Abstract

Every day, we make decisions based on learning the results of our actions and those around us. For example, if we see a person sink deep into a seemingly shallow puddle, we will walk around it. This intuition is, for the most part, alien to intelligent, multi-agent systems, who can react based only on the results they have achieved and analyzed through testing and the environment around them. When the agents can take the results of others in similar circumstances under advisement, the learning in these autonomous systems will progress much faster than if each agent had to go through the rigorous test-retest procedure.

This flexibility excels when it is combined with a learning approach such as a Genetic Algorithm, where many agents with different, adapted responses are used to react to similar stimuli. When you can take into account the success of these responses individually, the system will arrive at the goal state much faster than with traditional methods.

I was primarily concerned with the use of the goalie’s genetic and memetic learning and the fundamentals (pass to, kick to, etc.) to be used by the rest of the players, while Brian used his knowledge of traditional AI strategies to make the other positions reasonable opponents for the user, in addition to the majority of the communication, message parsing, environmental representation, etc. required for functionality of a project of this magnitude. Although these were our focuses, we often asked each other for assistance with the other’s functions, as well as general aid when we were bogged down.
Introduction

Many computer scientists around the world recognize the various forms of Robocup Soccer Leagues as a cornerstone of multi-agent systems, and it is those researchers who participate who are often on the cutting edge of artificial intelligence research. Various methods of learning and other artificial intelligence are employed. These include neural nets, machine learning, genetic algorithms, finite state machines, fuzzy logic, hard coding, and other methods and combinations thereof. This pursuit on a simulated soccer field breeds innovation among the experts in the field and can be a great learning experience for those involved.

Currently, there are five categories of competition: four-legged (featuring the Sony Aibo dog), small-size, medium size, humanoid, and the one in which we are participating, simulation. We chose simulation because it was the only way we could participate in this exciting area of research without enormous time and monetary investments.

The Robocup Simulation League uses a rather efficient method of ensuring that the teams are in fact multi-agent systems, a server with player connections over a network (using TCP/IP or UDP/IP). This allows efficient access to the game world, and guarantees both teams that they will be receiving the same game-state information, as well as the knowledge that the opposing team is not sharing data between its players.

To watch these events, humans must use one of the monitors designed by the Robocup people, or design their own. These are traditionally two-dimensional representations of the field, players, and ball with minimal detail. We chose to elaborate on the ordinary method of displaying the game, and design our own, allowing direct interaction by the user, rather than limiting them to a spectator.

We also created the clients, and integrated them into the monitor for a friendly interface for those wishing to modify the AI or simply play a game of soccer. The AI itself is a combination of flock behavior, genetic and memetic algorithms, which will be described later.

Components

The Robocup system consists of three parts: Server, Monitor, and Clients. Our research focused on the client and the monitor, which were both modified heavily, but the server was left for the most part unaltered. We combined these with the Modeler program described below to display graphics, and finally into a game.

- **Server** - The Robocup Server is host to a game of soccer with a maximum of eleven players per team. The server is where the game world exists. It is here that the physics and rules are assessed to create as realistic a two dimensional soccer game as is possible. It interacts with individual clients over a network so that the computer hosting the game need not be burdened with the often complex, processor intensive artificial intelligence computations required for the more intricate teams which compete in the tournaments.

  Only 2 minor modifications were made to the server. One of which was the reduction of stamina requirements for the players, both to improve game play
and to assist in learning. The other was the acceleration of the game world. We reduced the server’s cycle time from 100 milliseconds to 15 (an easy 666% boost in learning speed). This allowed us to train the goalie in 2 hours, rather than the better part of a day.

• **Trainer** - The server can operate in two modes: regular and what amounts to a practice session. It was in this practice session that we used our learning algorithms to train the goalie to intercept the path of the ball from various directions. The controller of this session is what is called the trainer, which can control almost any aspect of the game state.

  Our implementation utilized his ability to place the ball and the goalie precisely where we wished, so we could cycle through tests as quickly as possible without having to reposition the ball and kick it using another player. With this method and the accelerated server, we were able to reduce the time for a generation of twenty children to run forty tests each in less than three minutes.

  We placed the ball randomly along the line of the penalty-area and shot the ball in semi-random directions towards the goal. I believe that this method allowed for as much variability in goalie situation as possible while maintaining the validity of the tests.

• **Monitor** - We decided to base our monitor on the core of the Modeler application provided by Prof. Dorsey. This core allowed us a great deal of freedom while maintaining stability and simplicity. We benefited greatly from establishing our monitor upon a system with which we were already familiar with. Also, the Faster Than Light Tool Kit (FLTK) allowed us access to many simple implementations of complex features for maximum functionality with minimum programming overhead.

  The first step in developing a monitor is initiating communication with the server. Without information about the state of the game, there would be little sense looking at a three-dimensional representation of the field. We applied Robocup server’s structure dispinfo_t to this end, and created the connections for the monitor, players, and the trainer using Winsock TCP/IP for receiving positional and game state data as well as commands to the server.

  We then integrated all these connections so that we could have control of all the entities in the world from within the monitor code. If we didn’t have this total control, we would not have been able to allow the user to take control of more than one player. In addition to convenience, our monitor program also handles the timing for control. This means that the AI’s or player’s commands will only be sent to the server when a command can be issued.

  Also, incorporating the learning allowed us more freedom and an easier interface with which to work. With this implementation, we could use FLTK to alter the testing conditions without recompiling.

• **Client** - As aforementioned, we integrated the various clients into our monitor so that the player could take control of the players at will, during which the AI would be disabled. We decided to use various methods of control for the different
players. Most the players are controlled using a combination of flock behavior and hard coded behaviors. This allows for fluid movement while maintaining the precise behavioral control necessary for a team to be successful.

When the user pushes one of the number keys, the artificial intelligence for that player is ignored, and the player is controlled by a number of keys on the keyboard. With this interface, we were able to attain near total control of the game state for both the user and developer. Everything from positioning to play state can be controlled in this fashion.

- **Graphics** - In addition to the learning and integration of the above features, we also created a three dimensional display of the game. To do this, Brian created a series of classes and objects to contain the data from the server and describe the objects to the renderer.

- **The Game** - While this project is an experiment, the monitor we designed was intended to be a game. To begin this game, the user must initiate the soccer server on his or her computer (“localhost”) so the monitor and clients have something to connect to.

  Once the connection is established, the players on the field are initialized to random positions, but using the “move” command, you can place them wherever you want before the ball is kicked. In addition to the basic move commands (WASD—movement, ERFV—various kicks), you can switch players using the number keys (non-num pad), and adjust the camera using ‘[’, ‘]’, ‘;’, ‘’ (apostrophe), ‘,’ (comma), ‘.’ (period), ‘/’ for a number of camera angles you would not ordinarily find in most soccer games.

  To begin the game, press the `k` key, and after that, the field is yours.

**Related Research**

Genetic algorithms allow the programmer to arrive at a goal state with little input other than the test state and a set of random values to initialize the control and a function that describes the success or “fitness” of the agent’s behavior. However, with this flexibility comes a long learning process and uncertainty about the quality and relevance of the fitness function.

To use an evolutionary algorithm (genetic algorithm) to solve a problem, you must first describe your goal. In this case, we wanted to have a goalie who will intercept the ball when kicked from a variety of positions at different angles. To do this, you base success on the interception of the ball, and failure on a goal scored. As simple as this premise sounds, it is difficult to tell, and impossible for the genetic algorithm’s selection process to determine which decision or decisions directly resulted in success.

Memetic algorithms can reduce the ambiguity of selection by use of what is known as eligibility. It is assumed that the importance a decision made when the ball was close to the goalie is more important than the first decision made when the ball was twenty steps away. Therefore the relevance of the later decisions is taken into account, and the fitness no longer rests on the player, but on the success of his decisions. This is done by decaying the eligibility (relevance) of a decision as time progresses towards the state in which the success or failure of the agent is assessed, so that when judgment time
is at hand, the confidence of a decision’s contribution to the success or failure of the test determines its likelihood to be reproduced in a genetic fashion.

With this quantization of the importance of decisions, beneficial actions contained in the genotype will be preserved, and unfavorable decisions will be omitted. For example, if an agent (Agent 1) has all but a few good genes, chances are those maladaptive ones will be perpetuated (they will succeed at most, but fail during the problematic stimuli). However, if the specific features of each gene are taken into account, the genes of another agent (Agent 2) who succeeded where Agent 1 failed would most likely be selected, allowing for a more rapid approach to the goal state.

Method

For simplicity, the goalie itself functioned in two states: dash and turn, and during either it could interrupt to catch the ball. The value from the learned behavior is used during the turning phase.

For the both algorithms, we used standard roulette wheel selection to choose the parents. In the genetic algorithm, this means that each player has a chance of selection proportional to his success compared to the success of the other members of that generation. The members of the generation in the memetic algorithm were selected in two different ways. In the first, the member was selected by the overall confidence assigned by the evaluation function:

\[
\text{Confidence [allele]} = \text{Eligibility [allele]} \times \text{result_value};
\]

where result_value is the value assigned to the outcome of the test.

Our fitness values were as follows:

- Missed ball = -.25
- Ended outside Goal = +.25
- Block = +3

The second selection method chose what information was to be carried on differently. To create parents, the selection function used roulette wheel selection to choose alleles based on their confidence rather than the whole gene. This way, the parent’s gene is composed of the more confident alleles, possibly making for a better member in each case.

Eligibility is how memetic algorithms weight the importance of the actions they decide upon. Memetic learning assumes that earlier decisions are less likely to affect the outcome than those made in the moments immediately before.

The decay function:

\[
\text{Eligibility [allele]} /= \text{pow (2, (int) (timeDiff / HalfLife))};
\]

reduces the importance of decisions made earlier so that later decisions are weighted higher.

Although the second selection method for memetic learning allows for improvement in target areas, the implementation of a memetic algorithm is much more difficult than that of the evolutionary algorithm because it leaves much less room for error than genetic algorithms. For example, if the decay function is off, your selection function puts emphasis on the wrong decisions, and the learning system fails (as described below).
Results

Initially, we were having difficulties receiving positive results with both learning algorithms. Upon examining the data, we realized our algorithm had a number of weaknesses.

The first, and most peculiar, was the effect of a fast computer on seeding a random number. With the learning algorithm running, our monitor peaks at 10 - 15% (3-5% vs. steady 50%) of the average processor time required by the two-dimensional windows monitor we found (Tsinghu Aeolus) on a 1.3ghz computer. While I’m proud of this fact that our program runs well, the problem was that in seeding the random number for each parent selected with the time, there wasn’t sufficient delay for the return of the time function to change between iterations. Thus the random function was being repeatedly seeded with the same number, so the same parent was being selected twice.

Another problem was the lack of consistency in the results as the algorithm progressed. The first place we looked was the angles generated as a result of the learning, and found that there was little consistency between generations and between similar sets of angles. This led me to believe that our testing was unreliable, so the first things I did were implement Elitism and increase the test number as we started out with ten members per generation doing ten tests on each. I first increased the number of tests to twenty, and it was an improvement, but not what we were going for (figure 1).

Once these fixes were in place, the test became more reliable among Elites between generations, but the rest of the members of a generation were still slow to converge, and at some points divergent. To remedy this, we increased the number of members from ten to twenty, at which point the convergence stabilized (figure 2).

Once the genetic algorithm stabilized, we achieved an average fitness of 67 and a peak fitness of 93 when using a generation size of 20, with 40 tests per child, and elitism enabled (figure 2).

Fitness is calculated as follows:
- Missed ball = -.25 fitness
- Ended outside Goal = +.25 fitness
- Block = +3 fitness

To get scores of 67 and 93, the goalie would need to block at least twenty-four shots (balls go outside posts rarely) out of forty (60%) and thirty-two out of forty (78%), respectively.

Discussion

The evolutionary algorithm worked out well once the bugs were ironed out. When proper generation size and test reliability were achieved, the performance of the goalie was surprisingly good given it reached a plateau of 80% after only 10 generations.

Although the results from the genetic algorithm were excellent, through all our efforts, I was unable to get the memetic algorithm to function well. Each test was less than twenty cycles long (15msec each), and it was difficult to apply and decay eligibility to such a short test. The half-life functions are designed for much longer tests, so they
did not work well for this application. Some benefits in performance were obtained, but there was not a statistically significant difference between the random angles assigned at initialization and the average performance of the system.

**Future Research**

For many years, Robocup will be a challenge for developers of multi-agent systems. I believe that given a more complex set of behaviors or better training system, a memetic algorithm would excel in this application. The ability to judge relevance of an action allows the system to develop faster and more efficiently than just by the overall success of a series of tests.

In addition, the interface with the server that we designed is a more attractive, customizable, and efficient version than those currently available. It would provide the perfect basis from which to continue work to refine and add to the behaviors discussed above for persons of any ability level in either graphics programming or artificial intelligence.

My future goals for the project are to get memetic learning working in its current setup, and implement it in areas of the control system other than goalie-ball-interception. I believe it would also work very well in other basic applications such as passing and shooting, and even more complex behaviors.

Also, I would like to merge the goalie artificial intelligence with the rest of the team, because right now it is an independent entity. Also, a focus on the rest of the team’s intelligence would do the game justice, as I neglected it to try to get memetic learning working on the goalie.
Figure 1 – The genetic algorithm is stable, but still not reliable with only 20 tests, hence the sporadic nature of the best player’s fitness (top line).

Figure 2 – A nice, steady (for the most part) climb to 80%.
References

- TsingHua Robocup Team. http://robocup.lits.tsinghua.edu.cn/