Co-Evolving Soccer Softbot Team Coordination with Genetic Programming

Sean Luke
seanl@cs.umd.edu

Charles Hohn
hohn@wam.umd.edu

Jonathan Farris
jfarris@wam.umd.edu

Gary Jackson
garyj@wam.umd.edu

James Hendler
hendler@cs.umd.edu

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Genetic Programming is a promising new method for automatically generating functions and algorithms through natural selection. In contrast to other learning methods, Genetic Programming's automatic programming makes it a natural approach for developing algorithmic robot behaviors. In this paper we present an overview of how we apply Genetic Programming to behavior-based team coordination in the RoboCup Soccer Server domain. The result is not just a hand-coded soccer algorithm, but a team of softbots which have *learned on their own* how to play a reasonable game of soccer.

1 Introduction

The RoboCup competition pits robots (real and virtual) against each other in a simulated soccer tournament [Kitano *et al*, 1995]. The aim of the RoboCup competition is to foster an interdisciplinary approach to robotics and agent-based Artificial Intelligence by presenting a domain that requires large-scale cooperation and coordination in a dynamic, noisy, complex environment.

For RoboCup's "virtual" competition, players are not robots but computer programs which manipulate virtual robots through RoboCup's provided simulator, the RoboCup Soccer Server [Itsuki, 1995]. Players programs may not communicate with each other except through the limited "speech" provided them by the Soccer Server itself. The RoboCup Soccer Server's loosely distributed nature of agent coordination and dynamic environment make appealing a reactive, behavior-based approach to coordinating the soccer team. However, there are a wide variety of possible behaviors (even very simple ones), and the number of permutations of behavior combinations amongst eleven independent agents can be quite high. Instead of hand-coding these behaviors for each agent, it's attractive to have the agents *learn* good behaviors and coordination on their own.

Beyond its interesting AI and Alife aspects, getting agents to learn on their own can result in interesting solutions to the problem that hand-coding may overlook. The dynamics of the RoboCup soccer simulator are complex and difficult to optimize for. Given sufficient time, a learned strategy can evaluate a broad range of different behaviors and hone in on those most successful. However, many learning strategies (neural networks, decision trees, etc.) are designed not to develop algorithmic behaviors but to learn a nonlinear function over a discrete set of variables. In contrast, Genetic Programming (GP) [Koza 1992] uses evolutionary techniques to learn *algorithms* which operate in some domain environment. This makes it natural for learning programmatic behaviors in a domain like the Soccer Server.

Genetic Programming has been successfully applied in the field of multiagent coordination a number of times. [Reynolds, 1993] used GP to evolve "boids" in his groundbreaking work on flocking and herd coordination. [Raik and Durnota, 1994] used GP to evolve cooperative sporting strategies, and [Luke and Spector, 1996] and [Iba, 1996] used GP to develop cooperation in predator-prey environments and other domains). The bulk of this paper describes GP and how we use it to evolve coordinated team behaviors and actions for our soccer softbots in RoboCup-97.

2 Genetic Programming

Genetic Programming is a variant of the Genetic Algorithm [Holland, 1975] whose aim is to optimize, through pseudo-evolution, functions or algorithms ("individuals") to solve some task. The most common form of Genetic Programming is due to John Koza [Koza, 1992]. This form optimizes one or more LISP-like "program-trees" formed from a primordial

[0]To appear in the proceedings of The First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence (IJCAI-97), Nagoya, Japan, 1997.

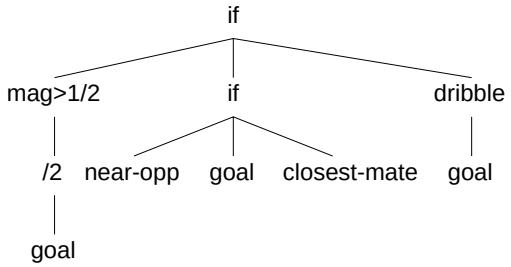

Figure 1: A Typical GP algorithm tree

soup of atomic functions. These trees serve both as the genetic material of an individual, and as the code for the resultant algorithm itself; there is no intermediate representation.

An example GP tree is shown in Figure 1. A GP individual's tree can be thought of as a chunk of LISP program code: each node in the tree is a function, which takes as arguments the results of the children to the node. In this way, Figure 1 can be thought of as the LISP code

```
(if (mag>1/2 (/2 goal))
    (if near-opp goal closest-mate)
    (dribble goal))
```

GP trees are executed as algorithms by running them in some program domain as if they were this LISP code.

Genetic Programming optimizes individuals very similarly to the Genetic Algorithm. The user supplies the GP system with a set of atomic functions with which GP may build tree individuals. Additionally, the user provides an *evaluation function*, a procedure which accepts an arbitrary GP individual, and returns an assessed fitness for this individual. The GP system begins by creating a large population of random individuals for its first generation. It then uses the evaluation function to determine the fitness of the population, selects the more fit individuals, and performs various breeding operators on them to produce a new generation of individuals. It repeats this process for successive generations until either an optimally fit individual is discovered or the user stops the GP run.

GP's breeding operators are customized to deal with GP's tree-structured individuals. The three most common operators we use are *subtree crossover*, *point mutation*, and *reproduction*. GP's mutation operator (shown in Figure 2) takes a single individual, replaces an arbitrary subtree in this individual with a new, randomly-generated subtree, and adds the resultant individual to the next generation. GP's crossover operator swaps random subtrees among two fit individuals to produce two new individuals for the next generation, as shown in Figure 3. GP's reproduction operator simply takes a fit individual and adds it to the next generation.

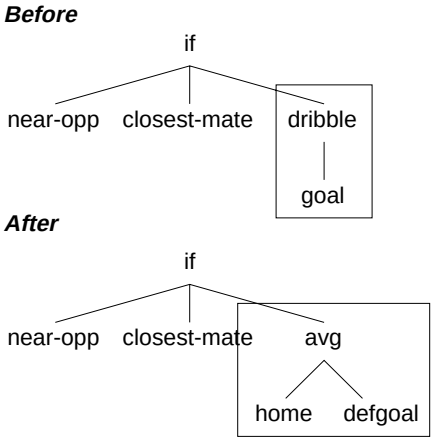

Figure 2: The point mutation operator in action. This operator replaces some subtree in an individual with a randomly-generated subtree.

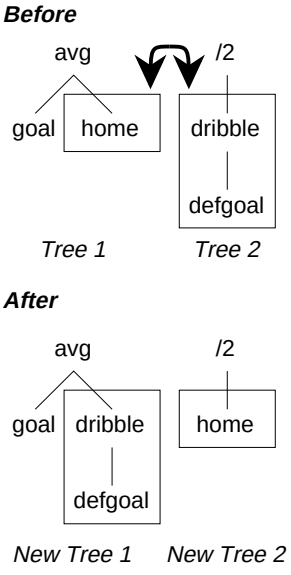

Figure 3: The subtree crossover operator in action. This operator swaps subtrees among two individuals.

Function	Returns	Description
(s1)	bool	Returns my internal state flag (1 or 0).
(mate-closer)	bool	1 if a teammate is closer than I am to the ball, else 0.
(near-opp)	bool	1 if there is an opponent within *r* distance from me, else 0.
(squad*n*)	bool	1 if I am squadmate *n*, else 0.
(rand)	bool	1 or 0, depending on a random event.
(home)	vect	A vector to the my "home position".
(ball)	vect	A vector to the ball.
(defgoal)	vect	A vector to goal I am defending.
(goal)	vect	A vector to the goal I am attacking.
(closest-mate)	vect	A vector to my closest teammate.
(not *bool1*)	bool	Logical not.
(and *bool1 bool2*)	bool	Logical and.
(->+25 *vect1*)	vect	Rotates *vect1* 25 degrees in the direction I turned last.
(/2 *vect1*)	vect	Divides the magnitude of *vect1* by 2.
(dribble *vect1*)	vect	Sets the magnitude of *vect1* to some constant *c*.
(avg *vect1 vect2*)	vect	Returns the average of *vect1* and *vect2*.
(if *bool1 vect1 vect2*)	vect	Evaluates *bool1*. If it is 1, evaluates and returns *vect1*, else evaluates and returns *vect2*.
(change-state *vect1*)	vect	Toggles my internal state flag, and returns *vect1*.

Table 1: A small sample of some of the functions in the "primordial soup" GP soccer function set.

3 Using Genetic Programming to Evolve Coordinated Soccer Behaviors

The basic function set with which our soccer softbots are built consists of *terminal functions* of arity 0 which return sensor information, and *nonterminal functions* which operate on this data, provide flow-control, or modify internal state variables. We use Strongly-Typed GP [Montana, 1995] to provide for a variety of different types of data (booleans, vectors, etc.) accepted and returned by GP functions. Table 1 gives a sampling of the basic functions we provide our GP system with which to build individuals.

The first step in evolving a team is to form a set of low-level "basic" behaviors to be used by its players. Many basic behaviors are so simple that there is little reason to "evolve" them, because the soccer server effectively provides the data for these behaviors. This includes behaviors like "kick the ball into the goal", or "go home", which can be represented as simple vectors to the appropriate places.

However, some low-level behaviors are more interesting to evolve: "go to the ball" is more than a simple vector towards the ball because the player must *intercept* a moving ball. Another interesting (and difficult) behavior is determining which of several teammates is the best to kick to, or if a kick to the goal is a better choice than a pass. We have used Genetic Programming to search for good solutions to these behaviors, using the resultant solutions are part of the function set available to our teams.

Once a suitable collection of basic functions has been developed, there are a variety of ways to use Genetic Programming to "evolve" a soccer team. An obvious approach is to form teams from populations of individual players. The difficulty with this approach is that it introduces the *credit assignment problem:* when a team wins (or loses), how should the blame or credit be spread among the various teammates? We took a different approach: the Genetic Programming "individual" is the entire team itself; all the players in a team stay together through evaluations, breeding, and death.

Given that the GP individual is the team itself, this raises the question of a *homogenous* or *heterogeneous* team approach. With a homogenous team approach, each soccer player would follow effectively the same algorithm. With a heterogeneous approach, each soccer player would develop and follow its own unique algorithm. In a domain where heterogeneity is useful, the heterogeneous approach provides considerably more flexibility and the promise of more finely optimized behaviors and coordination. However, homogenous approaches take far less time to develop, since they require evolving only a single algorithm rather than (in the case of the Soccer domain) eleven separate algorithms.

We have opted for a hybrid of the two: our teams are divided into *squads*. Each squad develops a separate algorithm used by all the players within the squad. It is still possible for each player to develop its own unique behavior: the primordial soup of functions includes functions allowing each player to distinguish itself algorithmically from its squadmates.

The algorithm for a squad consists of two separate functional Lisp-like programs, one executed whenever the player is able to kick the ball, and the other executed when he can see the ball but cannot kick it (whenever a player cannot see the ball, the player simply searches for the ball). Both programs take as input various information about the state of the world, and output a <*Distance, Direction*> vector indicating an action (turning or kicking when in possession of the ball, turning and dashing when not).

All told, a full team consists of between three and six squads (depending on squad size), with two trees each, for a total of between six and twelve trees. This is a large number of trees to evolve within a single GP individual, requiring a large number of generations to produce adequate teams. To address this problem, we have experimented with using various *stepped*

evolution strategies to first develop good individuals, then good squads from those individuals, then good teams from those squads.

Our team-evaluation function uses *co-evolution* to determine team quality. To evaluate the fitness of all the teams in the population, it first pairs off teams in the population, then plays matches for each pair using the evaluation algorithm shown in Figure 4. Fitnesses are based on a variety of factors including (but not limited to) the number of goals, time in possession of the ball, average position of the ball during the fitness evaluation, etc. The resultant fitness assessments are then used by the Genetic Programming system to determine selection and breeding to form the next generation of soccer teams.

We perform our GP runs using a custom strongly-typed multithreaded version of *lil-gp 1.1* [Zongker and Punch, 1995]. To compensate for the very long evaluation time necessary in this domain (several seconds to several minutes), we perform twenty to eighty evaluations in parallel on a DEC Alpha workstation cluster. During GP evaluation, players in a team are run in sync in a single thread of execution. However, in the final RoboCup-97 competition, each player's algorithm-trees will actually run in a separate process independent of other players.

```
Pair off all teams in the population
For each pair,
   Prepare competition in Soccer Server.
   Loop until evaluation is finished,
      For each player on both teams,
         Update player with any new sensor data.
         If the player can kick the ball,
            Call the player's KICK program.
            Turn in the direction of the resultant vector.
            Kick the ball as directed by the vector.
            Yell out the name of the teammate closest to
                  where the ball will go.
         Else if the player can see the ball,
            Call the player's MOVE program.
            Turn and dash as directed by the resultant vector.
            Turn to face the ball again.
         Else
            Turn to look for the ball.
         Update the state estimator.
         Gather per-move information to evaluate fitness.
   Compute and return each team's fitness.
Based on fitness assessments in the population, perform GP
         selection, mutation, crossover, and reproduction
         to produce a new population.
Repeat as necessary.
```

Figure 4: Co-evolution evaluation algorithm for competitions

4 Acknowledgements

This research is supported in part by grants to Dr. James Hendler from ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), ARL (DAAH049610297), and ARPA contract DAST-95-C0037.

Our thanks to Lee Spector, Kilian Stoffel, Bob Kohout, Daniel Wigglesworth, John Peterson, Shaun Gittens, Shu Chiun Cheah, and Tanveer Choudhury for their help in the development of this project.

References

[Holland, 1975] J.H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1996.

[Iba, 1996] H. Iba. Emergent Cooperation for Multiple Agents using Genetic Programming. In J.R. Koza, editor, *Late Breaking Papers of the Genetic Programming 1996 Conference*. Stanford University Bookstore, Stanford CA, pages 66–74, 1996.

[Itsuki, 1995] N. Itsuki. Soccer Server: a simulator for RoboCup. In *JSAI AI-Symposium 95: Special Session on RoboCup*. December, 1995.

[Kitano *et al*, 1995] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. RoboCup: The Robot World Cup Initiative. In *Proceedings of the IJCAI-95 Workshop on Entertainment and AI/ALife*, 1995.

[Koza, 1992] J.R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge MA, 1992.

[Luke and Spector, 1996] S. Luke and L. Spector. Evolving Teamwork and Coordination with Genetic Programming. In J.R. Koza et al., editors, *Proceedings of the First Annual Conference on Genetic Programming (GP-96)*. The MIT Press, Cambridge MA, pages 150–156, 1996.

[Montana, 1995] D.J. Montana. Strongly Typed Genetic Programming. In *Evolutionary Computation*. The MIT Press, Cambridge MA, 3(2):199–230, 1995.

[Raik and Durnota, 1994] S. Raik and B. Durnota. The Evolution of Sporting Strategies. In R.J. Stonier and X.H. Yu, editors, *Complex Systems: Mechanisms of Adaption.* IOS Press, Amsterdam, pages 85–92, 1994.

[Reynolds, 1993] C.W. Reynolds. An Evolved, Vision–Based Behavioral Model of Coordinated Group Motion. In Jean-Arcady Meyer *et al.*, editors, *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. The MIT Press, Cambridge MA, 384–392, 1993.

[Zongker and Punch, 1995] D. Zongker and B. Punch. *lil-gp 1.0 User's Manual.* Available through the World-Wide Web at http://isl.cps.msu.edu/GA/software/lil-gp, or via anonymous FTP at isl.cps.msu.edu in the /pub/GA/lilgp directory. 1995.