Artificial Intelligence Review **11:** 343–370, 1997.

© 1997 *Kluwer Academic Publishers. Printed in the Netherlands.*

A Teaching Strategy for Memory-Based Control

JOHN W. SHEPPARD and STEVEN L. SALZBERG
Department of Computer Science, The Johns Hopkins University, Baltimore, Maryland 21218
E-mail: lastname@cs.jhu.edu

Abstract. Combining different machine learning algorithms in the same system can produce benefits above and beyond what either method could achieve alone. This paper demonstrates that genetic algorithms can be used in conjunction with lazy learning to solve examples of a difficult class of delayed reinforcement learning problems better than either method alone. This class, the class of differential games, includes numerous important control problems that arise in robotics, planning, game playing, and other areas, and solutions for differential games suggest solution strategies for the general class of planning and control problems. We conducted a series of experiments applying three learning approaches – lazy Q-learning, k-nearest neighbor (k-NN), and a genetic algorithm – to a particular differential game called a pursuit game. Our experiments demonstrate that k-NN had great difficulty solving the problem, while a lazy version of Q-learning performed moderately well and the genetic algorithm performed even better. These results motivated the next step in the experiments, where we hypothesized k-NN was having difficulty because it did not have good examples – a common source of difficulty for lazy learning. Therefore, we used the genetic algorithm as a bootstrapping method for k-NN to create a system to provide these examples. Our experiments demonstrate that the resulting joint system learned to solve the pursuit games with a high degree of accuracy – outperforming either method alone – and with relatively small memory requirements.

Key words: lazy learning, nearest neighbor, genetic algorithms, differential games, pursuit games, teaching, reinforcement learning

1. Introduction

When two people learn a task together, they can both benefit from the different skills that each brings to the table. The result is that both will learn better than they would have on their own. Likewise, machine learning methods should be able to work together to learn how to solve difficult problems. This paper describes how a lazy learning algorithm and a genetic algorithm can work together to produce better solutions than either method could produce by itself.

To explore our hypothesis that two learning algorithms can work together to outperform either individually, we focused on a particular problem in which an agent must perform a task, and the task requires several steps to

accomplish. We limit feedback on how well the agent is performing to the end of the task. Several learning algorithms have been applied to this family of problems, called *delayed reinforcement problems* (Widrow 1987; Atkeson 1990; Watkins 1989; Barto, Sutton and Watkins 1990; Millan and Torras 1992; Moore and Atkeson 1993), but little has been done to evaluate the power of combining different types of learning algorithms to these problems.

One way of characterizing these delayed reinforcement problems is as learning to solve a *Markov decision problem* (van der Wal 1981). Markov decision problems are those in which an agent develops a mapping from a set of states to a set of actions, possibly different ones for each state, and the optimal strategy from a given state depends only on the current state. The actions are directed toward achieving some goal or performing some task, and payoff or penalty for that action is awarded immediately. Delayed reinforcement problems apply zero payoff at intermediate states and apply the actual reward at the end of the sequence.

One class of problems frequently modeled as a Markovian decision problem is the class of differential games. Differential games require the players to make a long sequence of moves where the behaviors and strategies are modeled by differential equations. Finding a solution to the differential game consists of computing the "value" of the game in terms of expected payoff and determining the optimal strategies for the players that yield this value. Differential games are difficult to solve yet are important for solving a wide variety of multi-agent tasks. They have had widespread application in the military and entertainment industries, but more recently, systems for intelligent highways, air traffic control, railroad monitoring, and ship routing are using differential game theory to assist agents in optimizing their often competing goals. More generally, strategies for solving these games can be used for planning and intelligent agents, thus making the approach discussed applicable to the broader domain of control problems.

For this study, we begin by considering a differential game that involved one agent trying to pursue and capture another (i.e., a pursuit game). Earlier research showed that at least one implementation of this task, known as *evasive maneuvers* (Grefenstette, Ramsey and Schultz 1990), can be solved by a genetic algorithm (GA). We developed a lazy learning approach using k-nearest neighbor (k-NN) for the same task, hoping to demonstrate lazy learning could perform as well or better than the GA. Then we made the task substantially harder to study the limitations of lazy learning methods on this class of problems. The more complicated task, which is described further in Section 3.2, also resembles complicated planning tasks in which an agent has to satisfy several goals simultaneously (Chapman 1987).

As our experiments will show, we were successful at developing a method to solve our difficult reinforcement learning task. The key idea behind our success was the combined use of both lazy learning and GAs. We observed after comparing two lazy methods (k-NN and an adaptation of Q-learning) with genetic algorithms that lazy methods can learn to solve the task, but were dependent on having good examples in the database. Later, we found that the best learning agent first used a GA to generate examples, and then switched to k-NN after reaching a certain performance threshold. Our experiments demonstrate significant improvement in the performance of lazy learning, both in overall accuracy and in memory requirements, as a result of using these techniques. The combined system also performed better than the GA alone, demonstrating how two learning algorithms working together can outperform either method when used alone.

2. Previous Work

Recently, considerable work has been done applying learning algorithms to Markov decision problems. To date, little has been done to apply these algorithms to differential games. One exception to this is Grefenstette's SAMUEL system, which uses a genetic algorithm. In addition to the evasive maneuvers task, Grefenstette (1991) has applied SAMUEL to aerial dogfighting and target tracking. Ramsey and Grefenstette (1994) have used a case-based method of initializing SAMUEL with a population of "solutions" dependent on the current environment. Where we use a GA to "jump start" a lazy learner, Ramsey and Grefenstette use a lazy learner to jump start a GA. This suggests that a combined strategy where the lazy learner and the GA transmit information in both directions could be a powerful combination.

In related research, Gordon and Subramanian (1993a, 1993b) use an approach similar to explanation based learning (EBL) to incorporate advice into a genetic algorithm, using SAMUEL for the GA. In their multistrategy apprach, a *spatial knowledge base* and high-level strategic guidance from a human teacher are encoded using rule compilation that operationalizes the rules, by encoding them in a form suitable for SAMUEL to use. SAMUEL then uses and refines the advice with its genetic algorithm.

The idea of using lazy learning methods for delayed reinforcement tasks has only recently been studied by a small number of researchers. Atkeson (1990) employed a lazy technique to train a robot arm to follow a prespecified trajectory. Moore (1990) took advantage of the improved efficiency provided by storing examples in kd-trees in using a lazy approach to learn several robot control tasks. More recently, Moore and Atkeson (1993) developed

their prioritized sweeping algorithm, in which “interesting” examples in a Q table are the focus of updating.

McCallum (1995) developed the “nearest sequence memory” algorithm, which is a lazy algorithm for solving control problems plagued by hidden state. Hidden state is an artifact of *perceptual aliasing* in which the mapping between states and perceptions is not one-to-one (Whitehead 1992). McCallum showed through his algorithm that lazy methods can reduce the effects of perceptual aliasing by appending history information with state information. Since our approach stores complete sequences, we too have minimized the effects of hidden state.

In another study, Aha and Salzberg (1993) used nearest-neighbor techniques to train a simulated robot to catch a ball. In their study, they provided an agent that knew the correct behavior for the robot, and therefore provided corrected actions when the robot made a mistake. This approach is typical in nearest-neighbor applications that rely on determining “good” actions before storing examples. In our case, we had no idea which examples were good and needed an approach to determine these examples.

One of the most popular approaches to reinforcement learning has been using neural network learning algorithms, most often the error back-propagation algorithm. This has been used for simple multi-step control problems (Widrow 1987; Nguyen and Widrow, 1989), using knowledge of the correct control action to train the network. Millan and Torras (1992) used a reinforcement learning algorithm embedded in a neural net in which the control variables were permitted to vary continuously. They addressed the problem of teaching a robot to navigate around obstacles.

Considerable research has been performed using a form of reinforcement learning called *temporal difference* learning (Sutton 1988). Temporal difference methods apply reinforcement throughout a sequence of actions to predict both future reinforcement and appropriate actions in performing the task. Specifically, predictions are refined through a process of identifying differences between the results of temporally successive actions. Two popular temporal difference algorithms are ACE/ASE (Barto, Sutton and Anderson 1983; Barto et al. 1990) and Q-learning (Watkins 1989). The original work by Barto et al. (1983) demonstrated that the cart and pole problem could be solved using this method. Clouse and Utgoff (1992) later used ACE/ASE with a separate teacher for the cart and pole problem, and applied Q-learning to navigating a race track. Lin (1991) used Q-learning to teach a robot to navigate the halls of a classroom building and plug itself into a wall socket to recharge its batteries. Below we describe a lazy variant of Q-learning, and show that it is also capable of learning complex control tasks.

In addition, Dorigo and Colombetti (1994) and Colombetti and Dorigo (1994) describe an approach to using reinforcement learning in classifier systems to teach a robot to approach and pursue a target. Their approach uses a separate reinforcement program to monitor the performance of the robot and provide feedback on performance. Learning occurs through a genetic algorithm applied to the classifiers with fitness determined by the reinforcement program.

More recently, Michael Littman (1994) observed that reinforcement learning can be applied to multi-agent activities in the context of *Markov games*. Littman expanded Watkins' Q-learning algorithm to cover two players in a simplified game of soccer. He embedded linear programming to determine the optimal strategy prior to play, and applied a modified Q backup operator (which accounted for the competitive goals of the players) to update the estimates of the expected discounted reward for each player.

Some other recent work in learning strategies for game playing has begun to deal with issues of co-learning at a superficial level with strategic games such as chess and othello. Pell (1993) developed an environment for deriving strategies in what he calls "symmetric chess-like games." His METAGAMER focused on translating the rules and constraints of a game into a strategy using a declarative formulation of the game's characteristics. METAGAMER has been applied to chess, checkers, noughts and crosses (i.e., Tic-Tac-Toe), and Go, and has yielded performance at an intermediate level for each of these games. Smith and Gray (1993) applied what they call a *co-adaptive* genetic algorithm to learn to play Othello. A co-adaptive GA is a genetic algorithm in which fitness values of members of the population are dependent on the fitness of other members in the population. They found they were able to control the development of niches in the population to handle several different types of opponents. Finally, Tesauro used temporal difference learning (Tesauro 1992) and neural networks (Tesauro and Sejnowski 1989) to train a backgammon program called TD-GAMMON. Backgammon's stochastic component (each move is determined in part by a roll of dice) distinguishes it from deterministic games such as chess, but despite this additional complexity, TD-GAMMON is currently playing at a master level.

3. The Problem

Reinforcement learning (RL) is challenging in part because of the delay between taking an action and receiving a reward or penalty. Typically an agent takes a long series of actions before the reward, so it is hard to decide which of the actions were responsible for the eventual payoff. Both lazy and eager approaches to reinforcement learning can be found in the literature. The

most common eager approach is the use of temporal-difference learning on neural networks (Barto et al. 1983, 1990; Clouse and Utgoff 1992; Tesauro 1992). The advantages to a lazy approach are three-fold. First, minimal computational time is required during training, since training consists primarily of storing examples (in the most traditional lazy approach, k-nearest neighbor). Second, lazy methods have been shown to be good function-approximators in continuous state and action spaces (Atkeson 1992). As we will see, this will become important for our task of learning to play differential games. Third, traditional eager approaches to reinforcement learning assume the tasks are Markov decision problems. When the tasks are non-Markovian (e.g., when history is significant), information must be appended to the state to encapsulate some of the prior state information, in order to approximate a Markov decision problem. Since the lazy approach stores complete sequences, non-Markovian problems can be treated in a similar fashion to Markovian problems.

The class of RL problems studied here has also been studied in the field of *differential game theory*. Differential game theory is an extension of traditional game theory in which a game follows a sequence of actions through a continuous state space to achieve some payoff (Isaacs 1963). This sequence can be modeled with a set of differential equations which are analyzed to determine optimal play by the players. We can also interpret differential games to be a version of optimal control theory in which players' positions develop continuously in time, and where the goal is to optimize competing control laws for the players (Friedman 1971).

3.1. *Differential games and pursuit games*

Differential game theory originated in the early 1960s (Isaacs 1963) as a framework for a more formal analysis of competitive games. In a differential game, the dynamics of the game (i.e., the behaviors of the players) are modeled with a system of first order differential equations of the form

$$\frac{dk_j^t}{dt} = h_j^t(k^t, a^t), j = 1, ..., n \quad (1)$$

where $a^t = (a_1^t, ..., a_p^t)$ is the set of actions taken by p players at time t, $k^t = (k_1^t, ..., k_n^t)$ is a vector in real Euclidean n-space denoting a position in play (i.e., the state) for the game, and $h_j^t()$ is the history of the game for the jth dimension of the state space. In other words, the differential equations model how actions taken by the players in the game change the state of the game over time. In these games, the initial state of the game k^0 is given. The object of analyzing a differential game is to determine the optimal strategies

for each player of the game and to determine the value of the game (i.e., the expected payoff to each player) assuming all of the players follow the optimal strategies. For more details, see (Sheppard and Salzberg 1993).

A pursuit game is a special type of differential game that has two players, called the pursuer (P) and the evader (E). The evader attempts to achieve an objective, frequently to escape from a fixed playing arena, while the pursuer attempts to prevent the evader from achieving that objective. Examples include such simple games as the children's game called "tag," the popular video game PacMan, and much more complicated predator-prey interactions in nature. These examples illustrate a common feature of pursuit games – the pursuer and the evader have different abilities: different speeds, different defense mechanisms, and different sensing abilities.

One classic pursuit game studied in differential game theory is the *homicidal chauffeur* game. In this game, we can think of the playing field being an open parking lot with a single pedestrian crossing the parking lot and a single car. The driver of the car (the chauffeur) is trying to run down the pedestrian. Although the car is much faster than the pedestrian, the pedestrian can change direction much more quickly than the car. The typical formulation has both the car and the pedestrian traveling at fixed speeds, with the car having a fixed minimum radius of curvature, and the pedestrian able to make arbitrarily sharp turns (i.e., the radius of curvature is zero) (Basar and Olsder 1982).

In analyzing this game, it turns out that the solution is relatively simple and depends on four parameters – the speed of the car, the speed of the pedestrian, the radius of curvature of the car and the "lethal envelope" of the car (i.e., the distance between the car and the pedestrian that is considered to be "close enough" to hit the pedestrian). Isaacs (1963) shows that, assuming optimal play by both players, the ability of P to capture E (or conversely for E to escape) depends on the ratio of the players' speeds and P's radius of curvature. Intuitively, optimal play for P is to turn randomly if lined up with E and to turn sharply toward E otherwise. The optimal strategy for E is to head directly towards P until inside P's radius of curvature, and then to turn sharply. Since E's strategy is the more interesting, we will focus on learning to evade in a similar game.

3.2. *The evasive maneuvers task*

The evasive maneuvers task as a differential game is a variation on the homicidal chauffeur game. Even though the solution to the homicidal chauffeur game is intuitive, the actual surface characterizing the solution is highly nonlinear. Thus we should reasonably expect the surface for extensions to the problem (such as those discussed in this paper) to be more difficult to characterize. Grefenstette et al. (1990) studied the evasive maneuvers task to demonstrate

the ability of genetic algorithms to solve complex sequential decision making problems. In their two-dimensional simulation, a single aircraft attempts to evade a single missile.

We initially implemented the same pursuit game as Grefenstette et al., and later we extended it to make it substantially more difficult. In this game, play occurs in a relative coordinate system centered on the evader, E. Because of the relative frame of reference, the search space is reduced and games are determined by their starting positions. P uses a fixed control law to attempt to capture E, while E must learn to evade P. Even the basic game is more difficult than the homicidal chauffeur game, because the pursuer has variable speed and the evader has a non-zero radius of curvature. Our extended version includes a second pursuer, which makes the problem much harder. Unlike the single-pursuer problems, the two-pursuer problem has no known optimal strategy (Imado and Ishihara 1993), and for some initial states, there is no possibility of escape. Second, we gave the evader additional capabilities: in the one-pursuer game, E only controls its turn angle at each time step. Thus E basically zigzags back and forth or makes a series of sharp turns into the path of P to escape. In the two-pursuer game, we gave E the ability to change its speed, and we also gave E a bag of "smoke bombs," which will for a limited time help to hide E from the pursuers.

In our definition of the two-pursuer task, both pursuers ($P1$ and $P2$) have identical maneuvering and sensing abilities. Further, they use the same control strategy: they anticipate the future location of E and aim for a location where they can capture in the fewest time steps. They begin the game at random locations on a fixed-radius circle centered on the evader, E. The initial speeds of $P1$ and $P2$ are much greater than the speed of E, but they lose speed as they maneuver, in direct proportion to the sharpness of the turns they make. The maximum speed reduction is 70%, scaled linearly from no turn (with no reduction in speed) to the maximum turn angle allowed of 135°. They can regain speed by traveling straight ahead, but they have limited fuel. If the speed of both $P1$ and $P2$ drops below a minimum threshold, then E escapes and wins the game. E also wins by successfully evading the pursuers for 20 times steps (i.e., both $P1$ and $P2$ run out of fuel). If the paths of either $P1$ or $P2$ ever pass within a threshold range of E's path during the game, then E loses (i.e., the pursuer will "grab" E) (Figure 1). We use the term "game" to include a complete simulation run, beginning with the initial placements of all of the players, and ending when E either wins or loses, at most 20 time steps later.

When playing against one pursuer, the capabilities of E are identical to the simulated aircraft used by Grefenstette et al. Against one pursuer, E controls only its turn angle, which is sufficient to play the game well. With

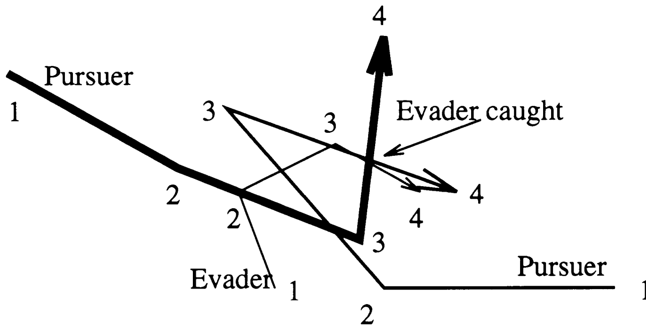

Figure 1. A game where E is caught.

two pursuers $P1$ and $P2$ in the game, E has additional information about its opponents. This information includes 13 features describing the state of the game, including E's own speed, the angle of its previous turn, a game clock, the angle defined by $P1$-E-$P2$, and the range difference between $P1$ and $P2$. It also has eight features that measure $P1$ and $P2$ individually: speed, bearing, heading, and distance. Bearing measures the position of the pursuer relative to the direction that E is facing (e.g., if E is facing north and $P1$ is due east, then the bearing would be 3 o'clock). Heading is the angle between E's direction and the pursuer's direction. When fleeing two pursuers, E can adjust its speed and turn angle at each time step, and it can also periodically release a smoke bomb, which introduces noise into the sensor readings of $P1$ and $P2$. If smoke is released, the turn angle of the pursuer is shifted by a random factor up to 50% of the current turn angle. As the severity of the turn increases, so does the potential effect from smoke.

4. The Learning Algorithms

The following sections discuss the details of the experiments with the three learning algorithms and motivate the need for a learning strategy combining eager learning (as a teacher) and lazy learning (as a performer). We explored several algorithms to determine the applicability of lazy learning to control problems in general, and pursuit games in particular. We began by examining the ability of Q-learning to learn to play the evasive maneuvers game. We had to adapt Q-learning because of the large, continuous state space, which resulted in a lazy variant of standard Q-learning. We then tried a traditional lazy learning approach, k-nearest neighbors. Finally, we experimented with an eager learning method, genetic algorithms, to compare with the two lazy methods.

4.1. *Lazy Q-learning for evasive maneuvers*

Q-learning solves delayed reinformement learning problems by using a temporal difference (TD) learning rule (Watkins 1989). TD methods usually assume that both the feature space and the variables being predicted are discrete (Sutton 1988; Tesauro 1992). Q-learning represents a problem using a lookup table that contains all states, which naturally causes problems with large, continuous state spaces such as those encountered in differential games. We therefore had to develop a method for predicting the rewards for some state-action pairs without explicitly generating them. The resulting algorithm was a lazy version of Q-learning.

Rather than constructing a complete lookup table, our implementation of Q-learning stores examples similar to the set of instances produced for a lazy method such as k-NN. It begins by generating a set of actions at random for a particular game; these actions do not have to result in successful evasion. Instead, the algorithm applies a payoff function (defined below) to determine the reward for that sequence of state-action pairs. Initially, it stores the actual payoff values with these pairs. After generating the first set of pairs, learning proceeds as follows.

First, assuming that neighboring states will require similar actions, we specify two distance parameters: one for the states and one for the actions ($d_1 = 0.01$ and $d_2 = 0.005$ respectively), noting that all distances are normalized. The purpose of these parameters is to guide a search through the instance database. The system begins an evasive maneuvering game by initializing the simulator. The simulator passes the first state to the state matcher which locates all of the states in the database that are within d_1 of the current state. If the state matcher has failed to find any nearby states, the action comparator selects an action at random. Otherwise, the action comparator examines the expected rewards associated with each of these states and selects the action with the highest expected reward. The resulting action is passed to the simulator, and the game continues until termination. It also has a probability (0.3) of generating a random action regardless of what it finds in the table. This permits it to fill in more of the database; i.e., it is exploring the state space as it is learning. It passes the resulting action to the simulator, and the game continues until termination, at which point the simulator determines the payoff. The Q function then updates the database using the complete game.

At the end of a game, the system examines all of the state-action pairs in the game. It stores in the database any state-action pair that is new, along with the reward from the game. If the pair already exists, the predicted reward is updated as follows:

$$Q(x, a) = Q(x, a) + \eta[\rho + \gamma E(y) - Q(x, a)] \tag{2}$$

where $Q(x, a)$ is the predicted reward for state x with corresponding action a, η is a learning rate, ρ is the actual reward, γ is a discount factor, and $E(y)$ is the maximum Q value for all actions associated with state y. State y is the state that follows when action a is applied to state x. Reward is determined using the payoff function in (Grefenstette et al. 1990), namely

$$\rho = \begin{cases} 1000 & \text{if } E \text{ evades the pursuers} \\ 10t & \text{if } E \text{ is captured at time } t. \end{cases} \tag{3}$$

Each of the pairs in the game are then compared with all of the pairs in the database. If the distance between a stored state and action are less than d_1 and d_2 respectively for some state-action pair in the game, then the stored state-action pair's Q value is updated.

4.2. *K-NN for evasive maneuvers*

Lazy learning is a classical approach to machine learning and pattern recognition, most commonly in the form of the k-nearest neighbor algorithm. K-NN is rarely used for Markov decision problems, so we had to represent the pursuit game in a format amenable to this algorithm. Further, to be successful, a lazy approach must have a database full of correctly labeled examples, because k-NN expects each example to be labeled with its class name. The difficulty here, then, is how to determine the *correct* action to store with each state.

We formulate Markov decision problems as classification problems by letting the state variables correspond to features of the examples, and the actions correspond to classes. Typically, classification tasks assume a small set of discrete classes to be assigned. We do not require quantization of the state space or the action space, but instead use interpolation so that any action can be produced by the k-NN classifier.

In order to know the *correct* action to store with each state, we must at least wait until we have determined the outcome of a game before deciding how to label each step. (One example can be added at each time step). However, even after a successful game where E evades P, we cannot be sure that the actions at *every* time step were the correct ones; in general, they were not.

To construct an initial database of instances, the simulator generated actions randomly until E evaded P for a complete game. The corresponding state-action pairs for that engagement were then stored. At that point, k-NN was used for future games. States were passed by the simulator to a classifier which searched the database for the k nearest neighbors and selected an action by averaging the associated actions. If k-NN failed to produce a game that ended in successful evasion, the game was replayed with the example generator randomly selecting actions until play ended in evasion. Once evasion occurred,

the corresponding sequence of states and actions (i.e., the complete game) was stored in the database.

Evasion usually occurred after 20 time steps since it was rare in the lazy learner that the pursuers' speeds dropped below the threshold. Thus a stored game typically consisted of 20 state-action pairs. Our implementation uses Euclidean distance to find the k nearest neighbors and the arithmetic mean of their control values to determine the appropriate actions. Distance is computed as follows:

$$\textit{dist}(\textit{state}, \textit{instance}) = \sqrt{\sum_{\forall attrib} (\textit{state}_{attrib} - \textit{instance}_{attrib})^2} \tag{4}$$

Then the nearest neighbor is determined simply as

$$\textit{nn} = \arg \min_{\forall instance} \{\textit{dist}(\textit{state}, \textit{instance})\} \tag{5}$$

If E fails to evade when using the stored instances, we reset the game to the starting position and generate actions randomly until E succeeds. We also generate random actions with probability 0.01 regardless of performance. The resulting set of examples is added to the database.

For the initial experiments using k-nearest neighbors, we varied k between 1 and 5 and determined that $k = 1$ yielded the best performance. (This was not completely surprising in that averaging control values with $k > 1$ tended to "cancel out" values that were extreme. For example, if three instances indicated turns of 90 degrees left, 5 degrees right, and 85 degrees right, the selected action would have been no turn. Of course, we are averaging "cyclic" values where, for example, 359 degrees is close to 1 degree. Improving the averaging process might enable $k > 1$ to perform better.) Examples consisted of randomly generated games that resulted in success for E; thus we could assume that at least some of E's actions were correct. (In random games, every action taken by E is random; the database is not checked for nearby neighbors.)

4.3. *GA for evasive maneuvers*

Grefenstette, et al. demonstrated that genetic algorithms perform well in solving the single pursuer game. Typically, GAs use rules called classifiers, which are simple structures in which terms in the antecedent and the consequent are represented as binary attributes (Booker, Goldberg and Holland 1989; Holland 1975). The knowledge for the evasive maneuvers problem requires rules in which the terms have numeric values; we therefore modified the standard GA representation and operators for this problem, using a formulation similar to (Grefenstette et al. 1990).

We call a set of rules a *plan*. For the GA, each plan consists of 20 rules with the general form:

$$\begin{array}{ll} \text{IF} & \text{low}_1 \leq \text{state}_1 \leq \text{high}_1 \wedge \ldots \wedge \text{low}_n \leq \text{state}_n \leq \text{high}_n \\ \text{THEN} & \text{action}_1, \ldots, \text{action}_m \end{array}$$

Each clause in the antecedant compares a state variable to a lower and upper bound. "Don't care" conditions can be generated by setting the corresponding range to be maximally general. To map this rule form into a chromosome for the GA, we store each of the attribute bounds followed by each action. For example, suppose we have the following rule (for the single pursuer problem):

$$\begin{array}{ll} \text{IF} & 300 \leq \text{speed} \leq 350 \wedge \\ & 25 \leq \text{previous turn} \leq 90 \wedge \\ & 3 \leq \text{clock} \leq 10 \wedge \\ & 875 \leq \text{pursuer speed} \leq 950 \wedge \\ & 8 \leq \text{pursuer bearing} \leq 10 \wedge \\ & 180 \leq \text{pursuer heading} \leq 270 \wedge \\ & 300 \leq \text{pursuer range} \leq 400 \\ \text{THEN} & \text{turn} = 45 \end{array}$$

The chromosome corresponding to this rule would be:

[300 350 25 90 3 10 875 950 8 10 180 270 300 400 45]

Associated with each rule is a rule strength, and associated with each plan is a plan fitness. A population may contain up to fifty plans, all of which compete against each other in the GA system. Strength and fitness values, described below, determine the winners of the competition.

Initially, all rules are maximally general. As a result, all rules will match all states, and one rule will be selected with uniform probability. Following each training game, the rules that fired are generalized or specialized by the GA, using hill-climbing to modify the upper and lower limits of the tests for each state variable as follows:

$$LB_i = LB_i + \beta(state_i - LB_i) \tag{6}$$

$$UB_i = UB_i - \beta(UB_i - state_i) \tag{7}$$

where LB_i and UB_i are the lower and upper bounds, respectively, of the rule that fired for $state_i$ and β is the learning rate. If the current state is within the bounds of the predicate, the bounds shift closer to the state based on the

learning rate ($\beta = 0.1$ for this study). On the other hand, if the state is outside the bounds, only the nearer bound is adjusted by shifting it toward the value $state_i$. Following a game the strengths of the rules that fired are updated based on the payoff received from the game (the same payoff used in Q-learning).

Given the payoff function, the strength for each rule that fired in a game is updated using the profit sharing plan (Grefenstette 1988) as follows:

$$\mu(t) = (1 - c)\mu(t-1) + c\,\rho \tag{8}$$

$$\sigma(t) = (1 - c)\sigma(t-1) + c(\mu(t) - \rho) \tag{9}$$

$$\mathit{strength}(t) = \mu(t) - \sigma(t) \tag{10}$$

where c is the profit sharing rate ($c = 0.01$ for our experiments), ρ is the payoff received, μ is an estimate of the mean strength of a rule, and σ is an estimate of the variance of rule strength. Plan fitness is calculated by running each plan against a set of randomly generated games, and computing the mean payoff for the set of tests. During testing, the plan with the highest fitness is used to control E.

The heart of the learning algorithm lies in the application of two genetic operators: *mutation* and *crossover*. Rules within a plan are selected for mutation using *fitness proportional selection* (Goldberg 1989). Namely, probability of selection is determined as

$$\Pr(r) = \frac{\mathit{strength}_r(t)}{\sum_{\forall s \in \mathit{rules}} \mathit{strength}_s(t)} \tag{11}$$

where *rules* is the set of rules in a plan and r is the rule of interest. Probability of selection for plans is determined similarly using plan fitness rather than rule strength. For more details of the implementation, see (Sheppard and Salzberg 1993).

4.4. *Results*

For each of the algorithms and for both variations of the evasive maneuvers game, we ran ten experiments. To produce learning curves, we combined the results of the ten experiments by averaging the algorithm's performance at regular intervals. We estimated the accuracy of each algorithm by testing the results of training on 100 randomly generated games.

The results of the Q-learning experiments were encouraging and led to the next phase of our study in which we applied a traditional lazy learning method, k-nearest neighbors (k-NN), to the evasive maneuvers task. When we found that k-NN did not work well, we considered an eager learning

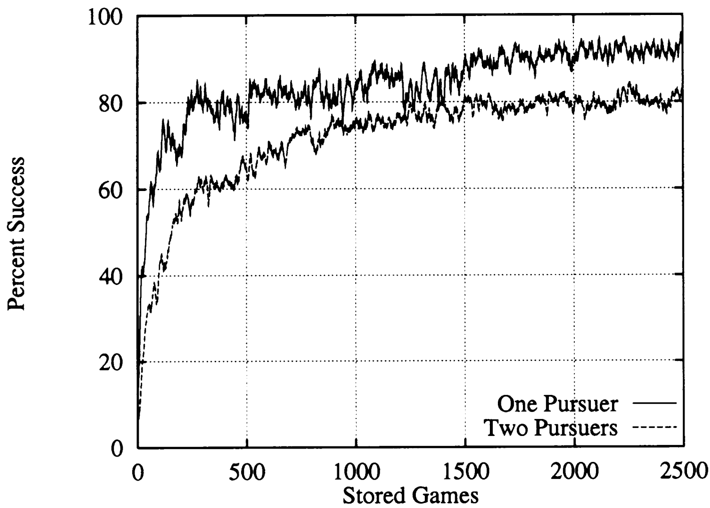

Figure 2. Performance of Q-learning on one- and two-player pursuit games.

algorithm, the genetic algorithm. This choice was motivated by the previous work by Grefenstette et al. which indicated the GA was capable of solving this type of task. In fact, we were able to replicate those results for the one-pursuer problem and scale up the GA so that it still worked quite well for the two-pursuer game.

4.4.1. *Performance of lazy Q-learning*

In the one-pursuer task, Q-learning did extremely well initially (Figure 2), reaching 80% evasion within the first 250 games, but then performance flattened out. Peak performance (when the experiments were stopped) was about 90%. There was an apparent plateau between 250 games and 1500 games where performance remained in the range 80%–85%. Then performance jumped to another plateau at 90% for the remainder of the experiment.

Q-learning's performance on the two-pursuer task was also encouraging. It reached 60% evasion within 250 games and continued to improve until reaching a plateau at 80%. This plateau was maintained throughout the remainder of the experiment. Since our implementation of Q-learning uses a form of lazy-learning, these results led us to believe it might be possible to design a more traditional lazy method (i.e., k-NN) to solve the evasion task. At first, however, our hypothesis was not supported, as we see in the next section.

4.4.2. *Performance of k-NN*

Figure 3 shows how well k-NN performed on the two versions of the evasive maneuvers game as the number of training examples (and games) increased.

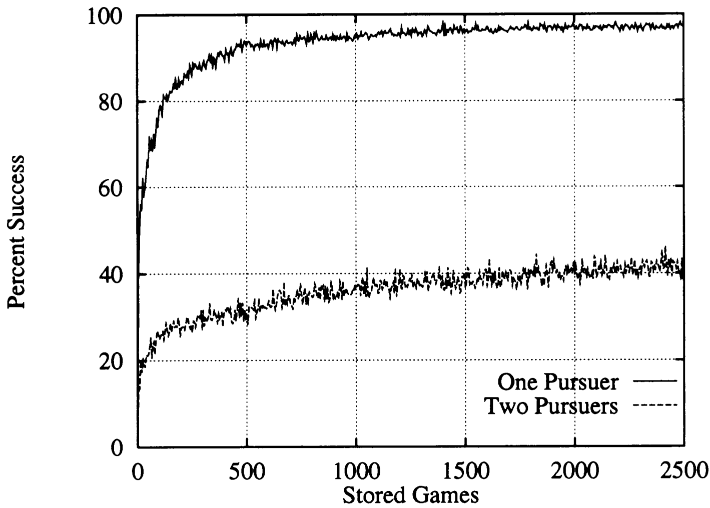

Figure 3. Performance of k-NN on one- and two-player pursuit games.

This figure compares the performance on the two problems with respect to the number of games stored, where a game contains up to 20 state-action pairs.

These experiments indicate that the problem of escaping from a single pursuer is relatively easy to solve. K-NN developed a set of examples that was 95% successful after storing approximately 1,500 games, and it eventually reached almost perfect performance. The distance between P and E at the start of the game guarantees that escape is always possible. However, the results were disappointing when E was given the task of learning how to escape from two pursuers. In fact, the lazy learning approach had difficulty achieving a level of performance above 45%. This demonstrates that the two-pursuer problem is significantly more difficult for k-NN.

One possible reason for k-NN's poor performance on the two-pursuer task is presence of irrelevant attributes, which is known to cause problems for nearest neighbor algorithms (Aha 1992; Salzberg 1991). We experimented with a method similar to stepwise forward selection (Devijver and Kittler 1982) to determine the set of relevant attributes. However, determining relevant attributes in a dynamic environment is difficult for the same reason that determining good examples is difficult: we do not know which attributes to use until many successful examples have been generated.

Another possible reason for the poor performance of k-NN on the two pursuer task is the size of the search space. For the one-pursuer problem, the state space contains $\approx 7.5 \times 10^{15}$ points, whereas for two-pursuer evasion, the state space has $\approx 2.9 \times 10^{33}$ points. The one-pursuer game showed good

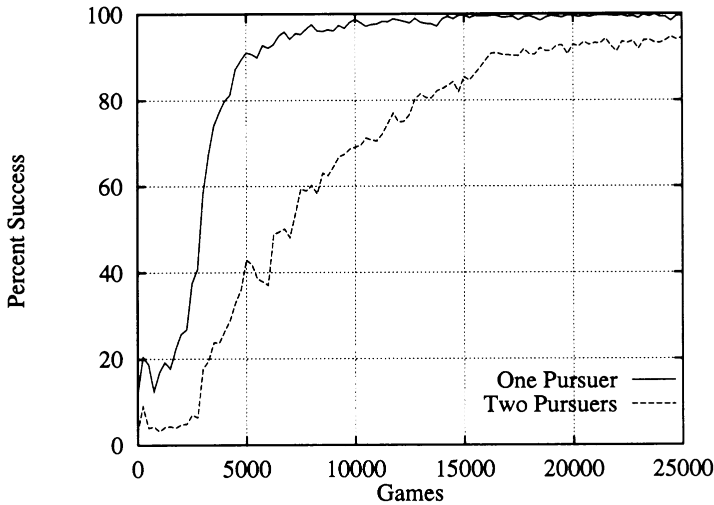

Figure 4. Performance of the genetic algorithm on one- and two-player pursuit games.

performance after 250 games; to achieve similar coverage of the state space in the two-pursuer game would require roughly 2.7×10^{21} games (assuming similar distributions of games in the training data).

But the most likely reason for k-NN's troubles, we concluded, was that we were generating bad examples in the early phases of the game. As stated above, a lazy learner needs to have the "correct" action, or something close to it, stored with almost every state in memory. Our strategy for collecting examples was to play random games at first, and to store games in which E succeeded in escaping. However, many of the actions taken in these random games will be incorrect. E might escape because of one or two particularly good actions, but a game lasts for 20 time steps, and all 20 state-action pairs are stored. Since our lazy learning approach had no way (at first – see section 5.2) to throw away examples, if it collected many bad examples it could get stuck forever at a low level of performance.

4.4.3. *Performance of the GA*

We show the results of the GA experiments in Figure 4. As with k-NN, the GA performs well when faced with one pursuer. In fact, it achieves near perfect performance after 15,000 games and very good performance (above 90%) after only 5,000 games. The number of games is somewhat inflated for the GA because it evaluates 50 plans during each generation, thus we counted one generation as 50 games. In fact, the simulation ran for only 500 generations (i.e., 25,000 games) in these experiments.

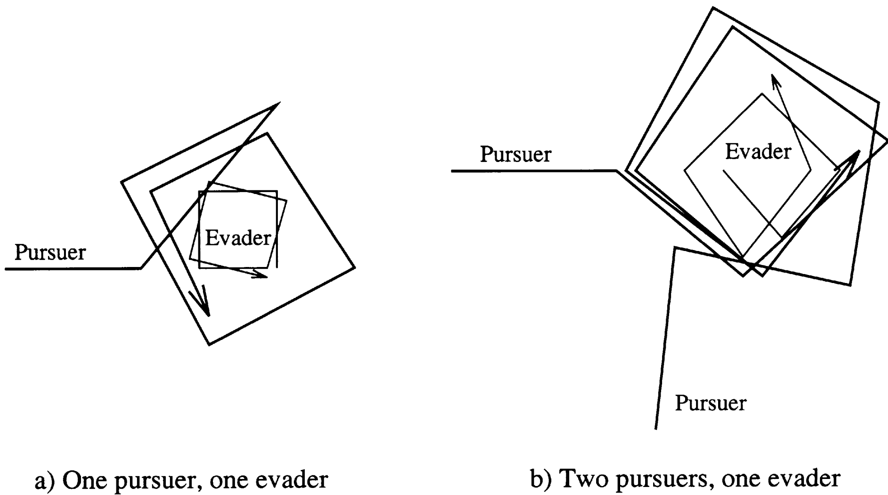

Figure 5. Sample games where E successfully evades.

The most striking difference in performance between k-NN and the genetic algorithm is that the GA learned excellent strategies for the two-pursuer problem, while nearest neighbor did not. Q-learning's performance, though much better than k-NN, is still inferior to the GA. Indeed, the GA achieved above 90% success after 16,000 games (320 generations) and its success rate continued to improve until it reached approximately 95%.

4.4.4. *Comparing one- and two-pursuer evasion*

Figure 5a shows a sample game in which E evades a single pursuer, which gives some intuition of the strategy that E had to learn. Essentially, E just keeps turning sharply so that P will be unable to match its changes of direction. Although all three algorithms did well on this task, a closer examination of the results reveals some interesting differences.

K-NN eventually reached a successful evasion rate of 97%–98%, and it reached 93% evasion after only 500 games. This was superior to Q-learning's asymptotic performance, and k-NN performed better than the GA through 250 games. Of course, the GA eventually achieved near perfect performance. Q-learning also learned very rapidly in the beginning, exceeding the GA's ability through the first 150 games, but then its learning slowed considerably. In fact, at the point the GA was performing nearly perfectly, Q-learning's performance was only around 85%. After twice as many games as the GA, Q-learning (now achieving 91% evasion) was still performing considerably poorer than both the GA and k-NN.

Table 1. Comparing learning for the evasive maneuvers task at convergence.

Algorithm	One Pursuer	Two Pursuers
k-NN	96.9%	42.3%
Q-learning	93.3%	81.7%
GA	99.6%	94.5%

Table 1 shows the results of comparing the three algorithms on the two evasion tasks at convergence. We considered the algorithms to have converged when they showed no improvement through 500 games (for k-NN and Q-learning) or through 100 generations (for the GA). Recognizing the difficulty of the two-pursuer task (relative to the one-pursuer task), we now see profound differences in the performance of the three approaches. (See Figure 5b for a sample game where E evades two pursuers.) As before, the GA started slowly, being outperformed by both k-NN and Q-learning. After about 3,000 games (60 generations), the GA began to improve rapidly, passing k-NN almost immediately, and catching Q-learning after an additional 5,000 games (100 generations). The end results show the GA surpassing both Q-learning (by a margin of 11%) and k-NN (by a margin of 52%). The more striking result, though, is the poor performance of k-NN for the two-pursuer game. We next set out to improve this figure.

5. Combining the GA with Lazy Learning

Initially, we were surprised with k-NN's performance on the two-pursuer task. In an attempt to improve its performance, we considered how to provide "good" examples to k-NN, based on our hypothesis that the primary cause of its poor performance is the poor quality of its training experiences. For lazy learning to work effectively on control tasks, the stored examples must have a high probability of being good ones; i.e., the action associated with a stored state should be correct or nearly correct. Because of this credit assignment problem, and because of the difficulty of the tasks we designed, initial training is very difficult for a lazy learner. In contrast, a GA initially searches a wide variety of solutions, and for the problems we studied tends to learn rapidly in the early stages. These observations suggested the two-phase approach that we adopted, in which we first trained a GA, and then used it to provide examplars to bootstrap k-NN.

```
algorithm GLL;
init population;
do
    run genetic algorithm;                  /* Run the GA for one generation */
    perf = select best plan;                /* Determine performance of GA */
    if perf ≥ θ                             /* Evaluate performance against θ = 0, 50, 90 */
        do i = 1, n                         /* For our experiments n = 100 */
            evade = evaluate best;          /* Determine if best plan from GA evades */
            if evade
                store examples;             /* Stores up to 20 examples */
    evaluate lazy;                          /* Test on 100 games */
```

Figure 6. Pseudocode for GLL.

5.1. *Bootstrapping nearest neighbor*

Our bootstrapping idea requires that one algorithm train on its own for a time, and then communicate what it has learned to a second algorithm. At that point, the second algorithm takes over. Later, the first algorithm adds additional examples. This alternation continues until the combined system reaches some asymptotic limit. Because the GA learned much better for the two-pursuer game, we selected it as the first learner, with k-NN second. Details of the communication or "teaching" phase are given in Figure 6. Using this approach, the examples continue to accumulate as the genetic algorithm learns the task.

The results of training k-NN using the GA as the teacher are shown in Figure 7. We call this system GLL because it first uses a GA and then uses a lazy learning algorithm (i.e., k-NN). All points shown in the graph are the averages of 10 trials.

The first threshold was set to 0%, which meant that the GA provided examples to k-NN from the beginning of its own training. The second threshold was set to 50% to permit the GA to achieve a level of success approximately equal to the best performance of k-NN on its own. Thus only plans that achieved at least 50% evasion were allowed to transmit examples to k-NN. Finally, the threshold was set at 90% to limit examples for k-NN to games in which a highly trained GA made the decisions about which examples to store.

When $\theta = 0\%$, GLL almost immediately reaches a level equal to the best performance of k-NN on its own (around 45%). From there, it improves somewhat erratically but steadily until it reaches a performance of approximately 97% success. The figure shows performance plotted against the number of examples stored. The number of examples stored here is higher than the number of examples stored for k-NN alone. If we halt learning after 50,000

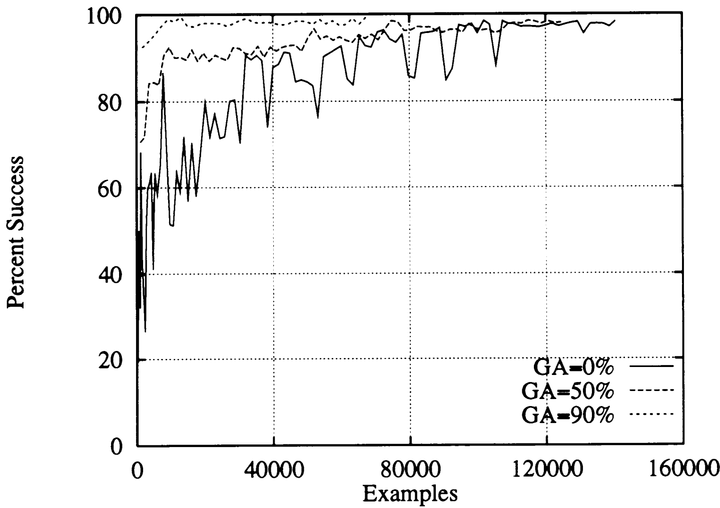

Figure 7. Results of GA teaching k-NN.

examples (which is consistent with the earlier k-NN experiments), performance would be in the 85% range, still an enormous improvement over k-NN's performance, but not better than the GA on its own.

When $\theta = 50\%$, GLL starts performing at a very high level (above 70%) and quickly exceeds 90% success. After 50,000 examples, GLL obtained a success rate above 95%, with some individual trials (on random sets of 100 games) achieving 100% success. In addition, the learning curve is much smoother, indicating that k-NN is probably not storing many "bad" examples. This confirms in part our earlier hypothesis that k-NN's fundamental problem was the storage of bad examples. If it stores examples with bad actions, it will take bad actions later, and its performance will continue to be poor whenever a new state is similar to one of those bad examples.

Finally, with $\theta = 90\%$, GLL's performance was always superb, exceeding the GA's 90% success rate on its very first set of examples. GLL converged to near-perfect performance with only 10,000 examples. One striking observation was that GLL performed better than the GA throughout its learning. For example, when $\theta = 0\%$, GLL achieved 50–80% success while the GA was still only achieving 2–10% success. Further, GLL remained ahead of the GA throughout training. Even when $\theta = 90\%$, GLL achieved 98–100% evasion while the GA was still only achieving around 95% evasion. Neither the GA nor k-NN were able to obtain such a high success rate on their own, after any number of trials.

5.2. *Reducing memory size*

Our bootstrapping algorithm, GLL, performs well even when only a small number of examples are provided by the GA, and it even outperforms its own teacher (the GA) during training. But the amount of knowledge required for the GA to perform well on the task was quite small – only 20 rules are stored as a single plan. The number of examples used by GLL, though small in comparison with k-NN, still requires significantly more space and time than the rules in the GA. Consequently, we decided to take this study one step further, and attempted to reduce the size of the memory store during the lazy learning phase of GLL (Zhang 1992; Skalak 1994).

In the pattern recognition literature, e.g., in (Dasarathy 1991), algorithms for reducing memory size are known as *editing* methods. However, because lazy learning is not usually applied to control tasks, we were not able to find any editing methods specifically tied to our type of problem. We therefore modified a known editing algorithm for our problem, and call the resulting system GLE (GA plus lazy learning plus editing).

GLL performs quite well as described above, and we would like to reduce its memory requirements without significantly affecting performance. Early work by Wilson (1972) showed that examples could be removed from a set used for classification, and suggested that simply editing would frequently improve classification accuracy (in the same way that pruning improves decision trees (Mingers 1989)). Wilson's algorithm classifies each example in a data set with its own k nearest neighbors. Those points that are incorrectly classified are deleted from the example set, the idea being that such points probably represent noise. Tomek (1976) modified this approach by taking a sample (> 1) of the data and classifying the sample with the remaining examples. Editing then proceeds using Wilson's approach. Ritter et al. (1975) described another editing method, which differs from Wilson in that points that are *correctly* classified are discarded. The Ritter method, which is similar to Hart's (1968), basically keeps only points near the boundaries between classes, and eliminates examples that are in the midst of a homogenous region.

The editing approach we took combined the editing procedure of Ritter et al. and the sampling idea of Tomek (Devijver 1986). We began by generating ten example sets with $\theta = 90$ where each set consisted of a single set of examples from the GA. We then selected the set with the best performance on 10,000 test games, which in this case obtained nearly perfect accuracy with 1,700 examples. Next we edited the memory base by classifying each example using all other examples in the set. For this phase, we used the five nearest neighbors. If a point was correctly classified, we deleted it with probability 0.25. (This probability was selected arbitrarily, and was used to

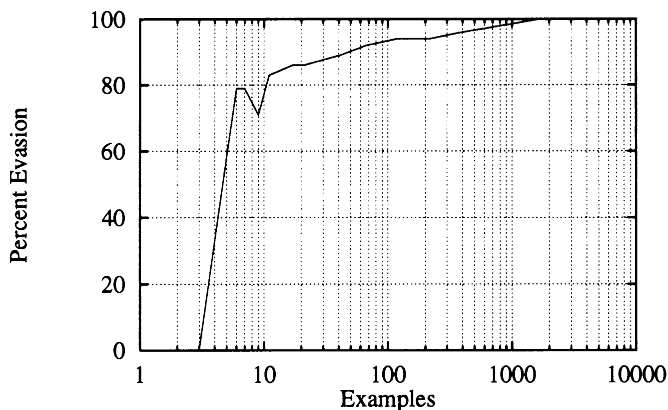

Figure 8. Results of editing examples provided by the genetic algorithm for k-NN.

show how performance changed as editing occurred.) Prior to editing and after each pass through the data, the example set was tested using 1-NN on 10,000 random games.

One complication in "classifying" the points for editing was that the class was actually a three-dimensional vector of three different actions, two of which were real-valued (turn angle and speed) and one of which was binary (emitting smoke). It was clear that an exact match would be too strict a constraint. Therefore we specified a range around each 3-vector within which the system would consider two "classes" to be the same. In addition, the three values were normalized to equalize their effect on this range measurement.

The results of running GLE on the 1,700 examples are summarized in Figure 8.

A logarithmic scale is used on the x-axis to highlight the fact that accuracy decreased very slowly until almost all the examples were edited. When read from right to left, the graph shows how accuracy decreases as the number of examples decreases. With as few as 11 examples, GLE achieved better than 80% evasion, which is substantially better than the best ever achieved by k-NN alone. With 21 examples (comparable in size to a plan in the GA), GLE achieved 86% evasion. Performance remained at a high level (greater than 90% success) with only 66 examples. Thus it is clear that a small, well chosen set of examples can yield excellent performance on this difficult task. Furthermore, such a small memory base guarantees that the on-line performance of k-NN will be quite fast.

6. Discussion and Conclusions

This study considered approaches to developing strategies for learning to play differential games. In particular, we examined several methods for learning evasion strategies in pursuit games. Optimal strategies for differential games are determined by solving a system of differential equations; for even simple games, the resulting strategies are complex. Many games do not have known closed-form solutions. We illustrated the complexity of differential strategies with the pursuit game.

These experiments demonstrated the ability of three algorithms (GA, k-NN, and lazy Q-learning) to perform well on a simple task with one pursuer, and then show how increasing the difficulty by adding a second pursuer adversely affects two of the algorithms (k-NN and lazy Q-learning). K-NN, in particular, had considerable difficulty scaling up to the more complex task. Thus we were left with the question of whether it is even possible for lazy learning techniques such as k-nearest neighbor to perform well on problems of this type. This motivated the second phase of the study, in which we used a GA-based teacher to train k-NN for the two-pursuer task.

The experiments reported here show that it is possible to use genetic algorithms in conjunction with lazy learning to produce agents that perform well on difficult delayed reinforcement learning problems. The experiments also demonstrate clearly the power of having a teacher or other source of good examples for lazy learning methods. For complex control tasks, such a teacher is probably a necessary component of any lazy or memory-based method. Our experiments show how a genetic algorithm can be used to learn plans or control laws in complex domains, and then to train a lazy learner by using its learned rules to generate good examples. The result was a hybrid system that outperformed both of its "parent" systems. This hybrid approach can of course be applied in many ways; for example, standard Q-learning is notoriously slow to converge, and approaches such as ours could be used to accelerate it.

One surprising result was that the performance of GLL outperformed the GA at the same point in training. We hypothesize this was because only the best examples of a given generation were passed to k-NN, rather than all of the experiences of the GA during that generation. The fact that GLL outperformed GA right away indicates that perhaps it could have been used to teach the GA, instead of the other way around.

In addition, we found that editing the example set produced a relatively small set of examples that still play the game extremely well. Again, this makes sense since editing served to identify the strongest examples in the database, given that poor examples were still likely to be included in the early stages of learning. It might be possible with careful editing to reduce the

size of memory even further. This question is related to theoretical work by Salzberg et al. (1991) that studies the question of how to find a minimal-size training set through the use of a "helpful teacher," which explicitly provides very good examples. Such a helpful teacher is similar to the oracle used by Clouse and Utgoff (1992) except that it provides the theoretically minimal number of examples required for learning.

7. Next Steps

Our current implementation takes only the first step towards a truly combined learning system in which the two learners would assist each other in learning the task. Our approach uses one algorithm to start the learning process and hands off the results of the first algorithm to a second algorithm to continue learning the task. We envision a more general architecture in which different learning algorithms take turns learning, depending on which one is learning most effectively at any given time. Such an architecture should expand the capabilities of learning algorithms as they tackle increasingly difficult control problems.

One possible future direction is to use genetic operators (or other methods) directly on the examples in a lazy learning approach. That is, rather than producing rules, we can begin with a set of examples and mutate those directly using genetic operators to evolve a database (i.e., example set) to perform the task. One such approach might be to examine the frequency with which the examples are used to successfully evade and then select the n most frequently used examples. These examples can then be converted into a plan for the GA by specifying a range about each attribute in each example. This results in a new set of rules which is sufficient to construct a plan, and the new plan can be seeded into the population for the GA to use.

The general problem of determining optimal strategies in differential games is complex. Solving the games involves solving a system of differential equations. Learning solutions to the games involves simultaneous learning by all of the players. This means that the players must learn in a highly dynamic environment. Rather than a player learning to counter a single, constant strategy, the player must adapt its strategy to the changing strategy of the opponent. In such an environment, one must avoid prematurely converging on a fixed solution.

To study these problems, we are building an environment for analyzing learning algorithms in multi-agent environments. Specifically, we wish to explore the effects on sequential decision making when several agents are learning at the same time. We are exploring the ability of an agent to apply one approach to learn evasion tactics while another agent is using the same

or perhaps a different approach to develop pursuit strategies. We will also pit a strategy against itself and study whether a single learning algorithm can develop multiple solutions for the same reactive control task.

Acknowledgments

Thanks to David Aha and the anonymous reviewers of this special issue for their many valuable comments on an earlier draft of this paper. Also thanks to Diana Gordon, John Grefenstette, Simon Kasif, and S.K. Murthy for helpful comments and ideas during the formative stages of this work. This material is based upon work supported in part by the National Science foundation under Grant Nos. IRI-9116843 and IRI-9223591.

References

Aha, D. & Salzberg, S. (1993). Learning to catch: Applying nearest neighbor algorithms to dynamic control tasks. In *Proceedings of the Fourth International Workshop on AI and Statistics*, pp. 363–368. Ft. Lauderdale.

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

Atkeson, C. (1990). Using local models to control movement. In Touretzky, D. S. (ed.), *Advances in Neural Information Processing Systems* **2**, 316–323. San Mateo, CA: Morgan Kaufman.

Atkeson, C. G. (1992). Memory-based approaches to approximating continuous functions. In Casdagli, M. & Eubanks, S. (eds.), *Nonlinear Modeling and Forecasting*, pp. 503–521. Addison Wesley.

Barto, A., Sutton, R. & Anderson, C. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics* **13**: 835–846.

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

Basar, T. & Olsder, G. J. (1982). *Dynamic Noncooperative Game Theory*. Academic Press: London.

Booker, L., Goldberg, D. & Holland, J. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence* **40**: 235–282.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence* **32**: 333–377.

Clouse, J. & Utgoff, P. (1992). A teaching method for reinforcement leraning. In *Proceedings of the Ninth International Conference on Machine Learning*, pp. 92–101. Aberdeen, Scotland: Morgan Kaufman.

Colombetti, M. & Dorigo, M. (1994). Training agents to perform sequential behavior. *Adaptive Behavior* **2**(3): 247–275.

Dasarathy, B. V. (ed.) (1991). *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques*. Los Alamitos, CA: IEEE Computer Society Press.

Devijver, P. A. (1986). On the editing rate of the multiedit algorithm. *Pattern Recognition Letters* **4**: 9–12.

Devijver, P. A. & Kittler, J. (1982). *Pattern Recognition: A Statistical Approach*. Englewood Cliffs, New Jersey: Prentice-Hall.

Dorigo, M. & Colombetti, M. (1994). Robot shaping: Developing autonomous agents through learning. *Artificial Intelligence* **71**(2): 321–370.

Friedman, A. (1971). *Differential Games*. New York: Wiley Interscience.

Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, Massachusetts: Addison-Wesley.

Gordon, D. & Subramanian, D. (1993a). A multistrategy learning scheme for agent knowledge acquisition. *Informatica* **17**: 331–346.

Gordon, D. & Subramanian, D. (1993b). A multistrategy learning scheme for assimilating advice in embedded agents. In *Proceedings of the Second International Workshop on Multistrategy Learning*, pp. 218–233. George Mason University.

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. In *Proceedings of the Fourth International Conference of Genetic Algorithms*, pp. 303–310. Morgan Kaufmann.

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

Hart, P. (1968). The condensed nearest neighbor rule. *IEEE Transactions on Information Theory* **14**(3): 515–516.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, Michigan: University of Michigan Press.

Imado, F. & Ishihara, T. (1993). Pursuit-evasion geometry analysis between two missiles and an aircraft. *Computers and Mathematics wth Applications* **26**(3): 125–139.

Isaacs, R. (1963). Differential games: A mathematical theory with applications to warfare and other topics. Tech. Rep. Research Contribution No. 1, Center for Naval Analysis, Washington, D.C.

Lin, L. (1991). Programming robots using reinforcement learning and teaching. In *Proceedings of the Eight National Conference on Artificial Intelligence*, pp. 781–786. AAAI Press.

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

McCallum, R. A. (1995). Instance-based state identification for reinforcement learning. In *Advances in Neural Information Processing Systems* **7**, pp. 377–384.

Millan, J. & Torras, C. (1992). A reinforcement connectionist approach to robot path finding in non-maze-like environments. *Machine Learning* **8**: 363–395.

Mingers, J. (1989). An empirical comparison of pruning methods for decision tree induction. *Machine Learning* **4**(2): 227–243.

Moore, A. (1990). *Efficient Memory-Based Learning for Robot Control*. Ph.D. thesis, Computer Laboratory, Cambridge University.

Moore, A. & Atkeson, C. (1993). Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning* **13**: 103–130.

Nguyen, D. & Widrow, B. (1989). The truck backer-upper: An example of self learning in neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, pp. 357–363.

Pell, B. D. (1993). *Strategy Generation and Evaluation for Meta-Game Playing*. Ph.D. thesis, University of Cambridge, Cambridge, England.

Ramsey, C. L. & Grefenstette, J. J. (1994). Case-based anytime learning. In Aha, D. W. (ed.), *Case Based Reasoning: Papers from the 1994 Workshop*, pp. 91–95. Menlo Park, California: AAAI Press.

Ritter, G., Woodruff H., Lowry S. & Isenhour, T. (1975). An algorithm for a selective nearest neighbor decision rule. *IEEE Transactions on Information Theory* **21**(6): 665–669.

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

Salzberg, S., Delcher, A., Heath, D. & Kasif, S. (1991). Learning with a helpful teacher. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pp. 705–511. Sydney, Australia: Morgan Kaufmann.

Sheppard, J. W. & Salzberg, S. L. (1993). Memory-based learning of pursuit games. Tech. Rep. JHU-93/94-02, Department of Computer Science, Johns Hopkins University, Baltimore, Maryland. Revised May, 1995.

Skalak, D. (1994). Prototype and feature selection by sampling and random mutation hill climbing algorithms. In *Proceedings of the Eleventh International Machine Learning Conference*, pp. 293–301. New Brunswick, NJ: Morgan Kaufman.

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

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

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

Tesauro, G. & Sejnowski, T. J. (1989). A parallel network that learns to play backgammon. *Artificial Intelligence* **39**: 357–390.

Tomek, I. (1976). An experiment with the edited nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics* **6**: 448–452.

van der Wal, J. (1981). *Stochastic Dynamic Programming*. Amsterdam: Morgan Kaufmann.

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

Whitehead, S. (1992). *Reinforcement Learning for the Adaptive Control of Perception and Action*. Ph.D. thesis, Department of Computer Science, University of Rochester.

Widrow, B. (1987). The original adaptive neural net broom-balancer. In *International Symposium on Circuits and Systems*, pp. 351–357.

Wilson, D. (1972). Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man, and Cybernetics* **2**(3): 408–421.

Zhang, J. (1992). Selecting typical instances in instance-based learning. In *Proceedings of the Ninth International Machine Learning Conference*, pp. 470–479. Aberdeen, Scotland: Morgan Kaufman.