Evolving Team Darwin United

David Andre and Astro Teller

dandre@cs.berkeley.edu   astro@cs.cmu.edu

University of California at Berkeley, Berkeley, CA, 94720-1776
Carnegie Mellon University, Pittsburgh PA 15213

Abstract. The RoboCup simulator competition is one of the most challenging international proving grounds for contemporary AI research. Exactly because of the high level of complexity and a lack of reliable strategic guidelines, the pervasive attitude has been that the problem can most successfully be attacked by human expertise, possibly assisted by some level of machine learning. This led, in RoboCup'97, to a field of simulator teams all of whose level and style of play were heavily influenced by the human designers of those teams. It is the thesis of our work that machine learning, if given the opportunity to design (learn) "everything" about how the simulator team operates, can develop a competitive simulator team that solves the problem utilizing highly successful, if largely non-human, styles of play. To this end, Darwin United is a team of eleven players that have been evolved as a team of coordinated agents in the RoboCup simulator. Each agent is given a subset of the lowest level perceptual inputs and must learn to execute series of the most basic actions (turn, kick, dash) in order to participate as a member of the team. This paper presents our motivation, our approach, and the specific construction of our team that created itself from scratch.

1 Introduction

Imagine a group of human programmers attempting to create a team for RoboCup-98. They have to painstakingly design and code a complex set of routines and procedures to create a viable team. Not only do they have to think on the level of strategies, but also on the level of complex behaviors to achieve simple tasks. They have to learn the details of the simulator, not just as it ought to work, but as it actually works when these two are not the same. These human programmers have to try to not only impart their own soccer experiences to these procedures, but also to imagine ways in which the simulator differs from the real world and so too should the individual player and team strategies. On top of all this, the human programmers of such a RoboCup-98 team must design all this under the changing conditions of new rules, new simulator characteristics, and all of the noise and sensory limitations built into the fundamental model of the simulated world.

If a technique existed that allowed all of these issues to be solved automatically, such a technique would have certain advantages in the competition. In addition, such a technique would also be of immediate interest outside the soccer simulator domain, a claim that is more difficult to make for hand-coded solutions to the soccer simulator domain. It has been repeatedly been asserted that this problem is just too difficult for such a technique to exist.

> Because of the complexity of the [soccer server] domain, it is futile to try to learn intelligent behaviors straight from the primitives provided by the server.
>
> – From [Stone and Veloso., 1998]

The motivation for team Darwin United was to demonstrate that this claim is not true and that, in particular, genetic programming can be used successfully as a technique for training a team using the basic percepts and actions of the simulator. Luke's impressive results (1997a) at RoboCup-97 showed that genetic programming could tackle the task of generating competitive strategies, given a suite of hand-coded complex, low-level behaviors. Obviously, it is desirable that a technique for program induction work on multiple levels when applied to difficult problems. Thus, our goal has been to evolve a team to compete in the simulator league at RoboCup-98 in Paris that attacks the problem "from the ground up." This paper discusses our research and overviews the techniques that we are using to achieve the stated goals.

Of course it seemed highly unlikely that we would get superlative behavior on a known-to-be difficult problem without some modification to the standard genetic programming (GP) paradigm. How could GP solve the complex problem of designing teams for RoboCup given only the final score of a game? As this seems next to impossible, our solution is to replace the suite of hand-coded behaviors with a complex fitness function that provides reward for good play even when no goals are scored. As Koza, Andre, Bennett, and Keane have noted in their research on evolving analog circuits [Koza *et al.*, 1997], *it is significantly easier to write specifications for complex behavior than to write the programs to achieve the behavior.* As part of the specifications, we introduce a graduated fitness function that tests each individual for increasing levels of skill. For example, before evolving teams are allowed to compete with one another for representation in the next generation, they must first pass three *competition filters*, the first of which is simply the ability to score on an empty field. This graded set of fitness cases is described in Section 2.

Additionally, although we would like GP to solve the entire problem of team soccer play, it turns out that GP is remarkably slow to learn generalizable routines to reliably run to and kick the ball when given only the most basic of primitives. Thus, we give each of the teams in the initial generation a set of automatically defined functions (ADFs) [Koza, 1994] that encode some simple functionality such as running to and kicking the ball. These subroutines are very simple and highly non-optimal.

At first, these subroutines seem to violate the basic premise of our work, that we want Darwin United to learn to play soccer in the simulator in a style all its own. Two factors when taken together, show that this is still the case. The first is that a total of perhaps 2 hours of human time was spent creating these ADFs. Relative to the time usually spent creating such procedures in human designed solutions to the RoboCup'98 domain, this might as well have been 2 minutes. More importantly, these ADFs (parameterized subroutines available to the entire team) are subject to evolution as are each of the players. This means that these basic skills are greatly changed and refined during the learning process. The hand-created solutions are simply seeds from which the learning process begins.

2 The Coach: Specifications vs. Programming

In general, we don't want to tell our system how to solve the problem. However, if our system used nothing other than "scoring goals is good and scoring more goals than the opponent is great", the system would have an almost insurmountable learning problem. The reason that it is so difficult is that scoring even once is a needle in a haystack search problem. Our solution to this is to write a list of *specifications* that describe the desired behavior.

There are two ways in which the specifications are ordered. First, we specify that each team is to play in a graduated series of games, where if the team's performance in any game is too poor, it does not proceed to later games and gets a maximal value of fitness for those games that it does not play (in GP, lower fitness is better). This technique limits the amount of time spent evaluating poor teams and is reminiscent of many previous techniques for focusing computation on individuals with reasonable chances of success (e.g., [Teller and Andre, 1997]).

Second, within each game, there is a ordered list of scores where each score is a kind of success in the game, and each successive element in the list is more important than all the previous elements combined. Scoring more than the opponent is the last, and therefore most important element of the list. The lexical ordering is achieved by scaling each element to a single place value in a decimal score. In other words, the smallest score is scaled to be less than a thousandth, the next is scaled to be less than a hundredth, and so on. It is important to repeat that lower fitness is better.

We now describe the elements of the scoring list from least important to most important.

Getting Near the Ball Each player gets a value of 1.0 to begin with (given that its initial distance from the ball is X) and can reduce this value only by moving at least once and seeing the ball at distance X' (X' < X). The new value is set to (X'/ X). In essence, this encourages getting close to the ball. This factor is scaled to be equal to or less than a thousandth.

Kicking the Ball Each player gets an initial value of 0.5. This value can go as high as 1.0 if the player repeatedly has a chance to kick the ball and doesn't, and can go as low as 0.0 if the player repeatedly kicks the ball and sees it roll some minimum distance (just out of kicking range in this case). This score factor encourages kicking the ball and penalizes not kicking it when it is kickable. This factor is scaled to be equal to or less than a hundredth.

Sides Each player on a team gets a value that expresses the amount of time that the ball was on their half. This value can be as little as 0.0 if the ball is spends most of its time adjacent to the opponents goal, is about 0.5 if the ball is usually near the midline and near 1.0 if the ball spends most of its time near the team's own goal. The actual value is an integration of the X value of the ball as sampled repeatedly during the game where X is 0 at the halfline. This factor is scaled to be equal to or less than a tenth.

Being "Alive" Each player receives a penalty of 1.0 unless it turns at least once and runs forward at least once during the game in which case it receives a score of 0 for this factor.

Scoring a Goal Each player on a team receives a negative fitness point for each goal scored. This bonus is limited to 10 points.

Winning The Game Each player on a team receives 10 fitness points for a win, 30 points for a tie, and 40 points for a loss.

The following point can not be overstated. Because of this lexical dominance (each successive element in the list being more important than all previous elements combined), **once a team gets the hang of scoring goals against an opponent, none of these other factors has any appreciable effect on fitness.** Since winning is the only real metric for success, it should be (and is here) the final and dominant measure of fitness. This technique allowed us to help Darwin United get up to speed in the domain, but then allowed it to create its own unbiased solution, and all this without a human ever writing detailed code to teach Darwin United how to play.

Thus we can obtain a score for each game that a team plays in. If the team does well enough in a given game, it can move on to play a more difficult game (or set of games). A team's total fitness is an average over these separate game's measures. The effect each team has on the evolutionary process is directly determined by how this fitness score it receives for its play compares to the fitness scores of the other teams in the population.

Each team in the population follows the following schedule. First, the team is tested against an empty field. It passes this test if it scores within 30 seconds, and fails otherwise. Although this sounds easy, it requires a bit of evolution to obtain a team that reliably dribbles and shoots the ball into the opposing goal, rather than shooting out of bounds, towards one's own goal, etc. Second, the team plays against a hand-coded team of "kicking posts" – players that simply stay in one spot, turn to face the ball, and kick it towards the opposite side of the field whenever it is close enough. This promotes teams that can either dribble or pass around obstacles.

When a team scores against the kicking posts, it then plays the winning team from the 1997 RoboCup championships, the team from Humboldt University, Germany. Then, only if the team does well enough against the German team (i.e., scores at least one goal), is the team allowed to play three games in a tournament with other teams who have also made it through these three competition filters.

3 Team Structure and Team Transformations

This section provides an overview of how each team is represented in the evolutionary process and how teams are changed in the exploration phase of the machine learning (search) process.

Each team is composed of eleven distinct members. Each member is represented by an evolved program written entirely using the primitives shown in Table 1. Notice that all eleven team members in a team share the same set of ADFs. This set could have been of any size but was fixed at eight for the RoboCup'98 competition. Each team, however, has its own set of evolving subroutines. In this way each team can develop and share a certain style of play, but there is still diversity of these styles across the population of teams. This structure is shown in Figure 1.

In evolution, the two dominant forms of search operators are *crossover* and *mutation*. Crossover takes multiple (usually two) individuals, and exchanges "genetic material"

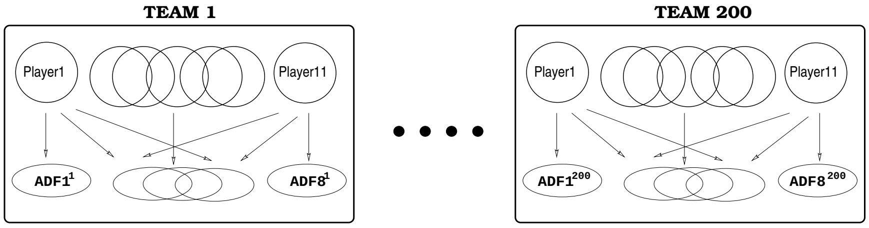

Fig. 1.: The population and team structures.

between them. Mutation typically selects some aspect of the evolving individual to be changed and replaces that aspects with a new, randomly generated piece of "genetic material." In this case, "genetic material" is the lisp-like code written using the primitives shown in Table 1.

Notice that there are no looping constructs in the primitives list for the evolving programs. This is because this evolved program for each team member is run *every 100ms* and returns one of three primitives actions (turn, dash, kick). So looping constructs are not necessary (though they might have turned out to provide additional benefit). Notice also that two of the primitives listed in Table 1 are READ and WRITE. These two primitive functions give the evolving programs access to an indexed memory. This memory is not cleared between executions of the program (every 100ms). As a result, it is this mechanism through which the evolving programs can learn to act in ways more complex that as purely reactive agents.

Now when a team is selected for "recombination" (a search step), either crossover or mutation is selected. If crossover is selected then a second team is also included with which genetic material can be exchanged. The details of how these transformations are done is not appropriate for this paper, but the key insight is that most of the time, when genetic material is exchanged between teams, genetic material from player 6 is exchanged with player 6 from the other team. This creates what is referred to (in both biology and in evolutionary computation) as a set of "niches." Niches tend to foster diversity of behavior, which is exactly what we desire of such a system. The goalie and the center forward should not act in similar ways, so it is appropriate that those two "types" of players rarely exchange piece of their code. A similar argument can be made for the niches of the different ADFs and so a similar constraint is put on the genetic material exchange between ADFs: it usually takes place with the similar numbered ADF from the other team.

Notice though that because Darwin United is evolved from scratch, there are not notions of "positions" (aside from the goalie). That is to say that if one player wants to (evolves to) play on the left side of the field, it will do this independent of whether a teammate is already in this space. No positions are enforced; even this must/can be learned as part of the evolutionary process. It has been our experience however that positioning is not something that tends to evolve in the way that humans enforce it. However, whether this is a deficiency in how evolution makes soccer players or whether it is a deficiency of how human's think about soccer is still an open question.

[1] The action is only available to players in the goalie position.

Inputs	**Constants**	**Memory**	**Calculate**	**Actions**	**Team-Shared**
Player.X.pos	Real Valued Consts	READ(x)	ADD	KICK(a, b)	ADF1
Player.Y.pos		WRITE(x, y)	SUB	TURN(a)	ADF2
DIST.To.Ball			MULT	DASH(a)	ADF3
DIR.To.Ball			DIV	GRAB[1]	ADF3
DIR.To.Goal			SIN		ADF4
DIST.To.Goal			COS		ADF5
Ball.DIST.Delta			IFLTE		ADF6
Ball.DIR.Dela					ADF7
T-DIST(a)					ADF8
T-DIR(a)					
O-DIST(a)					
O-DIR(a)					

Table 1.: The list of primitives for the evolving soccer players

It would initially seem reasonable to do "All Star" teams, that is, taking the "best" player from each “position” (player number) and making a team out of those individuals. To begin with, this is to imply that the best team is always composed of the best individuals. This is known not to be the case among human soccer teams and has repeatedly be shown not to be the case in distributed AI and multi-agent systems. Second, as has just been pointed out, there is no enforced relation between player number and position, so there is some danger that this process would selected “too many” forwards or “too many” defenders (though these descriptions are meaningless in an evolving system). Furthermore, each of the players of a team share certain behaviors (or parts of behaviors) through the common ADFs. This is a form of implicit communication through coordination of activity. There are a number of reasons to think that this will be useful in this domain and has certainly been useful in related domains (e.g. the predator-prey domain). Additionally, the relation of team members through the ADFs makes it much harder to separate them in a way that is likely to produce an all star team that has performance superior to that of any of the evolved teams.

4 Program Primitives for Evolution

To minimize the design time (which is, after all, part of the point of automatic programing) we use libsclient and give GP *some* of the inputs provided by libsclient. Specifically, each player has access to both the X and Y components of its position (XPOS and YPOS), its direction (DIR), the distance to the ball (BDIST), the angle to the ball (BDIR), the change in distance to the ball from the last time step (BCHNG), and the change in angle to the ball from the last time step (BDIRCHNG). Another libs-client value that is passed on the evolving players as an input variable is the distance and direction to the goal.

Additionally, the players have access to a function that can provide them information about the distances and angles to each of the other players that can be seen. T-DIST(a) is a one argument function that returns the distance to the player that is the 'a'th far away. If the argument is larger than or equal to the number of teammates seen, then undefined is returned. If the argument is below zero, the value of T-DIST(0) is returned. O-DIST(a) is similarly defined, but for the opponents. T-DIR(a) and O-DIR(a) are also similarly defined, but for the directions to the visible players. For any value that cannot be determined (such as the ball direction if the ball is out of view), a value of -999 is returned.

The function set for the players consists of a variety of functions for computation, memory, and setting the command to be executed. The computation include the two argument ADD, SUB, DIV, and MULT functions. Also, the one argument SIN and COS functions are included. The programs can use the four argument IFLTE function (if $x < y$ then u else v) as a conditional. Random constants are also included. Each player has access to its own array of indexed memory with 10 cells of memory. The one argument READ function allows the player to access the memory, and the 2 argument WRITE function allows the player to set the values of the memory cells.

The player also has access to three functions to set the command that will be executed on the next time step. KICK(a,b) sets the command to be a kick, with power a and direction b. DASH(a) sets the command to be a running movement, with power a. TURN(a) sets the command to be a turning movement with power a. The last of these functions to be executed is set to be the actual command issued to the server. There are also eight two argument functions ADF1 through ADF 8. For the ADFs, the terminal set additionally consists of the 2 zero argument functions ARG0 and ARG1 (i.e., "first parameter" and "second parameter"). All these primitives for the evolution of the soccer playing programs are shown in Table 1.

Acknowledgments

The authors gratefully thank Peter Stone and Sean Luke for advice and assistance throughout this project. David is supported by a National Defense Science and Engineering Grant, and Astro is supported through the generosity of the Fannie and John Hertz Foundation.

References

[Koza *et al.*, 1997] John R. Koza, Forrest H Bennett III, David Andre, Martin A. Keane, and Frank Dunlap. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*, 1(2):109–128, July 1997.

[Koza, 1994] J. Koza. *Genetic Programming 2*. MIT Press, 1994.

[Stone and Veloso., 1998] Peter Stone and Manuela Veloso. A layered approch to learning client behaviors in the robocup soccer server. *Applied Artificial Intelligence*, 12, 1998.

[Teller and Andre, 1997] Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.