy rather than the traditional off-line strategies of other dynamic programming algorithms. RTDP applies a greedy strategy with respect to the current estimate of value function $V(s)$, $\hat{V}(s)$, to define the policy for the controller. Whenever an action is taken, the cost/payoff of that action is applied immediately to update $\hat{V}(s)$. To ensure convergence of RTDP, it is necessary to visit all states “infinitely” often. One approach used to ensure this is to require that all states be selected infinitely often as initial states.

One of the problems with methods such as asynchronous dynamic programming and RTDP is that these methods require a complete understanding of the transition probabilities, $p(s' \mid s, a)$, underlying the MDP (Barto, Bradtke, & Singh, 1993). They also require knowledge of the immediate costs, $c(s, a)$. The requirement to know the cost holds, in particular, in the off-line case but can be relaxed when learning on line as in RTDP. In many control tasks, such as the differential games studies in this paper, such knowledge may not be available.

A conceptually simple approach to solving MDPs with incomplete knowledge, called Q-learning was proposed by Watkins in 1989 (Watkins, 1989; Watkins & Dayan, 1992). In Q-learning, the controller maintains estimates of the optimal Q values for each admissible state-action pair. These Q values are estimated based on experience applying admissible actions in each state, rather than based on an evaluation function that includes the state-transition probabilities.

During control, the controller keeps track of the succession of states visited, the actions taken in each state, and the costs incurred as a result of taking the actions in each state. Either during control or upon termination, the Q values are updated as follows. Let $Q_t(s, a)$ be the Q value at time t when action a is performed in state s. Then this Q value is updated by computing

$$Q_{t+1}(s, a) = [1 - \alpha_t(s, a)]\, Q_t(s, a) + \alpha_t(s, a)[c(s, a) + \gamma\, Q_t(s', \pi(s'))]$$

where $\alpha_t(s, a)$ is the value of the learning rate for state-action pair $\langle s, a\rangle$ at time t, γ is the discount rate, $Q_t(s', \pi(s')) = \min_{a \in A}\; Q_t(s', a)$, and s' is the successor state.

In reinforcement learning, considerable attention has been given to Sutton's Temporal Difference Learning algorithm (Sutton, 1988; Dayan, 1992; Tesauro, 1992). The temporal difference method is intended to be applied in "multistep prediction problems" where payoff is not awarded until several steps after a prediction for payoff is made. This is exactly the problem that arises with delayed reinforcement. At each step, an agent predicts what its future payoff will be, based on several available actions, and chooses its action based on the prediction. However, the ramifications for taking the sequence of actions are not revealed until (typically) the end of the process. It has been shown that Q-learning is a special form of temporal difference learning where the "look-ahead" is cut off. Specifically, Q-learning is shown to be equivalent to TD(0) when there exists only one admissible action in each state (Barto, Bradtke, & Singh, 1993; Dayan, 1992).

3. Machine learning and games

Although games have been a popular topic for study in artificial intelligence and machine learning, little research has been done in multiagent learning and games. As with other problems in reinforcement learning, learning strategies for game playing can be posed as a control problem. The object is for an agent (or player) to learn the "best" or "optimal" strategy to use against its opponent. In the context of two-player games where one player is applying a fixed strategy, the second player optimizes its strategy to yield maximum payoff in the game. When both players are attempting to learn, each must be sensitive to the fact that opponent's strategy is not fixed. Thus, what may be "optimal" in one context (i.e., with a particular strategy applied) will not necessarily be optimal in another context. Of course, if the expected payoffs are known for the various joint strategies, then "learning" reduces to solving the game.

It is interesting that one of the earliest success stories in machine learning was an approach similar to temporal difference learning applied to game playing. In 1959, Arthur Samuel reported on experiments he performed with a computer learning an evaluation function for board positions in the game of checkers (Samuel, 1959). Samuel's idea was to use experience from actual play to learn the evaluation function. Then the computer could adapt its play to improve its performance by gradually improving the ability of the evaluation function to predict performance. The evaluation function was updated in the course of playing several games in which one player, using the current evaluation function, challenged a second player, using the best evaluation function found so far.

More recently, Gerald Tesauro applied temporal difference learning in self-play to the game of backgammon (Tesauro, 1995). As with Samuel's checkers player, Tesauro's TD-Gammon has the two players playing each other using an evaluation function (implemented as a feed-forward neural network). The approaches differ in that, for TD-Gammon, both players use the *current* evaluation function that has been learned. In addition, where Samuel constructed several abstract features for the terms in his scoring polynomial, Tesauro processes raw state information.

In the economics community, Roth and Erev (Erev & Roth, 1995; Roth & Erev, 1995) explored what they call "cognitive game theory." They focused their research on limited rationality game theory to facilitate modeling the learning process. Their goal was to better understand the nature of different economic games and the limitations of learning "rational" strategies to play these games. The learning model used by Roth and Erev is a relatively simple reinforcement learning model and applies to both players in a two-person game. In addition, compared to the self-play methodology of Samuel and Tesauro that requires homogeneous agents, Roth and Erev represented the players separately, thus allowing for heterogeneous agents.

Michael Littman explored using Q-learning for colearning among homogeneous agents in the context of Markov games (Littman, 1994, 1996). It appears his approach also applies to heterogeneous agents, but no such experiments were reported. A Markov game is a special form of Markov decision process in which actions by two (or more) competing players jointly determine the next state of the game. Solving the Markov game consists of developing a policy for play that maximizes the expected payoff to each player under the assumption the other players are playing optimally.

Littman proposes the following approach to applying Q-learning to solve two-person Markov games. Assuming a lookup table mapping current state-action-action triples to Q-values exists, play consists of selecting actions either at random (to promote exploration) or according to the current policy. This policy is given by returning an action according to mixed strategies derived for one player which is then fixed to permit selecting the other player's action through simple maximization (Littman, 1996). The mixed strategy for the first player is determined by solving the linear program derived from a game matrix of expected payoffs taken from the Q table.

In independent research, Harmon, Baird, and Klopf investigated applying reinforcement learning in function approximators (specifically, artificial neural networks) to learning solutions to differential games (Harmon, Baird, & Klopf, 1995). For their research, they focused on a single linear-quadratic differential game of pursuit in which a single missile pursues a single airplane, which is similar to the evasive maneuvers problem studied by Grefenstette et al. (Grefenstette, 1988; Grefenstette, Ramsey, & Schultz, 1990; Harmon, Baird, & Klopf, 1995; Rajan, Prasad, & Rao, 1980). As a linear-quadratic game, the kinematic equations are linear functions of the current state and action, and the payoff function is a quadratic function of acceleration and the distance between the players. In game playing, Harmon et al. note that optimal play may require application of mixed strategies. To simplify their experiments, however, they chose to assume pure-strategy solutions existed for their game.

Recently, work in coevolutionary algorithms has begun to suggest approaches to multi-agent colearning with some encouraging initial results. Potter, DeJong, and Grefenstette developed an approach to coevolution in which an agent is decomposed into "subagents" each responsible for learning an activity to be combined to solve a complex task (Potter, De Jong, & Grefenstette, 1995). In this approach, multiple agents operate on a single task in parallel. The agents are initialized with rules to bias their activity toward some subset of the total problem. The multiagent (or composite) plan then consists of the concatenation of the best plans learned by the subagents.

Grefenstette and Daley consider another coevolutionary strategy for cooperative and competitive multiagent plans (Grefenstette & Daley, 1995). These are the first experiments by Grefenstette et al. in which multiple competitive agents learn simultaneously. In their approach, rather than coordinating subplans, each agent is responsible for its complete plan. They apply their approach to a food-gathering task in which two agents compete against each other to obtain the most food.

Smith and Gray describe an alternative approach to coevolution applied to the game of Othello, which they call a coadaptive genetic algorithm (Smith & Gray, 1993). Their approach focuses on developing a fitness function that is derived from the ability of a member of the population to compete against other members of the population. Their coadaptive fitness function appears to be a variation on tournament selection, except that selection takes place at the end of a complete round-robin tournament.

The research described in this paper is similar to the work of Littman (1994) and Harmon et al. (1995) except the focus is on using colearning to determine *optimal* strategies in games with simultaneous play. For these experiments, behavior strategies (i.e., mixed strategies at each stage in the game) are determined for *both* players based on learned expected payoffs (Sheppard, 1996). Two novel algorithms are presented and compared—a memory-based algorithm and a decision-tree-based algorithm.

In static games, the strategies of all players are generally known. Further, in solving static games, payoffs assigned to selected strategy combinations are also known. If the players do not know the payoffs, then learning these payoffs is a variation of the k-armed bandit problem in which all players are trading exploration (determining expected payoffs for each strategy combination) with exploitation (playing the optimal mixture of strategies based on current knowledge about expected payoffs).

In dynamic games, the strategies of all the players are *not* necessarily known. Thus, it is impossible to assign payoffs to strategy combinations because the combinations are not known. Although the players do not necessarily know the strategies of their opponents, generally they do know the dynamics of their opponents (i.e., they know the differential equations that characterize the performance of all of the players). This leads to a solution concept in which the players attempt to characterize their strategies in terms of regions in strategy space based on the dynamics of the game and assign payoff to those regions.

4. Memory-based co-learning

Because the definition of a solution to any game (whether static or dynamic) involves determining strategies and expected payoffs for all players of the game, we wish to explore methods for the competing players to learn their optimal strategies on line. Fixing the strategy of one player while the other learns is interesting from a control theory perspective. It is uninteresting from a game theory perspective, however, because the learning agent is focusing on a single competing strategy. Usually, competing strategies are unknown, and players must learn to maximize their expected payoff in the presence of this unknown. In addition, in the case of learning strategies, the dynamics of the players may be unknown as well (i.e., the players do not know the differential equations characterizing performance).

Traditionally, memory-based learning consists of storing examples of classification instances or previous experiences in a memory base (i.e., the memory) and searching the database for “similar” examples when presented with a new problem. The action taken corresponds to the action associated with the closest example (or some combination of examples) in the database. Other variants on the memory-based approach consist of constructing tables of state-action combinations and storing expected payoff with these table entries. Learning consists of updating the expected payoffs over time.

4.1. The MBCL algorithm

Our memory-based approach (which we call *MBCL*) is based on a combination of these two variants. In particular, we use k-nearest neighbor to identify the examples that most closely match the current state of the game. We also use Q-learning to update expected payoff associated with each of the examples in the database. A high-level description of the learning algorithm is shown in figure 1.

The first step in the algorithm involves seeding the population with an initial set of games. This step is needed because memory-based reasoning cannot function without a preexisting memory base. Seeding consists of generating several random games. Specifically, games are generated where one-third of the games apply random actions for both players, one-third of the games fix P (i.e., the pursuer) on a single random move per game and generate random actions for E (i.e., the evader), and one-third of the games fix E on a single random move

```
algorithm MBCL
    seed(memory);
    do
        game = play-game(memory);
        store-game(game,memory);
        update-Q-values(memory);
    enddo;
end;
```

Figure 1. High-level pseudocode of memory-based colearning algorithm.

```
algorithm play-game(memory);
        neighbors = find-neighbors(state,memory,$k_s$);
        max-E-move= find-max-move(neighbors,E);
        min-E-move = find-min-move(neighbors,E);
        max-P-move = find-max-move(neighbors,P);
        min-P-move = find-min-move(neighbors,P);
        moves-E = partition(min-E-move,max-E-move,n);
        moves-P = partition(min-P-move,max-P-move,n);
        for i = 1 to n
                for j = 1 to n
                        move-neighbors = find-neighbors(moves-E[i],moves-P[j],$k_m$);
                        expected-payoff[i][j] = $\sum_{p=1}^{k_m} w_p move-neighbors.\rho_p$
                endfor;
        endfor;
        E-move = linear-program(expected-payoff);
        P-move = dual-linear-program(expected-payoff);
        game[state] = update-state(E-move,P-move,state);
        return(game)'
end;
```

Figure 2. Pseudocode for playing the differential game.

per game and generate random actions for P. The initial Q values associated with each of the examples are largely irrelevant, so we assign the value of zero to all nonterminating moves and the actual payoff of the game to the terminating moves.

When playing the game, the moves for both players are determined by examining the memory base. This process is illustrated by the pseudocode in figure 2. Following this procedure, in each state of the game, the k_s neighbors to the current state in the memory base are found (where k_s is the number of neighbors to the current state). Next only the range of moves found among the neighbors are considered (i.e., we consider the minimum and maximum values of the set of actions returned with the examples), and the range is partitioned into n representative moves for each player (i.e., we subdivide the range into n equal size intervals and consider the midpoints of these intervals as representative moves). These moves are used to determine the k_m nearest neighbors based on stored moves from among the k_s neighbors found previously (where k_m is the number of neighbors among the k_s examples to a specific combination of moves).

Because n moves are considered for each player, and we wish to compare the performance of the two players on each of the pairs of moves, we can construct a payoff matrix of the form

$$\begin{pmatrix} E[\rho_{1,1}] & E[\rho_{1,2}] & \cdots & E[\rho_{1,n}] \\ E[\rho_{2,1}] & E[\rho_{2,2}] & \cdots & E[\rho_{2,n}] \\ \vdots & \vdots & \ddots & \vdots \\ E[\rho_{n,1}] & E[\rho_{n,2}] & \cdots & E[\rho_{n,n}] \end{pmatrix}$$

where $E[\rho_{i,j}]$ represents the expected payoff when pairing moves i and j. This matrix is the basis for the tableau used to solve the corresponding linear program and its dual. The moves for the two players are then selected from the resulting mixed strategies. Because we quantize the possible actions to construct a tableau for linear programming, when one of these discrete actions is selected, an actual action is generated at random from the interval [action $-\delta$, action $+\delta$], where δ is equal to one-half of the size of a partition, derived from subdividing the original range of moves.

For each entry in the matrix, we compute the expected payoff as $E[\rho_{i,j}] = \sum_{p=1}^{k_m} w_p\, \rho_p$, where ρ_p is the payoff associated with example p, and $\rho_{i,j}$ indicates the matrix entry for the ith E-move and the jth P-move. The weights w_p are computed as,

$$w_p = e^{\left(\frac{-f_d(x_p,x_q)^2}{2K_w^2}\right)}/w^*$$

where K_w is a smoothing kernel width that determines the distance over which the weight is significant, w^* is the sum of the exponentials, used as a normalizer, and f_d is the normalized Euclidean distance between x_p and x_q. Here x_p corresponds to the example p and x_q corresponds to the pair of moves given by the current values for i and j.

When learning, we update the associated payoffs with the stored points using Q-learning. The points actually updated consist of the k_s nearest neighbors identified in each state. As indicated above, the initial payoff stored with the points will be the actual payoff received from the first play. Subsequent updates will only occur if the point is one of the k-nearest neighbors in some state. In this case, the Q-learning update rule is applied.

$$Q(s_i, a_p, a_e) = (1 - \alpha_j)Q(s_i, a_p, a_e) + \alpha_j(\rho + \gamma Q(s', \pi(s')))$$

where $Q(s_i,\ a_p,\ a_e)$ is the current Q value associated with applying the pair of actions $\langle a_p,\ a_e\rangle$ in state s_i, α_j is a learning rate associated with the specific point being updated, γ is a discount factor, $Q(s', \pi(s'))$ is the maximum Q value in state s', and s' is the state resulting from applying actions $\langle a_p, a_e\rangle$.

4.2. Experiments

To evaluate *MBCL*, we ran several experiments using four differential games. These games include a simple game of force, a pursuit game with simple motion, an extension to the pursuit game in which a boundary is placed at $x = 0$, and a pursuit game in which both the pursuer (P) and the evader (E) have limited mobility. This latter game is an extension of the traditional Homicidal Chauffeur game (Isaacs, 1975) which limits the mobility of P but not E. To reduce the size of the state space, all games are played with state variables relative to P. Further, all game matrices are constructed such that each player has $n = 10$ strategy ranges. Since the focus of this research is to provide alternative approaches to analyzing and solving differential games, the games were selected to represent the types of games studied in differential game theory. Further, the games were selected to show a steady increase in difficulty in determining a solution, rather than focusing on issues of dimensional scalability.

4.2.1. A game of force. For our first experiment, we consider a differential game in which two players are applying a force to a falling object in an attempt to make the object land at a certain point. Even though this game is not a pursuit game, we refer to the players as P and E for historical reasons. For player P, the objective is to push the object as far to the left as possible; player E is attempting to push the object as far to the right as possible. Each player is constrained differently, thus requiring different strategies. This means that a single strategy cannot be learned through self-play and given to both players. Thus, they are heterogeneous agents (Stone & Veloso, 1996a). For P, the magnitude of the force is fixed and P must determine the appropriate angle with which to apply the force. For E, the angle is fixed and E must determine the appropriate magnitude of force to apply.

The dynamics of the game are given by

$$\begin{aligned}\dot{x} &= Av + B\sin u\\ \dot{y} &= -1 + B\cos u\\ |v| &\leq 1\\ u &\in [0, 2\pi]\end{aligned}$$

where A and B are game parameters defining the dynamics of the game, and u and v are the controls set by P and E, respectively. The payoff function is defined to be the x value at the point the object lands. For this game, with $A = B = 1$, we expect optimal solutions when $u = \frac{3\pi}{2}$ and $v = 1$.

For these experiments, several parameters needed to be set. In particular, we ran each simulation for 10,000 games and tested the results of learning after every 250 games. For each test, we played 50 games and averaged the "payoff" received after each game. For each of the 50 test cases, we used a uniform probability distribution and randomly generated a new starting position such that $x_0 \in [-0.25, 0.25]$ and $y_0 \in [0.85, 1.0]$. For the game parameters themselves, we set $A = 1$ and $B = 1$. For the learning algorithm, we fixed the learning rate at $\alpha = 0.15$ and set $\gamma = 0.95$, $k_s = 30$, and $k_m = 5$. We also defined the kernel, $K_w = 4$.

The results of this experiment are shown in figure 3. This graph was produced by running the experiment ten times and averaging the results at 250 game intervals. The curve at the center of figure 3 marked with a cross shows optimal play throughout the experiment. The curve marked with diamonds indicates performance of both P and E as they learn. In addition, through the course of learning, P and E played against an optimal opponent to demonstrate their personal progress. The curve marked with a square indicates P's performance against an optimal E, and the curve marked with a plus indicates E's performance against an optimal P.

It is interesting to note that P learns to play optimally relatively quickly, achieving near-optimal performance after only 1250 games. P seems to converge to optimal performance after 2000 games, and E is still struggling. Finally, it seems E gains sufficient experience, playing against what is essentially a fixed opponent, and learns to play optimally after 4000 games.

One possible explanation for this difference in performance may arise by examining the landscape of the payoff function for each player. P is permitted to select any angle in the

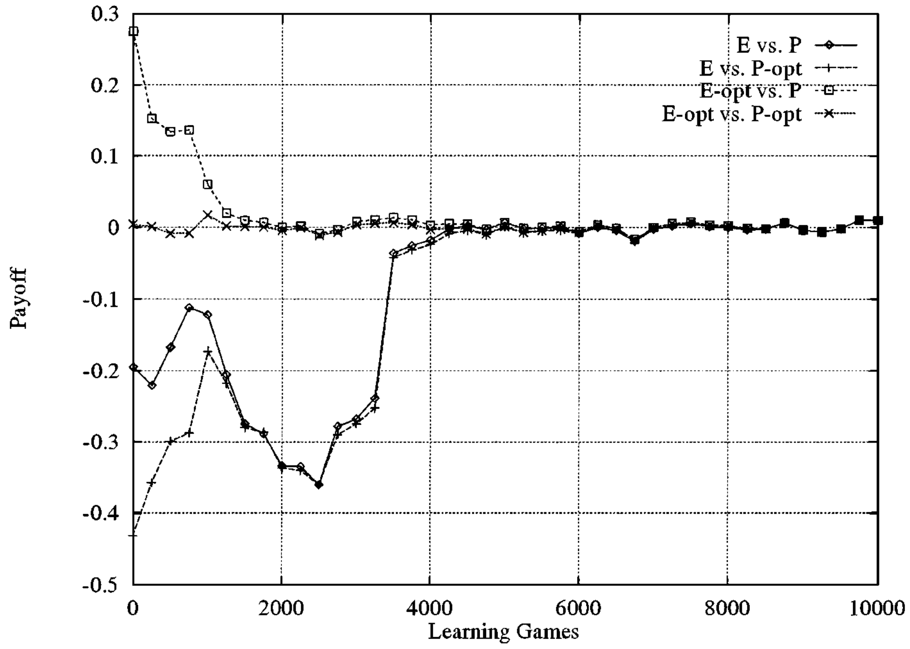

Figure 3. Learning performance for game of force.

range $[0, 2\pi)$. The optimal move comes at $u = \frac{3\pi}{2}$, and sampling the action space provides a smooth slope on either side of optimal. For E, on the other hand, the optimal move ($v = 1$), arises at the boundary of legal moves ($v \in [-1, 1]$). Sampling only has benefit on one side of optimal (because the other side is infeasible). When compared to P, which is able to sample on both sides, it is possible that E obtains half the benefit of exploration that P receives.

4.2.2. Pursuit with simple motion. For the second experiment, we consider a differential game in which one player is pursuing another player in a two-dimensional playing field. For player P, the objective is to capture E. For player E, the objective is to evade P. For purposes of this and subsequent games, a time limit was set to 200 discrete steps to bound the time required to run the experiments. Neither player has any constraints on its mobility, meaning each player can turn instantaneously in any direction, similar to two children playing tag. Each player moves at a fixed speed, and P is twice as fast as E. The kinematic structure, with the relative coordinate system for this and subsequent games, is shown in figure 4.

Again, we see that the two players cannot be modeled as one to allow “self-play” learning. In other words, each player has different objectives and capabilities and must learn appropriate strategies on their own. Further, this game is more difficult than the game of force in that a separate action must be taken depending on the position of the opponent—no single fixed action applies.

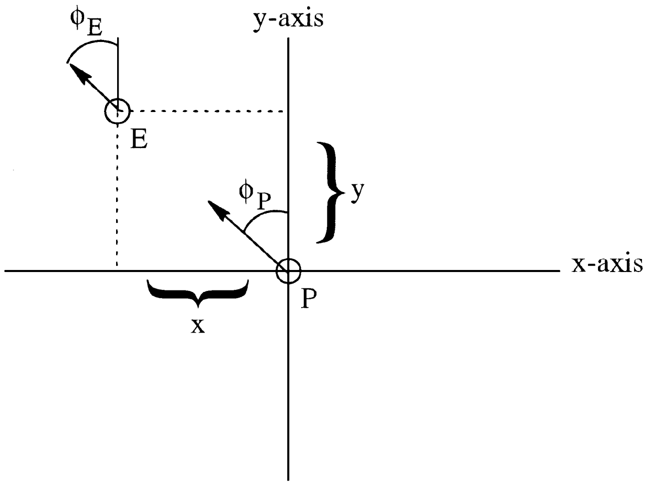

Figure 4. Kinematic structure of simple pursuit game in relative coordinates.

The dynamics of the game are given by

$$\begin{aligned}\dot{x}_P &= -v_P \sin\phi_P\\ \dot{y}_P &= v_P \cos\phi_P\\ \dot{x}_E &= -v_E \sin\phi_E\\ \dot{y}_E &= v_E \cos\phi_E\end{aligned}$$

where $\langle x_P, y_P\rangle$ and $\langle x_E, y_E\rangle$ are instantaneous positions of P and E, respectively. Also, we assume that P has a lethal envelope, $l > 0$, such that P captures E if

$$\sqrt{(x_P - x_E)^2 + (y_P - y_E)^2} \le l$$

We specify $l = 0.05$ for these experiments. Optimal solutions for the game exist for both players at

$$\phi_P = \phi_E = \arctan\frac{x}{y}$$

in coordinates relative to P. Payoff was defined to be the change in distance between P and E from the start of the game to the end of the game.

Again, several parameters needed to be set for the experiments. We ran each simulation for 5000 games and tested the results of learning after every 250 games. For each test, we played 50 games and averaged the payoff received after each game. For each of the 50 games, we used a uniform probability distribution and randomly generated a new starting position such that $x_P \in [-1, 1]$, $y_P \in [-1, 1]$, $x_E \in [-1, 1]$, and $y_E \in [-1, 1]$.

For these and all subsequent experiments, we used a variable learning rate. Specifically, a learning rate was associated with each example stored in the memory base. Initially, the

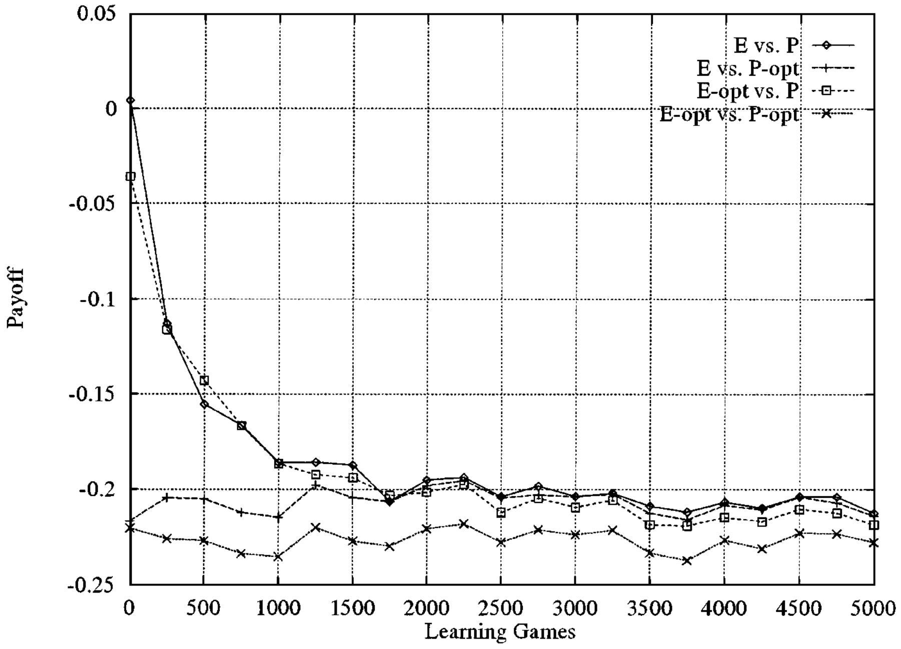

Figure 5. Learning performance for pursuit game with simple motion.

learning rate was set to 1.0 meaning that the first update of the associated Q value results in the actual payoff being assigned. Each time an instance is updated, the learning rate is changed according to the following schedule:

$$\alpha_i = \frac{1}{\chi_i}$$

where χ_i is a count of the number of times instance i has been updated, including the current time. Thus, initially, $\chi_i = 1$. In addition to the learning rates, we preset $\gamma = 0.95$, $k_s = 30$, $k_m = 5$, and $K_w = 4$.

The results of this experiment are shown in figure 5. This graph was also produced by running the experiment ten times and averaging the results at 250 game intervals. Examining this figure, we note that E is able to perform well throughout the game. It appears as if no learning is required for E to maximize its ability to evade. We know, in general, this is not true. If E does not proceed directly away from P, and P aims directly at E, then P must capture E more quickly.

When examining P's performance, we see that random motion is clearly not preferred for pursuit. In fact, initially, E is always able to get away from P. However, after only 1000 games, P has been able to direct its movements at E and brings its performance to a level comparable to optimal.

One curious result arises from examining this figure. If the simulation truly implements an optimal strategy for comparison, we would expect the optimal curve to lie between the two curves when each player is being tested against optimal players. In other words, when P and E both play optimally, the resulting performance should *always* be equal to or better than when one of the players is using the learned strategy. In figure 5, however, we find that this is not the case. Specifically, E's learned strategy appears to always beat the case where E plays optimally. Further, we find that all three curves (excluding the case where both players play optimally) converge to approximately the same performance after about 1750 games. We believe this performance is the result of the simulation quantizing the game, which has the effect of shifting the equilibrium point of the gram. We note that the players were still able to learn appropriate "optimal" strategies for this revised game. Experiments investigating the effects of quantizing the game are described in (Sheppard, 1996).

4.2.3. Pursuit with simple motion in a half plane. For the third experiment, we consider a variation of the pursuit game with simple motion in which a boundary exists at $x = 0$ thus forcing the game to take place in the half plane. On the surface, this game appears to be, essentially, the same as the previous game. In fact, this is true when the players are not near the boundary. However, E's optimal strategy changes sharply when the players are playing in close proximity to the boundary (figure 6).

This game further extends the difficulty in learning optimal strategies in three ways. First, the strategies for both players still depend on the position of the other player, but there is a new factor affecting the strategies—the boundary. Second, when play occurs in close

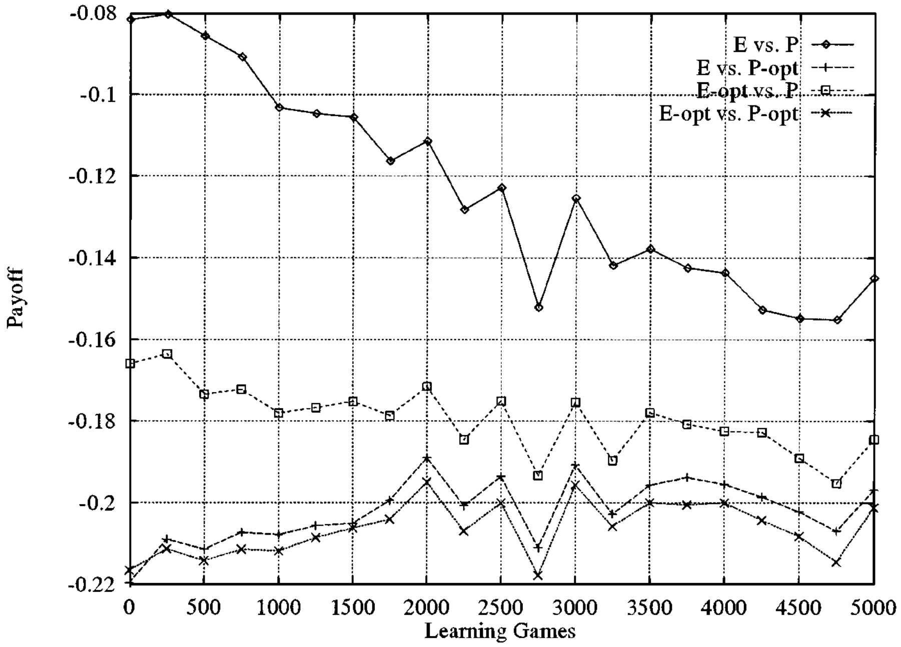

Figure 6. Learning performance for pursuit game in half plane.

proximity to the boundary, the strategies for the two players become different. Third, the transition between the two strategies for E is not smooth, indicating a discontinuity in the optimal strategy for E arising from the boundary.

The dynamics of this game are identical to the previous game. Further, both players continue to be able to move with no limitation on mobility (except for that arising from the boundary) and with fixed speeds. All of the parameters from the previous game were used for this game as well. If either E or P collides with the boundary during play, the player does not pass through the boundary but skids along the boundary a distance proportional to the y component of its force vector.

The results of this experiment are shown in figure 7. Again, the graph was produced by running the experiment ten times and averaging the results at 250 game intervals. It is interesting to note that the average performance for optimal play is slightly lower (i.e., more in favor of P) than in the previous experiments. This can be explained through the presence of the boundary. P's optimal strategy has not changed, but E's has. Further, the boundary forces E to move in such a way that would be suboptimal given the boundary did not exist. Therefore, it is reasonable to expect the boundary to favor P.

As with the previous experiment, we note that E appears to perform relatively well using the strategy implicit when the memory base was seeded (i.e., random motion). This time, however, there appears to be little convergence toward optimal. As we saw in the last section, when the boundary was absent, some convergence did occur. It is possible that the boundary can serve as an advantage to E under the condition P's reaction to E's action is delayed.

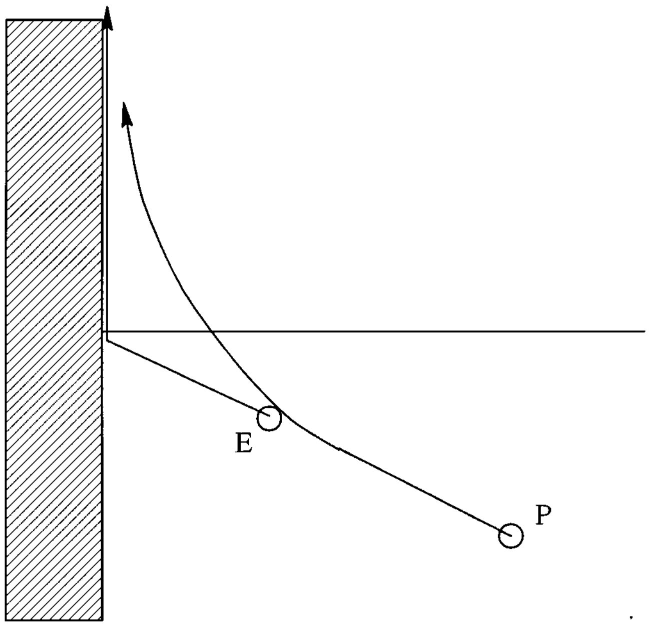

Figure 7. Optimal strategy for simple pursuit in the half plane.

We also note that P's performance does improve relative to E, but the level and rate of improvement has degraded. This would arise from the difficulties P would have when colliding with the barrier, thus slowing its advance towards E. Even though P's optimal action need not consider the boundary, the additional state variables in the examples increase the search space, making it more difficult for P to learn this fact. Indeed, the additional state information for P is "irrelevant," and irrelevant attributes are known to degrade performance in memory-based learning (Aha, 1992; Salzberg, 1991).

4.2.4. Pursuit with limited mobility. The final game we studied with *MBCL* further extends the pursuit game by limiting the mobility of both players. For this experiment, we removed the boundary, but we limited the players such that they can only make turns within a constrained range of possible turns. Specifically, we limited P's mobility such that it can turn only in the range $\pm\frac{\pi}{4}$, and we limited E's mobility such that it can turn only in the range $\pm\frac{\pi}{2}$.

This game is a generalization of the Homicidal Chauffeur game (Basar & Olsder, 1982; Isaacs, 1975; Lewis, 1994). In the Homicidal Chauffeur, only the mobility of P is limited. Given the added complexity of the game, no optimal solution was available; however, we were able to define a heuristic based on the optimal solution for the Homicidal Chauffeur. Specifically, the heuristic strategy for P was to aim, as closely as possible, at E. If turning towards E required an angle exceeding P's limits, P turned as sharply toward E as possible. E's heuristic strategy, again based on the optimal strategy for the Homicidal Chauffeur, was to turn sharply in the direction of P, attempting to get inside P's radius of curvature.

The dynamics of this game are identical to the original pursuit game with simple motion, except for the limitations on mobility. In addition, all of the experimental parameters are the same, except that we permitted learning to occur over 10,000 games rather than limiting to 5000 games.

The results of this experiment are shown in figure 8 with a comparison to heuristic play shown in figure 9. These graphs were produced by running the experiment ten times and averaging the results at 250 game intervals. This time, the curve marked with a cross indicates heuristic play by both players, the curve marked with a diamond indicates both players using their currently learned strategies, the curve with a square indicates P's performance against a heuristic E, and the curve marked with a plus indicates E's performance against a heuristic P.

Figure 8 appears to show little, if any, learning by the two players. However, if we examine figure 9, we find that learning does, indeed, occur. In all cases, E was able to evade P. When examining E's ability to play against the heuristic P, we find P losing a little ground at the start (demonstrating the power of random motions by E when P's mobility is limited) but losing about 30% more ground (relative to the heuristic) at the end of the experiment. Further, E was still improving when the experiment was terminated, albeit slowly.

When examining P's ability to play against the heuristic E, we find P losing considerable ground at the start but reducing its losses by approximately 40% (relative to the heuristic) by the end of the experiment. As with E, P was still improving when the experiment ended.

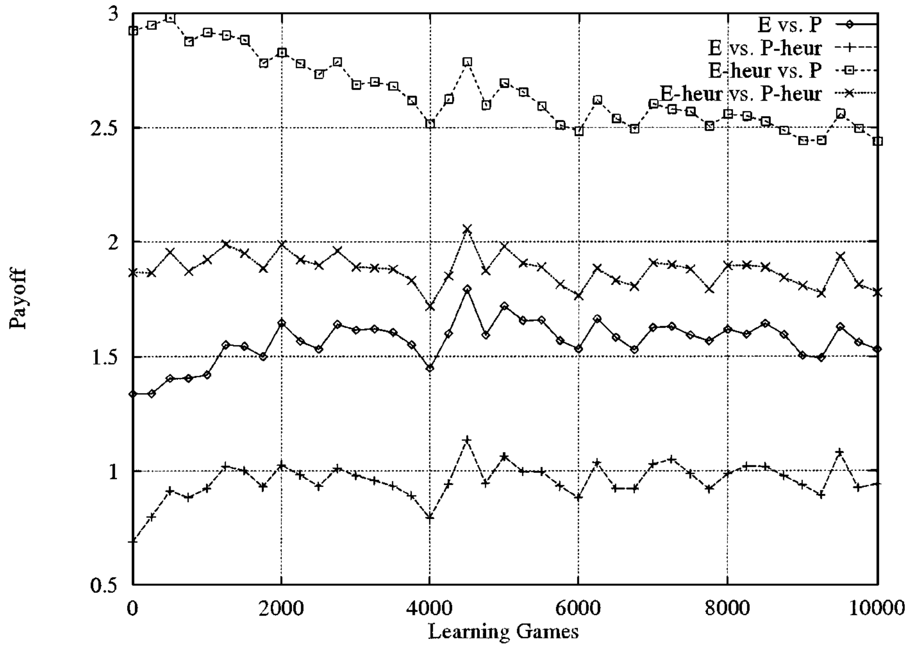

Figure 8. Learning performance for pursuit game with limited mobility.

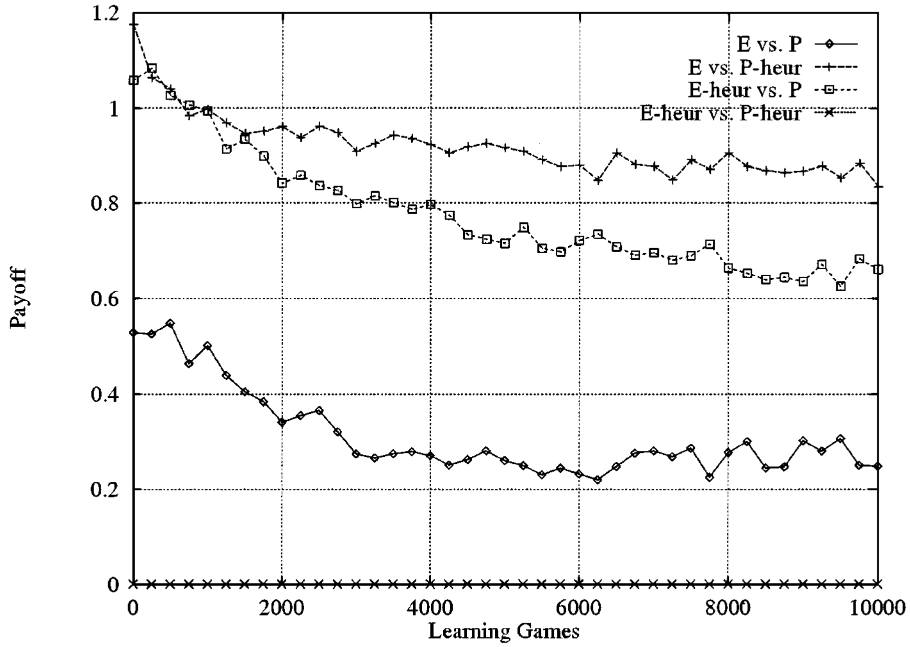

Figure 9. Deviation from heuristic for pursuit game with limited mobility.

4.3. Discussion

The results of applying *MBCL* to solving differential games are quite encouraging. In all cases, learning was demonstrated, and in the simplest cases, near optimal performance was achieved. Unfortunately, the computational burdens for learning these games was quite extensive.

As mentioned in Section 4.1, the process *MBCL* follows requires seeding the memory base with several examples. The current version of *MBCL* does not modify the memory base at all, except to update the Q values associated with each example. All of the experiments were run on either Sun Sparc 2 or Sun Sparc 10 processors. To give an idea of the computational burden, Table 1 shows the number of examples in each of the memory bases and the clock time required, in the best case, to complete the experiment.

Clearly, the time required to learn solutions to these games is excessive; however, our method of searching for neighbors did not use efficient structures to reduce search time, such as *kd*-trees. Even so, much of the computation time can be attributed to constructing and solving two linear programs at each stage of the game. The simplest game, in which only a single action needed to be found, required an hour and a half (actually less, given it converged in less than half of this time). When we considered only the one-step game of force, we found the solution immediately. The other games required approximately a day to run, and convergence only occurred with one game.

This observation points out the advantage of a proper representation for the problem to be solved. For example, when we posed the game of force as a delayed reinforcement task, we found the task learnable, but only after a large amount of simulation. On the other hand, posing the problem as an immediate reinforcement learning problem yielded a solution in one step. Considering the pursuit games, we could have represented them as immediate reinforcement learning problems as well, using the change in distance between the players as the immediate payoff. But in preliminary experiments doing just that, we found no difference in performance from using delayed reinforcement.

As can be seen, the number of examples required to learn the game of force was relatively small. The pursuit games required approximately an order of magnitude more examples and were not able to learn as well. This provides an indication of the difficulty of these tasks.

What is not shown in either the graphs or the table is that the examples for the pursuit games were probably not chosen intelligently. The approach to seeding was chosen to provide a wide sample of state-action combinations. Initial Q values did not matter since

Table 1. Relative computational burdens for solving games with *MBCL*.

Game	Games stored	Examples stored	Minutes
Force	7500	50,000	96
Simple	2000	400,000	1376
Half	2000	400,000	1372
Limit	2800	560,000	3193

they would be learned over time. Nevertheless, a uniform random sampling of the space was, apparently, not sufficient to approximate some of the surfaces encountered in these games. Applications of variable resolution techniques (Anderson & Crawford-Hines, 1994; Deng & Moore, 1995; Moore & Atkeson, 1995; Simons et al., 1982) may be more appropriate for problems such as these.

If we consider a more formal analysis of the computational burden of *MBCL*, we find the following. Assuming a fixed size memory base with n examples and d dimensions, we can consider two extremes—one where we perform a brute-force search of the memory base to find neighbors and one where we assume an efficient structure, such as a *kd*-tree. In the worst case, finding k_s neighbors will require $O(k_s nd)$ time where the efficient version would require $O(k_s d \log n)$ time. The process of populating the game matrix requires $O(k_s)$ time for the action range checking, populates $O(m^2)$ entries in the matrix, and needs search $O(k_s k_m)$ searches per entry. We applied the simplex method for linear programming; however, it is known linear programming can be performed in polynomial time using algorithms such as the interior point method. Selecting the move from the candidates is linear in m, and updating the state of the game is constant. Thus, the overall complexity is dominated by the size of the memory base (for linear search and by frequent solving of linear programs.

Nevertheless, the results from *MBCL* are highly encouraging. They indicate colearning can occur and suggest it is possible to learn optimal solutions to two-player differential games. Unfortunately, the computational resources required to learn these solutions are excessive. In the next section, we explore an alternative strategy to colearning with the focus being on reducing the computational requirements while maintaining or improving learning performance. We find several encouraging results improving the computational requirements and find that the two algorithms are comparable in their ability to learn to play differential games.

5. Tree-based colearning

In the previous section, we present an algorithm for colearning in differential games that, although providing promising results, had large computational and memory requirements. In this section, we consider an alternative algorithm that, although not memory-based, is inspired by the results of applying *kd*-trees in memory-based learning (Bentley, 1980; Deng & Moore, 1995; Moore, 1990).

A *kd*-tree is a data structure used to store a set of examples in a memory base such that nearest neighbors can be found in logarithmic expected time. Specifically, a *kd*-tree is a binary tree where each interior node of the tree tests the values of one of the attributes in the *k*-dimensional attribute space. In addition, each node corresponds to a single instance in the memory base (Moore, 1990). Nodes are selected for splitting until no further splits are required (i.e., until all points are represented in the tree).

In memory-based learning, the *kd*-tree can provide significant speed-up in searching for nearest neighbors; however, the size of the memory base does not change. Actually, the resulting memory base will be larger than a “monolithic” memory base because of the overhead associated with storing the tree.

To address the problem of storing all examples, alternatives such as editing have been offered to reduce the size of the memory base. However, striking a balance between sufficient coverage of the problem space and small size of the memory base is tricky at best. Methods for variable resolution memory-based learning also offer possible approaches to provide this balance.

5.1. The TBCL algorithm

Our approach applies the speed advantage of the *kd*-tree with the space advantage of variable resolution memory-based learning without the need to store explicit examples. Instead, we incrementally construct a "decision" tree that partitions the state space and strives to maintain balance to minimize search. Rather than storing examples at interior nodes of the tree, we store a game matrix at the leaves that represent behavioral strategies for playing the game. When performance converges, the game matrix can be discarded, and the mixed strategies associated with the game matrix retained. Further, if any of the pure strategies have an associated probability of zero, these can be dropped as well.

To describe the tree-based algorithm (which we call *TBCL*), we begin by considering the degenerate case where the tree consists of a single node. In this case, the node covers the entire state space of the game. Associated with this node is a single game matrix, pairing expected payoffs for the strategies of the two players. Mixed strategies are computed by solving the linear program defined by the game matrix. Learning consists of updating the entries in the game matrix based on actual play and re-solving the linear program.

Because most games will require different actions in different states, usually a single node with a single game tree will not be adequate. When learning converges, if the performance is not adequate, the node can be split into additional nodes and learning restarted. In the simplest case, splitting consists of selecting one of the state variables and dividing the state space along the midpoint of the dimension defined by that variable. The game matrix of the parent is then copied to each of the children, the learning rates reset, and learning proceeds as before. A high-level description of the learning algorithm is shown in figure 10.

The first step in the algorithm is to create the tree. This consists of creating a single node, covering the entire state space. A single game matrix is constructed with uniformly distributed random values. Then the corresponding linear program is solved to provide an initial set of mixed strategies for the two players to follow. This initial set of strategies is tested against 50 uniformly distributed random games, and the result is compared to a performance goal (either in terms of convergence or in terms of number of iterations).

If the performance goal is not met, *TBCL* passes into a two-part learning loop. The first part consists of performing Q-learning on the current game structure. The second part consists of selecting a node in the tree to split should the performance goal not be obtained.

In the Q-learning portion of the algorithm, a game is played and evaluated. Similarly in *MBCL*, the history of the game was stored to permit evaluation of the Q values associated

```
algorithm TBCL;
    tree = create-tree;
    initialize-game-matrix(tree);
    update-strategies(tree);
    performance = test(tree);
    do-until (performance $\geq$ goal)
        do-until converged
            game = play-game(tree);
            update-Q-values(tree,game);
            update-strategies(tree);
        enddo;
        performance = test(tree);
        if (performance < goal) then
            node = select-leaf(tree);
            split-node(node);
            update-strategies(node$\rightarrow$left-child);
            update-strategies(node$\rightarrow$right-child);
        endif;
    enddo;
end;
```

Figure 10. High-level pseudocode of tree-based colearning algorithm.

with the instances in the memory base. In *TBCL*, the sequence of the game is traversed to determine which node in the tree was used at each step and what actions were taken at that step. The corresponding cell in the game matrix at that node is then updated using Q-learning. Once the game is finished, all game matrices that were changed are solved using linear programming to find new strategies for the associated nodes.

The Q-learning loop continues until some convergence criterion is satisfied. This criterion could be a measure of the change in performance of the players, or it could be a fixed number of iterations. We chose the latter for our experiments. Once the loop finishes, performance is measured and compared against the performance goal. If the goal is satisfied, the algorithm terminates. Otherwise, a node is selected and split. Nodes are selected by considering the number of updates. The node receiving the most updates in a Q-learning loop is split because this indicates a large number of visits to the states represented by that node. Splitting takes place according to the algorithm given in figure 11. Once the node has been split and new game matrices generated for the children, these matrices are also solved, and Q-learning continues.

If we consider the algorithm in figure 11, we see that the attribute is selected that maximizes the difference in the game matrices following the split. Specifically, each attribute is considered by assuming the split is made along the attribute. Two game matrices are generated for each split. A game matrix is constructed by initializing a game from the midpoint of the partition and playing a game. The game is played according to the strategies stored

```
algorithm split-node(node);
    for-every attrib do
        split = 0.5 node.attrib.low + node.attrib.hi;
        for-every P-strat do
            for-every E-strat do
                init-game(attrib,split,LEFT);
                matrix1[P-strat][E-strat] = play-from(P-strat,E-strat);
                init-game(attrib,split,RIGHT);
                matrix2[P-strat][E-strat] = play-from(P-strat,E-strat);
            enddo;
        enddo;
        diff[attrib] = distance(matrix1,matrix2);
    enddo;
    split-attrib = argmax(diff);
    split-along(node,split-attrib);
end;
```

Figure 11. Pseudocode for splitting algorithm.

in the tree, except for the first move. All pair-wise combinations of moves are considered for this first move, and the results of the game are stored in the matrix cell indicated by the initial pair of moves.

Each pair of matrices is compared by computing the Euclidean distance between the matrices. The attribute whose pair of matrices is maximally distance is selected for splitting. When the node is split, the limits for that attribute are updated within the node, and the game matrix from the parent node is copied into each child. All of the counts used to update the learning rate for Q-learning are reset to one for that partition.

When playing the game, the moves for both players are determined by traversing the tree to find the partition that covers the current state. When the partition is found, the mixed strategies associated with each player are used directly to pick their respective moves. Because the strategies are updated at the end of each learning game, there is no need to solve the linear programs on-line as in the memory-based approach. The result is fast search through the state space and fast determination of the actions.

5.2. Experiments

We evaluated the performance of *TBCL* using the same games and same procedures as for the evaluation of *MBCL*. As before, we applied *TBCL* to four games, including the simple game of force, the pursuit game with simple motion, the pursuit game in a half plane, and the pursuit game with limited mobility. Again, we assume all states are represented relative to P, and the game matrices are constructed with $n = 10$ for each player.

5.2.1. A game of force. For the simple game of force, we used the same kinematic equations as in Section 4.2.2. Because of the nature of the algorithm, several different parameters were set. For this game, we trained for 100,000 games and only generated one node in the

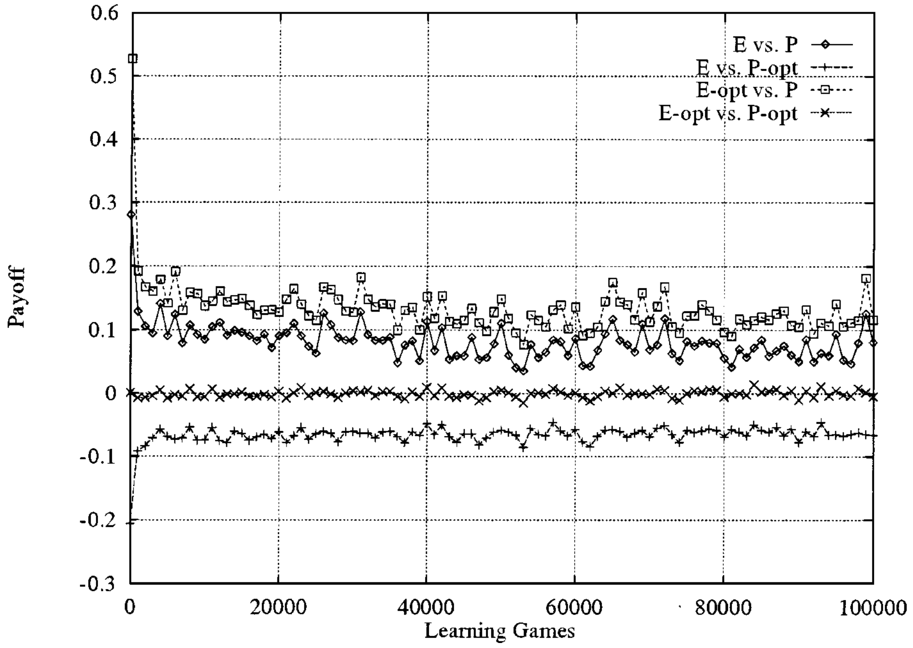

Figure 12. Learning performance for game of force.

tree. We decided not to subdivide the space given the simplicity of the game. We tested the algorithm after every 1000 games and averaged the "payoff" received after each game. For each of the 50 test cases, using a uniform probability distribution, we randomly generated a new starting position such that $x_0 \in [-0.25, 0.25]$ and $y_0 \in [0.85, 1.0]$. For the game parameters themselves, we again set $A = 1$ and $B = 1$. For the learning algorithm, we allowed the learning rate to vary and set $\gamma = 0.95$.

The results of this experiment are shown in figure 12. As before, this graph was generated by running the experiment ten times but this time averaging the results at 1000 game intervals. In these experiments, we find convergence occurs relatively quickly with performance settling after approximately 20,000 games. Performance seems to improve some through 60,000 games, but then there is a small jump causing performance to degrade followed by a return to the previous level of performance. It is also interesting to note that when P and E play each other, performance is fairly constant throughout and is degraded from optimal in favor of E. Further, performance seems to converge to this level rather than to the optimal level. This is due, most likely, to quantizing the available actions at a courser level than *MBCL* and interpolating between strategies.

5.2.2. *Pursuit with simple motion.* For the game of pursuit with simple motion, we used the kinematic equations as in Section 4.2.2. Again, we trained for a period of 100,000 games, but this time we split a node in the tree after every 5000 games. This resulted in a tree with 20 leaf nodes. We tested the results of learning after every 1000 games to monitor

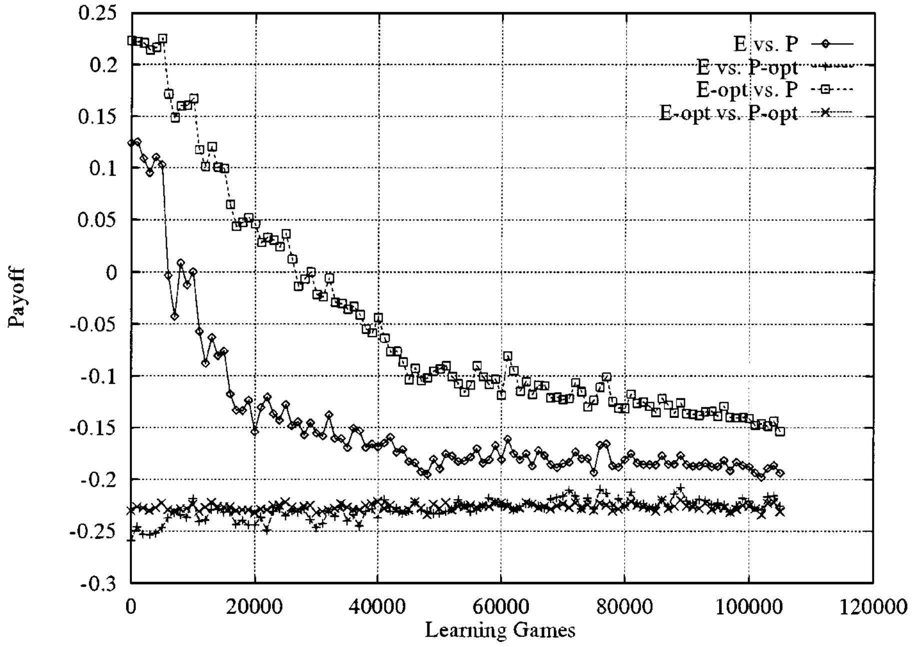

Figure 13. Learning performance for pursuit game with simple motion.

the level of convergence while a tree's structure was fixed and to observe the effects of adding a new node to the tree. Whenever we tested the performance of the algorithm we played 50 games generated at random according to a uniform probability distribution and averaged the payoff received after each game. For each of the 50 games, we generated starting positions such that $x_P \in [-1, 1]$, $y_P \in [-1, 1]$, $x_E \in [-1, 1]$, and $y_E \in [-1, 1]$. We permitted the learning rate to vary and set $\gamma = 0.95$. The results of this experiment are shown in figure 13. This graph was produced by running the experiment ten times and averaging the results at 1000 game intervals.

Examining this figure, several interesting results can be observed. First, similar to *MBCL*, performance by E appears to be good without any learning, thus indicating the power of random moves in our simulation. In fact, we find when E applies its initial strategy against an optimal P, it is able to do nearly as well as an optimal E. We do note some movement toward optimal through 40,000 games, however.

When examining P's performance, we find it starts out performing poorly, never capturing E (as shown by a positive payoff). However, after 5000 games, P has been able to improve to at least prevent E from gaining any additional ground. By the time 40,000 games have been played, P is able to advance on E fairly consistently. When the experiment was terminated at 100,000 games, the slope of P's learning curve indicated it was still improving.

Taking a closer look at these learning curves reveals an interesting, but not surprising, behavior of *TBCL*—especially when examining the performance of E playing against P rather than their optimal counterparts. Notice that performance is fairly constant through the first 5000 games. At 5000 games, the first split occurs and average payoff drops from 0.1 to about 0.0. This suggests a single node was not sufficient for improving the performance of either player. In fact, if we examine the performance of each player against the optimal counterpart, we find similar flat performance. Examining performance between 5000 and 10,000 games, we find a similar flat trend. When the tree splits again at 10,000 games, a similar change in payoff is experienced with average payoff dropping from 0.0 to about −0.075. We find yet another drop at 15,000 games. From approximately 20,000 games onward, we do not see any additional sudden changes but note a relatively steady improvement when examining *P*'s performance. It is possible the performance change should still be attributed to the tree splitting, but by this time the impact of splitting on the total tree is so small, it is difficult to discern the reason for improvement.

5.2.3. Pursuit with simple motion in a half plane. Continuing with the experiments, we next added the barrier at $x = 0$ to play the pursuit game in the half plane (see Section 4.2.3). All of the parameters used in this experiment were identical to the parameters in Section 5.2.2. The dynamics of the game are identical to the previous game, and each player still has the ability to turn instantaneously in any direction (except when constrained by the boundary). If either *P* or *E* collides with the boundary, the player skids along the boundary a distance proportional to the *y* component of its force vector.

The results of this experiment are shown in figure 14. This graph was produced by running the experiment ten times and averaging the results at 1000 game intervals. We find

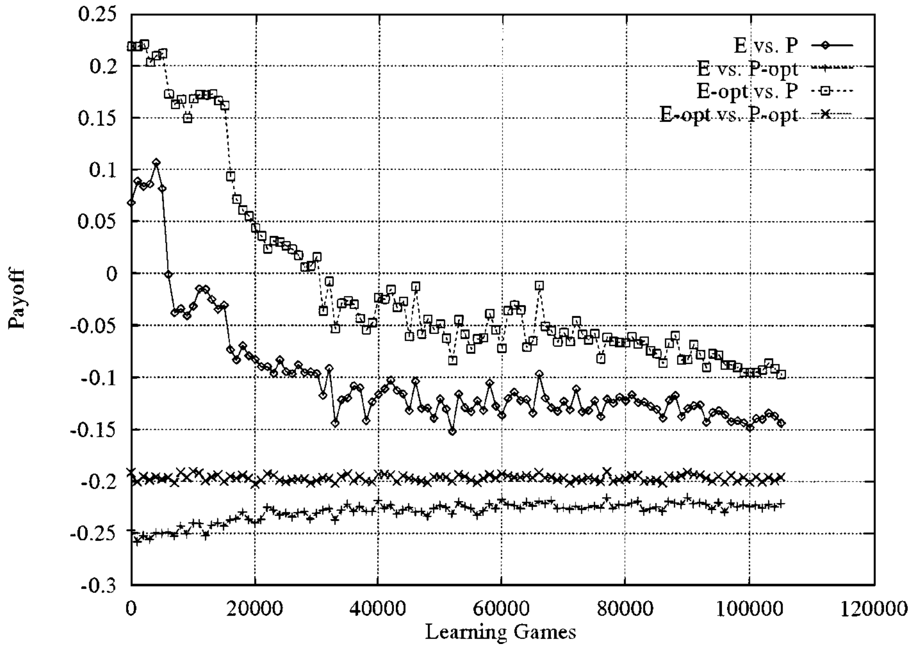

Figure 14. Learning performance for pursuit game in half plane.

that the performance of each player is similar to performance without the barrier, except that the barrier apparently causes some difficulties that need adaptation. In particular, we note that E does not start performing “optimally” as before, but its performance is still relatively good. Further, as learning proceeds, E clearly changes its strategy and approaches optimal, indicating the game is more difficult for E than the game without the barrier. After 100,000 games, E’s performance appears to have flattened out.

For the pursuer, performance is also similar to no boundary. Again, we find that performance still seems to be improving through 100,000 games. We also see the trend moving from E always getting away to E losing ground. Nevertheless, the performance compared to Section 5.2.2 is degraded as well.

These degradations in performance are not surprising for several reasons. First, we introduced an “obstacle” that increases the state space (we must now keep track of our distance to the boundary). Second, this boundary complicates E’s strategy due to the sudden shift in performance when E approaches the boundary.

We also note that the impact of node splitting is visible again. This time, however, there appears to be a slight improvement for both E and P during the first 5000 games. As before, a sudden change in performance occurs when the first split is made, but the second split (at 10,000 games) has no noticeable effect. This suggests the possibility of *TBCL* periodically choosing an inappropriate node to split. However, at 15,000 games, we see another sudden change, indicating *TBCL* found a node to split that would help. This suggests that further study in selecting a node for splitting would be appropriate and beneficial.

5.2.4. Pursuit with limited mobility. Finally, we apply *TBCL* to the pursuit game with limited mobility. This game is identical to the game described in Section 4.2.4. Again, we removed the boundary, but we also limited the mobility of both players to permit them to make instantaneous turns within a constrained range of possible turns.

We do make one change in the experiments. Specifically, we only permit training to take place through 20,000 games. We test after 250 games, as before, but this time we also split after 1000 games. This change was justified by the fact there was little evidence of change between splits. Consequently, we reduced the number of games played before splitting. We still wanted to monitor the progress between splits, so we increased the frequency of testing accordingly.

The results of these experiments were both encouraging and surprising. The performance learning curves are shown in figure 15. This graph was produced by running the experiment ten times and averaging the results at 1000 game intervals. First, it was clear that E’s random strategy was not satisfactory against a heuristic P, and E learned a strategy that became competitive (although still somewhat inferior). Improvement by P relative to a heuristic opponent was similar.

The surprising result concerns comparing the curve for E versus P to the curve for heuristic E versus heuristic P. As we see, heuristic performance is fairly constant throughout (which it should be). The performance of E versus P; however, was not constant. Of course, this is what we would like, except that we found the performance to *diverge* from the heuristic. Although heuristic performance appears to yield an average payoff of approximately 1.9, we found the average payoff for E versus P to drop from around 1.7 (at 10,000 games) to about 1.5 (at 20,000) games.

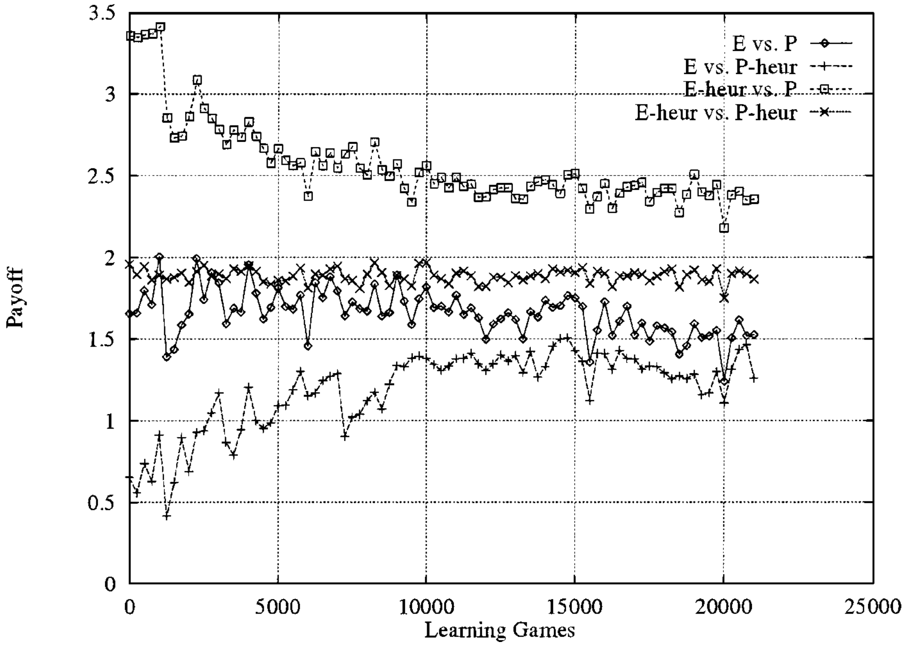

Figure 15. Learning performance for pursuit game with limited mobility.

We believe that the reason for this unpredicted behavior is associated to the algorithm for selecting a node to split. Evasion depends on E getting inside P's radius of curvature. According to Basar and Olsder (Basar & Olsder, 1982), this requirement is further complicated by the fact that the decision surface characterizing the optimal solution of the Homicidal Chauffeur game has a "leaky corner." A leaky corner is a characteristic in the surface between terminal conditions of the game in which performance cannot be forced by either player. The leaky corner is even more problematic in our game in which *both* P and E have limited mobility (rather than just P).

To be able to learn this surface may require many more splits in the decision tree. Further, because our method for selecting a node to split is biased towards nodes that are frequently updated, and we hope that we are in the region with the leaky corner relatively infrequently, our method is probably not well suited for learning these characteristics. Thus, a substantially larger number of node splits may be required with our method.

5.3. *Discussion*

We were pleased with the results of *TBCL*, especially when compared to the performance of *MBCL*. First, we found the overall learning performance to be quite good. In fact, we feel the performance was clearly "comparable" to the performance of *MBCL*—even with several known deficiencies in *TBCL*. Further, as the games became more complex, *Table 2*. Relative computational burdens for solving games with *TBCL*.

Game	Games stored	Leaves	Minutes
Force	100,000	1	31
Simple	100,000	20	49
Half	100,000	20	43
Limit	20,000	20	22

the experimental results seemed to indicate that the tree-based approach could ultimately adapt better to the underlying state-action space; however, we did not try any variable-resolution memory-based strategies to compare. It is possible that the advantages of the tree-based strategy can be attributed to the variable resolution, which can be replicated in memory-based approaches.

In addition to the comparable performance of *TBCL* relative to *MBCL*, we also found a substantial improvement in computational and memory burden. As before, all of the experiments for *TBCL* were run on either Sun Sparc 2 or Sun Sparc 10 processors. To give an idea of the improvement in computational burden, Table 2 shows the time required for playing the games (in the best case) and the size of the associated trees (in number of leaves—because all of the trees are binary, we know the number of nodes in the tree is twice the number of leaves minus 1).

As we see, the times required to learn these games were substantially less than the times required for *MBCL* (Table 1).[1] Given a more clever approach to splitting nodes, we may have been able to yield even stronger performance with the same, or possibly fewer, numbers of nodes. Further, we could have increased the search space by providing a finer resolution on the strategy space and still been able to learn in a reasonable period of time.

If we perform a complexity analysis similar to the one performed for *MBCL*, we find *TBCL* has several advantages. First, if we assume the tree will grow to some maximum size T (given in terms of the number of leaves of the tree), we find the following. First, linear programming does not need to be performed during play since it is only done after learning. Assuming the decision tree can be constructed to maintain reasonable balance, time to traverse the tree at any given step in the game should be $O(\log T)$. As with *MBCL*, selecting a move is $O(m)$, and updating the state is constant. Thus, play is very efficient.

The tradeoff comes when considering learning. In *MBCL*, learning consisted of updating the Q values associated with known examples in the memory base. This is no more than $O(k_s)$. Every time a matrix changes, a linear program needs to be solved. Thus, the computational burden associated with linear programming is transferred from the on-line evaluation and play of the game to learning. Further, *TBCL* periodically splits nodes in the tree which involves evaluating potential splits and then performing a final split. The evaluation step for a single split requires $O(m^2 p + m^2)$ and the actual split requires $O(dm^2 p)$. Since T splits occur as the tree is generated, the total time to learn is $O(Tdm^2 p)$.

In all of the experiments, the sizes of the trees were limited such that these trees would be significantly smaller than a *kd*-tree for *MBCL*. If the trees were made comparable in size, it is likely the total time required for the two algorithms (including play and learning) would be comparable. However, the results of our experiments seem to indicate that small trees are sufficient to learn fairly complex tasks. This implies small memory-bases may also be effective, given the right set of examples. Note this is consistent with experiments reported in (Sheppard & Salzberg, 1997).

One possible limitation in both *MBCL* and *TBCL* is the reliance on a common set of Q-values. While this is a valid concern in the context of *independent* agents learning to play against each other, it is clear that the algorithms apply directly when separate experience bases are maintained. Our research, as noted earlier, did not maintain separate experiences because our primary goal was developing an approach to learn "optimal" solutions to the games. We believe convergence to optimal with separate experience would have occured, but a much slower rate.

6. Future work

Because the games reported in this paper are limited to two dimensions, work exploring games of higher dimensionality (e.g., x, y, z) is necessary. Further, in many ways, the games described do not correspond to similar games in the real world; therefore, games characterizing more realistic capabilities (e.g., noisy sensors, and imperfect controllers) should be encouraged.

We note that all the algorithms discussed in this paper are limited to symbolic reasoning systems. Specifically lacking are any algorithms derived from the connectionist (i.e., neural network) community, despite the fact that much of the successful research in reinforcement learning has been applied to connectionist systems, with symbolic systems largely limited to lookup tables. Additional work that applies the ideas in this paper to connectionist systems would be warranted. Of particular interest would be work integrating connectionist and symbolic systems into a cohesive multiagent learner. For example, an interesting architecture might include an artificial neural network to provide a fitness function for filling out a game matrix. Harmon and Baird's (1995) approach using a neural network could easily be extended to include a full evaluation of the linear program and its dual. Given this extension, comparable performance might be achieved from their approach.

Several variations of *MBCL* should be considered. For example, limited seeding followed by variable resolution memory-based learning would provide a potential solution to the problem of appropriate sampling and excessive memory-base size. In addition, using a data structure such as the *kd*-tree to store the memory-base would significant speed up learning and testing.

Throughout this paper, no concerted effort was made to identify "optimal" parameters for the learning algorithms. Both *MBCL* and *TBCL* have a relatively large number of parameters that need to be set. Evaluating the effects of various parameter settings, for example through a factorial study, would provide considerable insight into the power of the algorithms and their ability to find reasonable parameters in other games.

As with *MBCL*, several variations on *TBCL* could be explored. For example, we note that the method for selecting a node to split is naive at best. It may be possible to apply selection techniques such as those used in traditional classification decision trees to characterize potential improvement (e.g., entropy reduction, minimum description length, and minority measures). As an alternative, it may be worthwhile to explore techniques of k-step look-ahead to evaluate a node for splitting. Under such a method, a small number of splits are selected and evaluated. The best split is then selected, and learning continues from that point.

Related to the problem of selecting a node for splitting is the problem of selecting an attribute and associated value for splitting. In our experiments, we selected the attribute that maximized the difference between resultant submatrices. Again, principles such as entropy reduction or minimum description length may be appropriate.

In all cases, we assumed that the split value would be the midpoint of the region. Again, this may not be appropriate. Because we are not splitting examples, we cannot select regions between neighboring examples; however, a similar quantization of the attribute space may be appropriate.

Finally, it may be appropriate to consider nonaxis-parallel trees in growing trees for *TBCL*. Work by Heath and Murthy has pointed out several issues and offered several suggestions for constructing oblique decision trees and addressing concerns such as look-ahead and splitting criteria (Heath, 1992; Murthy, 1995).

One of the significant differences between *MBCL* and *TBCL* is that Q-updates in *MBCL* occur over a region in the instance space where Q-updates in *TBCL* apply only to individual cells in the game matrix. An interesting variation to *TBCL* would apply a weighted update, such as

$$Q(s, a_p, a_e) = (1 - w\alpha_i)Q(s, a_p, a_e) + w\alpha_i[\rho + \gamma Q(s', \pi(s'))]$$

where w is a weight based on proximity to the target cell. Thus, cells in a region around the target cell to be updated could also be updated. This approach is motivated by the fact that the game matrix for a differential game would, frequently, be fit by a smooth surface.

7. Summary

In this paper, we provided a new algorithm for memory-based colearning in which two opposing agents learn control strategies simultaneously. These results (and the results of *TBCL*) can be extended to alternating Markov games (in which players take turns) (Littman, 1996), team games (in which teams of players cooperate to devise mutual strategies) (Tambe, 1996a, 1996b), and community games (in which players choose opponents to maximize their personal payoff) (Stanley, Ashlock, & Tesfatsion, 1993). They can also be applied to games that are more traditional with homogeneous agents such as backgammon, checkers, and othello. The strengths of the approach include the relative simplicity in storing examples and updating value estimates for game play. Unfortunately, the approach is both memory and computation intensive.

We also provided a second novel algorithm for colearning based on dynamically partitioning the state space of the game. The focus of the approach was on reducing memory and computation requirements while maintaining or improving upon the performance obtained by *MBCL*. The resulting algorithm, *TBCL*, accomplished these goals by not requiring explicit storage of examples in a memory base and by keeping game matrices at each of the leaves of the tree. Examples were replaced by state space partitions covering a region of the space. Since the regions can be partitioned to any required resolution, this approach can maintain the level of performance of the memory-based approach without explicitly storing examples.

In addition, since game matrices are kept with each partition, several computational advantages are obtained. First, the game matrix does not need to be regenerated at every step of the game. Second, if multiple steps in the game take place within the same partition of the state space, only one linear program needs to be solved (rather than one for each visit to the partition). Finally, following learning, the game matrix can be thrown away (thus further reducing memory requirements), and the current strategies stored with the partition used for play—there is no need to solve *any* linear programs during actual use. The result is an approach to colearning that is faster than memory-based learning during both training and actual use.

Acknowledgments

Thanks go to Steven Salzberg, Simon Kasif, and Fernando Pineda for many ideas and comments on the research described. In addition, thanks are due to Scott Weiss and Sreerama Murthy for many useful discussions while developing and testing the algorithms. This research has been supported, in part, by the National Science Foundation under Grant Nos. IRI-9116843 and IRI-9223591 and through the educational assistance program at ARINC.

Note

1. Clearly, using *kd*-trees or similar data structures in *MBCL* would reduce this difference.

References

Aha, D. (1992). Tolerating noise, irrelevant, and novel attributes in instance-based learning algorithms. *International Journal of Man-Machine Studies, 16,* 267–287.

Anderson, C. & Crawford-Hines, S. (1994). *Multigrid Q-learning*. (Technical Report CS-94-121), Department of Computer Science, Colorado State University.

Barto, A., Bradtke, S., & Singh, S. (1993). Learning to act using real-time dynamic programming. *Artificial Intelligence*.

Barto, A., Sutton, R., & Watkins, C. (1990). Learning and sequential decision making. In Gabriel & Moore (Eds.), *Learning and computational neuroscience* (pp. 539–602). Cambridge, MA: MIT Press.

Basar, T. & Olsder, G. (1982). *Dynamic noncooperativeg game theory*. London: Academic Press.

Bentley, J. (1980). Multidimensional divide and conquer. *Communications of the ACM, 23*(4), 214–229.

Bertsekas, D. (1987). *Dynamic programming: Deterministic and stochastic models*. Prentice-Hall, Inc.

Collins, R. (1992). *Studies in artificial evolution*. Ph.D. thesis, Department of Computer Science, University of California at Los Angeles, Los Angeles, CA.

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning, 8,* 341–362.

Deng, K. & Moore, A. (1995). Multiresolution instance-based learning. *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, 26*(1), 1–13.

Erev, I. & Roth, A. (1995). On the need for low rationality, cognitive game theory: Reinforcement learning in experimental games with unique mixed strategy equilibria. Unpublished manuscript, August.

Grefenstette, J. (1988). Credit assignment in rule discovery systems based on genetic algorithms. *Machine Learning, 3*, 225–245.

Grefenstette, J. (1991). Lamarkian learning in multi-agent environments. *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 303–310). Morgan Kaufman.

Grefenstette, J. & Daley, R. (1995). Methods for competitive and cooperative coevolution. *Adaptation, Coevolution, and Learning in Multiagent Systems (ICMAS '95)*, (pp. 276–282), AAAI Press.

Grefenstette, J., Ramsey, C., & Schultz, A. (1990). Learning sequential decision rules using simulation models and competition. *Machine Learning, 5,* 355–381.

Harmon, M. & Baird, L. (1995). Residual advantage learning applied to a differential game. *Neural Information Processing Systems*, *7*.

Harmon, M., Baird, L., & Klopf, A. (1995). Reinforcement learning applied to a differential game. *Adaptive Behavior*.

Heath, D. (1992). *A geometric framework for machine learning*. Ph.D. thesis, Department of Computer Science, The Johns Hopkins University, Baltimore, MD.

Huberman, B. & Glance, N. (1995). Evolutionary games and computer simulations. *Proceedings of the National Academy of Sciences*.

Isaacs, R. (1975). *Differential Games*. New York, NY: Robert E. Krieger.

Lewin, J. (1994). *Differential Games*. New York, NY: Springer-Verlag.

Littman, M. (1994). Markov games as a framework for multi-agent reinforcement learning. *Proceedings of the Eleventh International Machine Learning Conference* (pp. 157–163). New Brunswick, NJ: Morgan Kaufmann.

Littman, M. (1996). *Algorithms for sequential decision making*. Ph.D. thesis, Department of Computer Science, Brown University.

Moore, A. (1990). *Efficient memory-based learning for robot ccntrol*. Ph.D. thesis, Computer Laboratory, Cambridge University.

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

Murthy, S. (1995). *On growing better decision trees from data*. Ph.D. thesis, Department of Computer Science, The Johns Hopkins University, Baltimore, MD.

Potter M., De Jong, K., & Grefenstette, J. (1995). A coevolutionary approach to learning sequential decision rules. *Proceedings of the International Conference on Genetic Algorithms* (pp. 366–372).

Rajan, N., Prasad, U., & Rao, N. (1980). Pursuit-evasion of two aircraft in a horizontal plane. *Journal of Guidance and Control*, *3*(3), 261–267.

Rosenschein, J. & Genesereth, M. (1985). Deals among rational agents. *Proceedings of the 1985 International Joint Conference on Artificial Intelligence* (pp. 91–99).

Roth, A. & Erev, I. (1995). Learning in extensive-form games: Experimental data and simple dynamic models in the intermediate term. *Games and Economic Behavior, 8*, 164–212.

Salzberg, S. (1991). Distance metrics for instance-based learning. *Methodologies for Intelligence Systems: 6th International Symposium* (pp. 399–408).

Samuel, A. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *3*(3), 211–229.

Sandholm, T. & Crites, R. (1995). Multiagent reinforcement learning in the iterated prisoner's dilemma. *Biosystems, 37*, 147–166.

Schmidhuber, J. (1996). A general method for incremental self-improvement and multi-agent learning in unrestricted environments. *Evolutionary Computation: Theory and Applications*.

Sheppard, J. (1996). *Multi-agent reinforcement learning in Markov games*. Ph.D. thesis, Department of Computer Science, The Johns Hopkins University, Baltimore, MD.

Sheppard, J. & Salzberg, S. (1997). A teaching method for memory-based control. *Artificial Intelligence Review, 11*, 343–370.

Simons, J., van Brussel, H., DeSchutter, J., & Verhaert, J. (1982). A self-learning automaton with variable resolution for high precision assembly by industrial robots. *IEEE Transactions on Automatic Control*, *27*(5), 1109–1113.

Smith, R. & Gray, B. (1993). *Co-adaptive genetic algorithms: An example in Othello strategy*. (Technical Report TCGA Report No. 94002), University of Alabama, Tuscaloosa, Alabama.

Stanley, E., Ashlock, D., & Tesfatsion, L. (1993). Iterated prisoner's dilemma with choice and refusal of partners. *Proceedings of Alife III*. Sante Fe Institute.

Suguwara, T. & Lesser, V. (1993). *On-line learning of coordination plans*. (Technical Report COINS TR 93-27), Amherts, MA: University of Massachusetts.

Sutton, R. (1988). Learning to predict by methods of temporal differences. *Machine Learning, 3,* 9–44.

Stone, P. & Veloso, M. (1995). Beating a defender in robotic soccer: Memory-based learning of a continuous function. *Proceedings of Neural Information Processing Systems*.

Stone, P. & Veloso, M. (1996a). Multiagent systems: A survey from a machine learning perspective. *IEEE Transactions on Knowledge and Data Engineering*, submitted.

Stone, P. & Veloso, M. (1996b). Towards collaborative and adversarial learning: A case study in robotic soccer. *AAAI Spring Symposium on Adaptation, Co-Evolution, and Learning in Multiagent Systems*.

Tambe, M. (1996a). Teamwork in real-world, dynamic environments. *International Conference on Multiagent Systems*, AAAI Press.

Tambe, M. (1996b). Tracking dynamic team activity. *Proceedings of the 13th National Conference on Artificial Intelligence*. AAAI Press.

Tan, M. (1993). Multi-agent reinforcement learning: Independent vs. cooperative agents. *Machine Learning: Proceedings of the Tenth International Conference*, San Mateo, CA: Morgan Kaufmann.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning, 8,* 257–277.

Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Communications of the ACM* (pp. 58–67).

Watkins, C. (1989). *Learning with delayed rewards*. Ph.D. thesis, Department of Computer Science, Cambridge University, Cambridge, England.

Watkins, C. & Dayan, P. (1992). *Q*-learning. *Machine Learning, 8,* 279–292.

Received January 31, 1997
Accepted March 14, 1998
Final manuscript January 27, 1998