Evolutionary Reinforcement Learning: A Cyclical Approach

Scott H. Burton
sburton@thinkoriginally.com

Abstract

Evolutionary algorithms and reinforcement learning techniques are two popular approaches to solving optimization problems. This paper examines a way of combining these methods in a cyclical form such that each one can feed off the progress of the other. This offers a unique method of introducing diversity to help avoid finding local minima. The problem domain chosen herein is trivial, but the concepts could easily be extended for future work.

Introduction

Evolutionary algorithms and reinforcement learning both provide methods of solving optimization problems. In evolutionary algorithms, possible solutions are encoded in a way they can be transformed and compared with other possible solutions (Fonseca and Fleming 1995). Reinforcement learning is a method of allowing an agent to discover a solution without initial guidance. To accomplish this correct decisions are learned through punishment or reward by interacting with an environment (Kaelbling, Littman, and Moore 1996). This paper considers a study of combining the two algorithms in a unique way to determine whether mixing these techniques would offer improvements over either one by itself.

A common approach to combining these two methods is to use an evolutionary algorithm to determine the parameters to set for a reinforcement learning algorithm, or vice versa (Moriarty, Schultz, and Grefenstette 1999)(Pettinger and Everson 2002). Another approach is to simply let the evolutionary algorithm evolve the optimal policy. For this project, the algorithms were layered, letting each one build off what the other had found.

Problem Definition

The problem domain used for this comparison was the "Racetrack" example from Section 5.6 in (Sutton and Barto 1998). In this example, an agent attempts to make a right turn around a race track while only being able to slightly vary its velocity.

The racetrack consists of a discretized section of the right turn of a race track broken in an amount of squares on the order of 500. The velocity is also discrete, a number of squares to move per time unit in either the horizontal or vertical direction. The velocity in either direction can only be altered by -1, 0, or $+1$ each step. This gives a total of nine actions $\{(-1,-1),(-1,0),...,(1,1)\}$ per state. The maximum velocity is 5 in either direction, and cannot be negative. The problem is episodic starting at one of several places on the starting line and finishing once the agent crosses the finish line. The rewards are a small cost amount for each step (0.1), and a larger cost (10) when the agent tries to leave the track. The goal is to minimize the cost. Actually leaving the track is not allowed. See Figure 1.

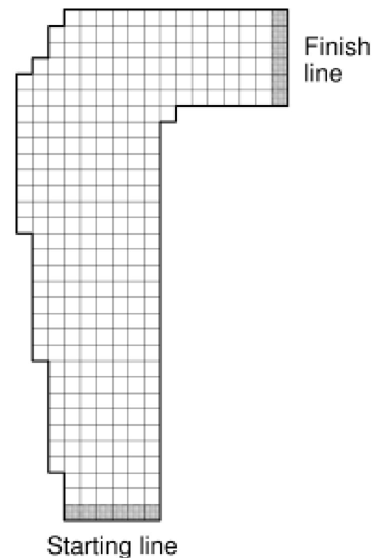

Figure 1: A simple racetrack. (Sutton and Barto 1998)

Approach/Methods

The hybrid approach is as follows:

1. Generate a population of random policies.
2. Use a simple genetic algorithm to evolve the policies for a certain amount of iterations.
3. Select the best of the population.
4. Starting with this best policy, use a reinforcement learning technique, $TD(0)$, and allow it to run for a certain amount of episodes.

Copyright © 2010, Scott H. Burton. All rights reserved.

5. Create a population based around the output of step 4.

6. Repeat steps 2-5 until some convergence limit is reached.

A simulator was used as the environment for the agent, moving it to the correct next state and issuing a reward.

Each state consists of the square the agent is in, and the current velocity in each direction. This gives ˜500 (number of squares) * 5 (current velocity possibilities in one direction) * 5 (velocities in the other direction) = ˜12,500 possible states. Each of these is represented in a tabular policy.

For the simple genetic algorithm (*SGA*), the following methods were used:

- *Encoding Scheme*: The tabular policy is represented as a chromosome, with each action for a state corresponding to a gene.
- *Mutation*: An action is changed to a randomly selected one.
- *Crossover*: Uniform crossover is used, where for each gene in the chromosome a random number is chosen. If it is greater than a threshold, the corresponding genes of the parents are switched, otherwise they are not.
- *Selection Method*: Tournament selection with a recombinative hill climbing replacement scheme, where children only replace parents when better.
- *Fitness Function*: The fitness of each policy was initially computed using the following: $F(\pi) = \sum_s V^\pi(s)$.

In order to compute the value of each state, some technique would have to be used such as policy evaluation. Because of the cost involved with computing the complete V table with policy evaluation, a more efficient approach was to run a simulation from each of the start states the sum of which is the fitness.

To create a population based around a specific policy (step 5 above), a specific policy was taken, cloned, and mutated to create each of the N chromosomes with a mutation rate of n/N, where n is the index of a specific chromosome and N is the population size. Thus the first chromosome will have a mutation rate of 0% (exact clone) and the last will have a mutation rate of 100% (random clone), and with each member of the population becoming more random than the previous one.

For comparison purposes, the *SGA* and the *RL* techniques were run on the problem separately, and the results compared.

Comparison Methods

Each of the three algorithms examined (*SGA*, *RL*, *hybrid*) was evaluated and compared by plotting the sum of the cost of a run starting from each of the states on starting line, versus time spent in computation. This gives the ability to see which of the methods converges most quickly to a good solution, and which finds the best solution overall. Using computation time rather than counting iterations gives a more fair comparison between algorithms that have vastly different processes taking place in each iteration.

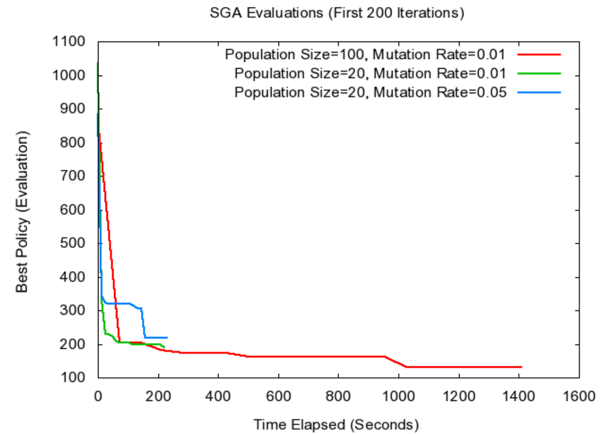

Figure 2: Simple genetic algorithm evaluations for the first 200 iterations. Because of the parameter differences, the runs took different lengths of time to complete. The largest population size led to the best results after a fixed 200 iterations, but took significantly more time.

Empirical Study

All results given in this document are actually the average of multiple runs to account for statistical outliers. The terms “run” and “trial” used herein will therefore actually refer to an average of multiple trials. All reported trials have been run on the same Pentium D processor with two 3.00 GHz cores, and 2 GB of RAM.

First, the SGA was run with several different parameter configurations. The main parameters that could be varied were the mutation rate, population size, and maximum number of iterations. Figure 2 shows the results for three configurations. The number of trials was limited in this case to 200 iterations. Because the parameters are different (most notably the population size), the runs took different lengths of time to complete the same amount of iterations.

It should also be noted that because the selection approach is a form of hill climbing, the functions are all nonincreasing.

The *RL* solver was also run with varying configurations. Running this experiment introduced an interesting problem. One of the great advantages *RL* has over *SGA* in this problem domain is that it does not have to go through the time consuming process of computing an evaluation of the policy each step of the way. This is necessary for the *SGA*, because evaluation is required to compare policies and therefore improve. Alternatively, the *RL* algorithm learns by simply updating the steps of the policy. The challenge is that in order to print out any sort of meaningful evaluation of the algorithm, it is necessary to compute and evaluate the policy from the Q table that is being learned.

This being the case, in order to save the best policy found to date, it would also be necessary to compute and evaluate the policy. Because of this, a conservative approach was used in how often to compute the policies for reporting and the saving of the best policy. The results of the *RL* solver are shown in Figure 3.

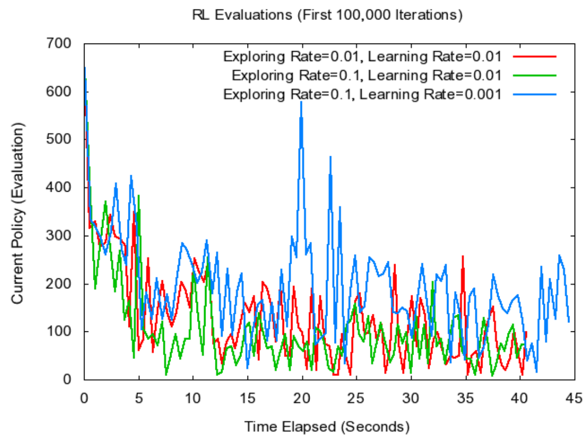

Figure 3: The learning process of the reinforcement learning agent for the first 100,000 iterations. It should be noted that, unlike with the *SGA*, due to exploration, a plot of the *RL* current policy is not a monotonically decreasing function.

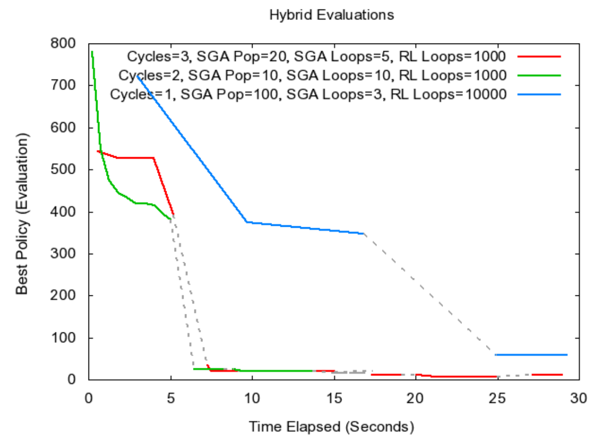

Figure 4: The learning process of the Hybrid solver. The switching between algorithms is shown with a dashed line and resulted in a large increase in progress. The parameters used for the different algorithms were: Mutation Rate=0.01, Learning Rate=0.01, Exploring Rate=0.1

Finally, the hybrid algorithm was also evaluated with several different configurations, with the results shown in Figure 4. The breaks in the line show the switch between the algorithms. Each setup/cycle began with the *SGA* solver and standard parameters (Mutation Rate=0.01, Learning Rate=0.01, Exploring Rate=0.1) have been used for the *SGA* and *RL* algorithms in the hybrid approach.

As can been seen from Figure 4, even though the hybrid algorithm was only allowed to "cycle" through the algorithms a very small amount of times (1, 2, and 3), it still had great improvement on performance.

Analysis

The results of the empirical study showed that, as expected, the genetic algorithm performed somewhat poorly due to the large state space to be encoded. It made reasonable progress, but got stuck in local minima in which it could not easily overcome. In addition to this limitation, the evaluation of the policies required for comparison caused it to take a very long time in computation, especially if the size of the population was significant.

Alternatively, the *RL* algorithm did fairly well on the problem and within 30 seconds could easily arrive at a very good solution. This was not surprising as the problem choice came from a text on Reinforcement Learning.

The hybrid solution was clearly the most effective of the three, as it could find near optimal solutions easily within 10 seconds of computation time. This significant improvement was the result of different factors. First, while the typical *RL* solution started with an arbitrary policy, in the hybrid approach, the starting policy for the *RL* piece was already on the right track. This is similar to doing a random restart a few times until a good solution is found. But it is better than simply doing a random restart because it mixes the good pieces of many random restarts. In this specific problem this has proven to be a very effective technique.

In addition, the *SGA* was ability to inject additional diversity to the *RL* algorithm, aiding it in exploration and giving it improvement over the algorithm by itself.

Conclusion and Future Work

This approach definitely holds merit in helping to "jump start" an *RL* solver, because starting with a fairly good policy allows the *RL* technique to simply do fine tuning, whereas the typical technique forces the learner to first do very coarse updating followed by fine tuning. This also helps to eliminate the need for adjusting the learning rate throughout the process.

This paper could easily be extended into future work. The most important thing that could be done is to apply this technique to a more meaningful problem and see how it performs. This would likely involve encoding the policy as a neural network or some form of function approximation rather than a table, because the size of the table would very quickly become very large. This might even assist the *SGA* process in evaluation, because it may remove the need to continually do a complete policy evaluation to determine fitness.

Also, the effects of this technique should be studied on problems with different types of solution spaces. For instance, it may be that the technique is very effective in a solution space with a single minimum, but more difficulty could result with increased numbers of local minima. It would be interesting to observe how well the technique would perform on classically "difficult" domains for reinforcement learning agents such as the Mountain Car problem.

References

Fonseca, C. M., and Fleming, P. J. 1995. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation* 3(1):1–16.

Kaelbling, L. P.; Littman, M. L.; and Moore, A. P. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.

Moriarty, D.; Schultz, A.; and Grefenstette, J. 1999. Evolutionary algorithms for reinforcement learning. *Journal of Artificial Intelligence Research* 11:241–276.

Pettinger, J. E., and Everson, R. M. 2002. Controlling genetic algorithms with reinforcement learning. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, 692. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Sutton, R., and Barto, A. 1998. *Reinforcement Learning*. MIT Press.