Combining Genetic Algorithms with Memory Based Reasoning

John W. Sheppard* **and Steven L. Salzberg**
Department of Computer Science
The Johns Hopkins University
Baltimore, Maryland 21218
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 memory-based reasoning to solve a difficult class of delayed reinforcement learning problems that both methods have trouble solving individually. This class includes numerous important control problems that arise in robotics, planning, game playing, and other areas. Our experiments demonstrate that by using one learning technique, genetic algorithms, as a bootstrapping method for the second learning technique, memory-based reasoning, we can create a system that outperforms either method alone. The resulting joint system learns to solve a difficult reinforcement learning task with a high degree of accuracy and with relatively small memory requirements.

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 problems. This paper describes how a genetic algorithm and a memory-based learning algorithm can work together to produce better solutions than either method could produce by itself.

Delayed reinforcement learning problems involve an agent that makes a sequence of decisions, or actions, in an environment that provides feedback about those decisions. The feedback about those actions might be considerably delayed, and this delay makes learning much more difficult. A number of reinforcement learning algorithms have been developed specifically for this family of problems, but little has been done evaluating the power of combining different types of learning algorithms to these problems.

We began by considering a reinforcement learning problem that involved one agent trying to pursue and capture another. Earlier research showed that at least one implementation of this task, known as *evasive maneuvers* (Grefenstette *et al.* 1990), can be solved by a genetic algorithm (GA). We first developed a memory-based approach for the same task, and then made the task substantially harder, in order to study the limitations of both GAs and memory-based reasoning (MBR) methods on this class of problems. The more complicated task, which is described further in section 3, also resembles complicated planning tasks in which an agent has to satisfy several goals at once (Chapman 1987). Our study is an attempt to develop a learning strategy that will improve the overall performance of either MBR, GAs, or both.

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 a GA and MBR. We found that the best learning agent first used a GA, and then switched to MBR after reaching a certain performance threshold. Our experiments demonstrate remarkable improvement in the performance of MBR, 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

The idea of using memory-based methods for delayed reinforcement tasks has only very recently been considered by a small number of researchers. Atkeson

*Also ARINC, Inc., 2551 Riva Road, Annapolis, Maryland 21401, sheppard@arinc.com

(1989) employed a memory-based technique to train a robot arm to follow a prespecified trajectory. More recently, Moore and Atkeson (1993) developed an algorithm called "prioritized sweeping" in which "interesting" examples in a Q table are the focus of updating. 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.

Genetic algorithms have also been applied to perform delayed reinforcement problems. In addition to studying the evasive maneuvers task, Grefenstette (1991) applied his genetic algorithm system SAMUEL to aerial dogfighting and target tracking (in a game of cat and mouse).

One of the most popular approaches to reinforcement learning has been using neural network learning algorithms. For example, back-propagation has been used to solve the cart and pole problem (Widrow 1987), in which a pole must be balanced vertically on a wheeled cart. A similar algorithm was applied to learning strategies for backing a truck into a loading dock (Nguyen and Widrow 1989). Both of these methods incorporated knowledge of the correct behavior during training. In addition, Millan and Torras (1992) demonstrated a technique for training a neural network using reinforcement learning in which the control variables are permitted to vary continuously. They addressed the problem of teaching a robot to navigate around obstacles.

Finally, considerable research has been performed using a form of reinforcement learning called *temporal difference learning* (Sutton 1988). Temporal difference methods apply delayed reinforcement to a sequence of actions to predict 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 *et al.* 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. Lin (1991) applied 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.

3 DIFFERENTIAL GAMES AND REINFORCEMENT LEARNING

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. An MBR approach must store experiences, in the form of state-action pairs, for later use. To make this work effectively for reinforcement learning, we needed a method that would increase the probability that a stored example was a good one; i.e., that the action associated with a stored state was correct. Because of this credit assignment problem, and because of the difficulty of the tasks we designed, initial training is very difficult for MBR. 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 exemplars to MBR.

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 an extension 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).

One class of differential games is the pursuit game. A pursuit game has two players, called the pursuer (**P**) and the evader (**E**). The evader attempts to achieve an objective, frequently just 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 the typical feature of pursuit games that the pursuer and the evader have different abilities: different speeds, different defense mechanisms, and different sensing abilities. One example of a genetic algorithm approach to pursuit games is Grefenstette *et al.*'s system using SAMUEL (Grefenstette *et al.* 1990), a 2-D simulation in which the pursuer represents a missile and the evader is an airplane. This system's success showed that GAs can solve reinforcement learning problems.

We initially implemented the same pursuit game as Grefenstette *et al.*, and later we extended it to make it substantially more difficult. First, we added a second pursuer. Unlike the single-pursuer problem, the two-pursuer problem has no known optimal strategy (Imado and Ishihara 1993). Second, we gave the evader additional capabilities: in the one-pursuer

game, the evader only controls its turn angle at each time step. Thus **E** basically zigzags back and forth or makes a series of hard 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. 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. 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** can escape and win the game. **E** also wins by successfully evading the pursuers for 20 times steps. 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**). We use the term "game" to include a complete simulation run, beginning with the initial placements of all three players, and ending when **E** either wins or loses.

When playing against one pursuer, the capabilities of **E** are identical to the "aircraft" used by Grefenstette. Against one pursuer, **E** controls only its turn angle, which is sufficient to play the game very well. With two pursuers **P1** and **P2** in the game, **E** has additional information about its opponents. The information provided to **E** includes 13 features describing the state of the game, including its 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 (or "sensors") 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 90 degrees. Heading is the angle between **E**'s direction and the pursuer's direction. When fleeing two pursuers, **E** has control over its speed and turn angle at every time step, and it can also periodically release "smoke"; i.e., send a signal that introduces noise into the sensor readings of **P1** and **P2**.

4 USING MBR FOR EVASIVE MANEUVERING

Memory-based reasoning is a classical approach to machine learning and pattern recognition, most commonly in the form of a k-nearest neighbor algorithm (k-NN), but it is rarely used for reactive control problems. We had to represent the pursuit game in a format amenable to MBR. We chose to let the state variables in the game correspond to features of an example, and the actions taken by **E** correspond to classes. To be successful, a memory-based approach must have a database full of correctly labeled examples. The difficulty here, then, is how to determine the *correct* action to store with each state.

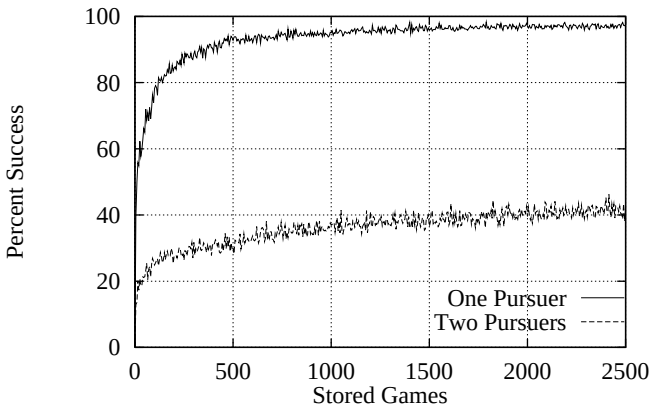

Figure 1: Performance of K-NN.

To illustrate the problems that MBR has with the two-player pursuit game, we briefly describe here some findings of our earlier study (Sheppard and Salzberg 1993), which compared k-NN, GAs, and the temporal difference algorithm called Q-learning (Watkins 1989). In that study, we found that the MBR method, k-NN, was by far the worst method in its performance on this problem. While Q-learning performed well, the GA was superior to both of the other two methods.

For the initial experiments using k-nearest neighbors, we varied k between 1 and 5 and determined that $k = 1$ yielded the best performance. 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. Figure 1 shows how well k-NN performed (where $k = 1$) on a test suite of games other than those used in training as the number of training examples (games) increased. This figure indicates performance averaged over 10 trials for **E** playing against one pursuer and two pursuers. The accuracy at each point in the graph was estimated by testing the MBR on 100 randomly generated games and recording the number of successful evasions. Note that this figure compares the performance between these two tasks 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. MBR was able to develop 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.) When **E** was given the task of learning how to escape from two pursuers, however, the results were

disappointing. In fact, the MBR approach had difficulty achieving a level of performance above 40%. This demonstrates that the two pursuer problem is significantly more difficult for memory-based reasoning.

One 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 performance after 5,700 games; to achieve similar coverage of the state space in the two-pursuer game would require roughly 5.4×10^{22} games. Another possible reason for MBR's problems is the presence of irrelevant attributes, which is known to cause problems for nearest neighbor algorithms (Aha and Kibler 1989, Salzberg 1991).

But the most likely reason for MBR's troubles, we concluded, was that we were generating bad examples in the early phases of the game. As stated above, MBR needs to have the "correct" action, or something close to it, stored with 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, it seems clear that 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 MBR approach had no way (at first) to throw away examples, if it collects many bad examples it could be forever stuck at a low level of performance. This seemed to be what was happening. As a result, we developed a two-phase approach in which we used a genetic algorithm to learn in the early phase. The GA is explained next.

5 THE GENETIC ALGORITHM

Genetic algorithm implementations for games require, first, a representation of the domain in terms of production rules, which test some features of the domain and then take an action (Booker *et al.* 1989, Holland 1975). The state variables in a pursuit game are primarily numeric variables measuring the relative positions and speeds of **P** and **E**, and the conclusion is **E**'s decision about what action to take. In our GA, **E** learned a set of rules, called a *plan*, that together told **E** how to act in every state. A plan consists of 20 rules of the form:

$$(low_1 \leq state_1 \leq high_1 \wedge \ldots \wedge low_n \leq state_n \leq high_n) \rightarrow (action_1, \ldots, action_m)$$

Each clause on the left hand side compares a state variable to a lower and upper bound. "Don't care" conditions can be generated by setting the corresponding upper and lower range bounds to be maximal. We associated a strength value with every rule, and a fitness value with each plan. In our implementation, the GA explored a population containing up to fifty plans. The plan with the greatest fitness was chosen at the end of each training run (i.e., generation).

The GA system contains two main components: an inference system and a learning system. The inference system consists of a rule matcher and a rule specializer. The rule matcher examines the set of rules in a plan to determine which rule to fire, and it selects the rule or rules with the most matches. In the event of a tie, it selects one of the tied rules using a random selection scheme based on the strengths of the rules.

At the start of learning, all rules are maximally general; i.e., all the lower and upper bounds are set to the minimum and maximum legal values. As a result, all rules will match all states, and one rule will be selected with probability proportional to its strength. All rule strengths are initially equal as well. Following each game, the GA generalizes or specializes the rules using a hill-climbing approach as follows.[1] The upper and lower limits of the tests for each state variable are modified according to:

$$bound_i = bound_i + \beta(state_i - bound_i)$$

where $bound_i$ represents one of the upper or lower bounds of the rule that fired in state i, and β is the learning rate ($\beta = 0.1$ for this study). If variable X in state i had a value v within the bounds tested by the rule, then the GA specializes the rule by shifting both bounds towards v. On the other hand, if v is outside the bounds, the nearer bound is adjusted towards v thus generalizing the rule. Following a game the strengths of the rules that fired are updated based on the payoff received from the game, namely

$$payoff = \begin{cases} 1000 & \text{if } \mathbf{E} \text{ evades the pursuers} \\ 10t & \text{if } \mathbf{E} \text{ loses the game at time } t. \end{cases}$$

Using this payoff function, rule strengths are updated using the profit sharing plan described by Grefenstette (1988). 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.

After each game, a rule is selected for mutation, using *fitness proportional selection* (Goldberg 1989). Once selected, the rule is mutated according to a fixed mutation rate, and each clause or action of the rule is mutated according to a fixed mutation probability. Mutation results in the actions or bounds being changed at random, keeping the ranges consistent (i.e., if the lower bound is changed to be greater than the upper bound, then the bounds are swapped). The mutated rule then replaces the rule with the lowest strength in the same plan. After selecting the rule to be mutated, we decided whether to mutate the rule using a

[1] While not a part of the genetic algorithm itself, the generalization and specialization operators are consistent with Grefenstette's implementation (Grefenstette *et al.* 1990).

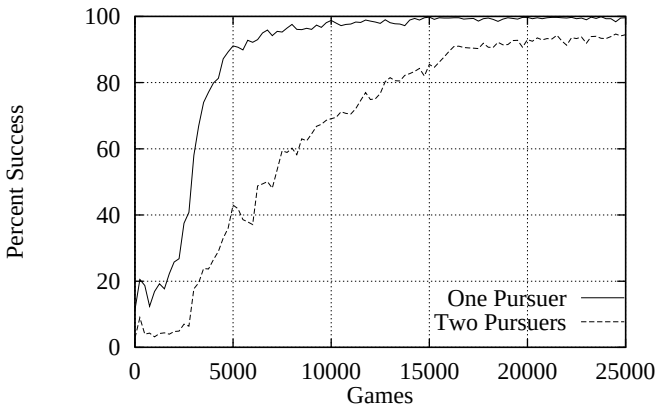

Figure 2: Performance of the GA.

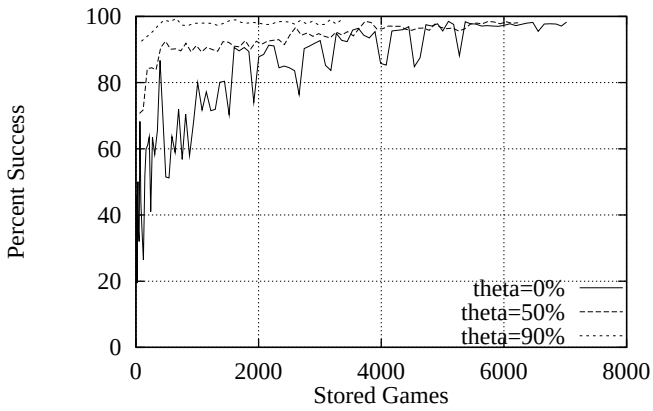

Figure 3: Results of MBR Using the GA as a Teacher.

mutation rate of 0.01. Each clause within a rule was considered for mutation with probability 0.1.

Crossover operates between plans. After each game, two plans are selected for crossover using fitness proportional selection, with a likelihood determined by the crossover rate. The rules in these plans are sorted by strength, and a new plan is generated by selecting m best rules from one plan and $(20 - m)$ best rules from the other plan. The new plan replaces the least fit plan in the population. Our crossover probability was 0.8.

We show the results of the GA experiments in Figure 2. As with MBR, the GA performs very well when faced with one pursuer. In fact, it is able to achieve near perfect performance after 15,000 games and very good performance (above 90%) after only 5,000 games. Note that 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. As a further basis for comparison, 500 generations for the GA on the one-pursuer problem required 1.5 days on a Sun SparcStation 2 (i.e., 16,667 games per day), while 5,700 games for MBR required 0.75 days (i.e., 7.600 games per day). For the two-pursuer problem, 500 GA generations used 3 days of CPU time (8,333 games per day) and 140,000 games for MBR required 21 days (6,667 games per day).

The most striking difference in performance between MBR and the genetic algorithm is that the GA learned excellent strategies for the two-pursuer problem, while nearest neighbor did not. Indeed, the GA achieved above 90% success after 16,000 games (320 generations) and continued to improve until it exceeded 95% success. This led us to hypothesize that the GA could provide a good source of examples for MBR. Thus, the GA became a "teacher" for MBR.

6 BOOTSTRAPPING MBR

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 can come back in. 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 MBR second. The communication or "teaching" phase occurs as follows. First, we train the GA until it reaches a performance threshold, θ. From that point on, the combined system begins providing test games to the GA. Whenever the GA succeeds, it transmits up to 20 examples (one for each time step) to MBR. After 100 test games, MBR is tested (to estimate its performance) with an additional 100 random games. The examples continue to accumulate as the genetic algorithm learns the task.

The results of training MBR using GA as the teacher on the two-pursuer task are shown in Figure 3. We call this system GANN because it first uses a GA and then uses a nearest-neighbor algorithm. 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 MBR from the beginning of its own training. The second threshold was set to 50% to permit GA to achieve a level of success approximately equal to the best performance of MBR on its own. Thus only plans that achieved at least 50% evasion were allowed to transmit examples to MBR. Finally, the threshold was set at 90% to limit examples for MBR to games in which a highly trained GA was making the decisions for **E**.

When $\theta = 0\%$, GANN almost immediately reaches a level equal to the best performance of MBR 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 games stored. Note that the number of games stored here is higher than the number of games stored for MBR alone. If we halt learning after 2,500 games (which is consistent with the earlier MBR experiments), performance would be in the 85% range, still an enormous improvement over MBR's performance, but not better than the GA on its own.

When $\theta = 50\%$, GANN starts performing at a very high level (above 70%) and quickly exceeds 90% success. After 2,500 games, GANN 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 MBR is probably not storing many "bad" examples. This confirms in part our earlier hypothesis that MBR'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 $\theta = 90\%$, GANN's performance was always superb, exceeding the GA's 90% success rate on its very first stored game. GANN converged to near-perfect performance with only 500 games. One striking observation was that GANN performed better than the GA throughout its learning. For example, when $\theta = 0\%$, GANN achieved 50–80% success while the GA was still only achieving 2–10% success. Further, GANN remained ahead of the GA throughout training. Even when $\theta = 90\%$, GANN was able to achieve 98–100% evasion while the GA was still only achieving around 95% evasion. Neither the GA nor MBR were able to obtain such a high success rate on their own, after any number of trials.

6.1 OTHER TEACHING STRATEGIES

Little other research has addressed teaching strategies for reinforcement learning problems. One approach that was investigated was Barto's ACE/ASE (Barto *et al.* 1983). This differs from the bootstrapping approach in that no feedback is provided to the GA to modify its learning algorithm. Further, ACE/ASE are both connectionist architectures whose weights are modified based on reinforcement received from experience. In our model, only the GA learns from reinforcement. Another related teaching method is that of Clouse and Utgoff (Clouse and Utgoff 1992), who used ACE/ASE with a separate teacher. Their teacher monitored the overall progress of the learning agent, and "reset" the eligibility traces of the two learning elements when the performance failed to improve. It then provided explicit actions from an external teacher (a kind of oracle) to alter the direction of learning. No external oracle exists in our bootstrapping method, and no modification of the learning process takes place.

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 (RP) to monitor the performance of the robot and provide feedback on performance. Learning occurs through the standard genetic algorithm applied to the classifiers with fitness determined by the RP.

6.2 REDUCING MEMORY SIZE

Our bootstrapping algorithm, GANN, performs well when only a small number of examples are provided by the GA, and it even outperforms its own teacher (the GA) during training. We decided to take this study one step further, and attempt to reduce the size of the memory store during the MBR phase of GANN. In the pattern recognition literature, a variety of algorithms for doing this can be found under the term "editing" nearest neighbors. However, because MBR is not frequently applied to control tasks (except within the context of reinforcement learning; see, e.g., (Moore 1992) and (Schaal and Atkeson 1994)), 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. We call the resulting system GANNE (GA plus nearest neighbor plus editing).

Early work by Wilson (Wilson 1972) showed that examples could be removed from a set used for classification, and that simply editing would frequently improve classification accuracy (in the same way that pruning improves decision trees (Mingers 1989)). Wilson's algorithm was to classify 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 (Tomek 1976) modified this approach by taking a sample (> 1) of the data and classifying the sample with the remaining examples. Editing then proceeds as in Wilson editing. Ritter *et al.* (1975) developed another editing method, which differs from Wilson in that points that are *correctly* classified are discarded. The Ritter method basically keeps only points near the boundaries between classes, and eliminates examples that are in the midst of a homogeneous region.

The editing approach we took combined the editing procedure of Ritter *et al.* and the sampling idea of Tomek. We began by selecting the smallest example set that yielded near perfect success in the pursuit game. This set contained 1,700 examples. Next we edited the examples by classifying each point using the remaining points in the set. For this phase, we used the 5 nearest neighbors. If a point was correctly classified, we deleted it with probability 0.25. (This probability was selected arbitrarily and was only used

Table 1: Results of Editing Examples.

Examples	Percent Success
397	96
216	94
118	94
66	92
41	89
21	86
11	83

to show the progression of performance as editing occurred.) Prior to editing and after each pass through the data, the example set was tested using MBR on 10,000 random games.

One complication in "classifying" the points for editing was that the class was actually a 3-D 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 affect on this range measurement.

The results of running GANNE on the 1,700 examples are summarized in Table 1. With as few as 11 examples, GANNE achieved better than 80% evasion, which is substantially better than the best ever achieved by MBR alone. With 21 examples (comparable in size to a plan in the GA), GANNE was able to achieve 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.

7 DISCUSSION AND CONCLUSIONS

The experiments reported here show that it is possible to use genetic algorithms in conjunction with memory-based reasoning 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 memory-based methods when applied to complex control tasks. Without a reliable source of good examples, our memory-based method (k-NN) performed very poorly at the two-pursuer game, but with the good examples provided by the genetic algorithm, it performed better than either the GA or MBR alone. In addition, we found that editing the example set produced a relatively small set of examples that still play the game extremely well.

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 theoretical minimum number of examples from which to learn the task.

Our current implementation only takes the first step towards a truly combined learning system. Our system has one algorithm starting the learning process, and handing off to a second algorithm to continue. 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.

Acknowledgments

We wish to thank David Aha, John Grefenstette, Diana Gordon, and Sreerama Murthy for several helpful comments and ideas. This material is based upon work supported by the National Science foundation under Grant Nos. IRI-9116843 and IRI-9223591.

References

D. Aha and D. Kibler. Noise-tolerant instance-based learning algorithms. In *Proceedings of IJCAI-89*, pages 794–799, Detroit, MI, 1989. Morgan Kaufmann.

D. Aha and S. Salzberg. Learning to catch: Applying nearest neighbor algorithms to dynamic control tasks. In *Proceedings of the Fourth International Workshop on AI and Statistics*, 1993.

C. Atkeson. Using local models to control movement. In *Neural Information Systems Conference*, 1989.

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

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

A. Barto, R. Sutton, and C. Watkins. Learning and sequential decision making. In Gabriel and Moore, editors, *Learning and Computational Neuroscience*, Cambridge, 1990. MIT Press.

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

J. Clouse and P. Utgoff. A teaching method for reinforcement learning. In *Proceedings of the Machine*

Learning Conference, 1992.

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

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

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

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

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

J. Grefenstette. Lamarckian learning in multi-agent environments. In *Proceedings of the Fourth International Conference of Genetic Algorithms*, pages 303–310. Morgan Kaufmann, 1991.

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

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

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

R. Isaacs. Differential games: A mathematical theory with applications to warfare and other topics. Technical Report Research Contribution No. 1, Center for Naval Analysis, Washington, DC, 1963.

L. Lin. Programming robots using reinforcement learning and teaching. In *Proceedings of the National Conference on Artificial Intelligence*, pages 781–786, 1991.

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

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

A. Moore. *Efficient Memory-Based Learnign for Robot Control*. PhD thesis, Cambridge University, Cambridge, England, 1992.

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

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

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

S. Salzberg. Distance metrics for instance-based learning. In *Methodologies for Intelligent Systems: 6th International Symposium, ISMIS '91*, pages 399–408, 1991.

S. Salzberg, A. Delcher, D. Heath, and S. Kasif. Learning with a helpful teacher. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 705–711, Sydney, Australia, August 1991. Morgan Kaufmann.

S. Schaal and C. Atkeson. Robot juggling: An implementation of memory-based learning. *Control Systems Magazine*, February 1994.

J. Sheppard and S. Salzberg. Sequential decision making: An empirical analysis of three learning algorithms. Technical Report JHU-93/94-02, Dept. of Computer Science, Johns Hopkins University, Baltimore, Maryland, January 1993.

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

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

C. Watkins. *Learning with Delayed Rewards*. PhD thesis, Cambridge University, Cambridge, England, 1989.

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

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