Co-Evolving Soccer Softbot Team Coordination with Genetic Programming

Sean Luke
seanl@cs.umd.edu

Charles Hohn
hohn@mailb.harris.com

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 USA

Abstract. In this paper we explain how we applied genetic programming to behavior-based team coordination in the RoboCup Soccer Server domain. 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. The RoboCup Soccer Server was a very challenging domain for genetic programming, but we were pleased with the results. At the end, genetic programming had produced teams of soccer softbots which had learned to cooperate to play a good game of simulator soccer.

1 Introduction

The RoboCup competition pits robots 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.

The RoboCup competition has two leagues, a "real" robot league and a "virtual" simulation league. In 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]. The Soccer Server provides a simulator environment with complex dynamics, noisy and limited sensor information, noisy control, and real-time play. To win a soccer match in the Soccer Server, players must overcome these issues and cooperate as a team in the face of limited communication ability and an incomplete world-view.

[0] In *Proceedings of The First International Workshop on RoboCup*, IJCAI-97, Nagoya, Japan. Springer-Verlag. 1997.

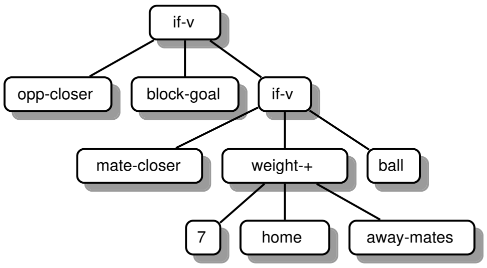

Fig. 1. A typical (small) GP algorithm tree

Given the RoboCup Soccer Server's model of loosely-distributed agent coordination and its dynamic environment, a reactive behavior-based approach is an appealing way to coordinate a softbot 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 players can be quite high. Instead of hand-coding these behaviors for each agent, we think it is attractive to have the agents *learn* good behaviors and coordination on their own. Beyond their interesting AI and Alife aspects, when agents learn on their own, they may find interesting solutions to the problem that hand-coding might overlook. The dynamics of the RoboCup soccer simulator are complex and difficult to optimize for. With enough time, a learned strategy can evaluate a broad range of different behaviors and hone in on those most successful.

Most 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. These strategies are effective for learning low-level soccer-player functions such as ball interception, etc., but may not be as well suited for learning emergent, high-level player coordination. In contrast, genetic programming (GP) uses evolutionary techniques to learn symbolic functions and algorithms which operate in some domain environment [Koza 1992]. This makes GP natural for learning programmatic behaviors in a domain like the Soccer Server. This paper describes GP and how we used it to evolve coordinated team behaviors and actions for our soccer softbots in RoboCup-97.

Genetic programming has been successfully applied many times in the field of multiagent coordination. [Reynolds, 1993] used GP to evolve "boids" in his later work on flocking and herd coordination. [Raik and Durnota, 1994] used GP to evolve cooperative sporting strategies, and [Luke and Spector, 1996], [Haynes *et al*, 1995], and [Iba, 1996] used GP to develop cooperation in predator-prey environments and other domains. Even so, evolutionary computation is rarely applied to a problem domain of this difficulty. As such, our goal was not so much to use evolutionary computation to develop finely-tuned soccer players as it was to see if evolving a fair team was even possible. As it turned out, we were pleasantly surprised with the results. Our evolved teams learned to disperse throughout the field, pass and dribble, defend the goal, and coordinate with and defer to other teammates.

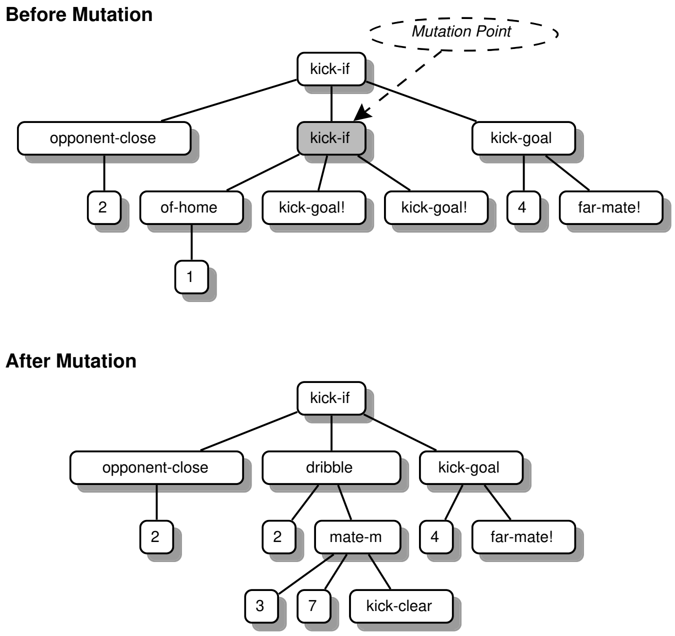

Fig. 2. The point mutation operator in action. This operator replaces one subtree in a genome with a randomly-generated subtree.

2 Genetic Programming

Genetic programming is a variant of the Genetic Algorithm [Holland, 1975] which uses evolution to optimize actual computer programs or algorithms to solve some task. In GP parlance, these algorithms are known as "individuals" or "genomes". 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 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 genome 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 viewed as the LISP code

```
(if-v opp-closer block-goal
    (if-v mate-closer
        (weight 7 home away-mates) ball))
```

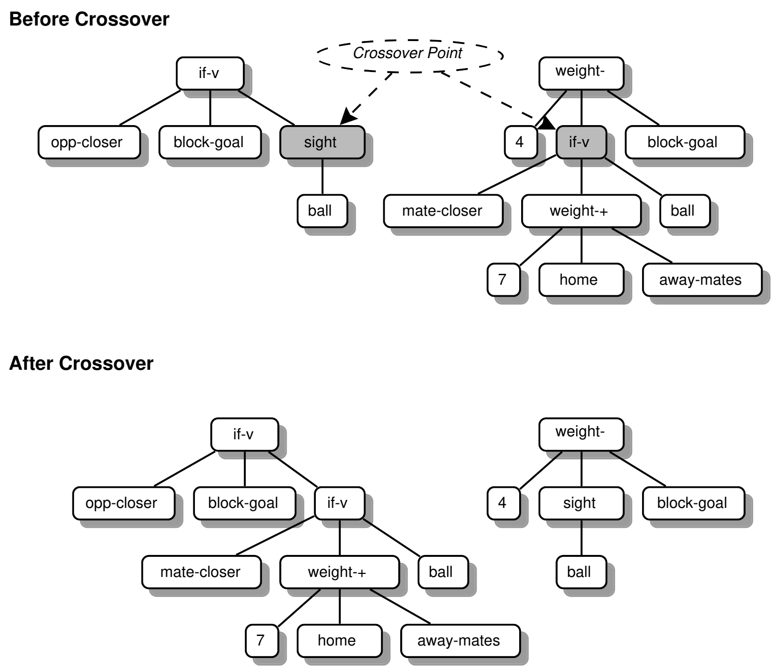

Fig. 3. The subtree crossover operator in action. This operator swaps subtrees among two genomes.

Genetic programming optimizes its genome trees with a process similar to the Genetic Algorithm. The user supplies the GP system with a set of atomic functions with which GP may build trees. Additionally, the user provides an *evaluation function*, a procedure which accepts an arbitrary GP genome tree, and returns an assessed fitness for this genome. The evaluation function assesses the fitness of a tree by executing its code in a problem domain.

The GP system begins by creating a large population of random trees for its first generation. It then uses the evaluation function to determine the fitness of the population, selects the more fit genome trees, and applies various breeding operators to them to produce a new generation of trees. It repeats this process for successive generations until either an optimally fit genome tree is found or the user stops the GP run.

GP's breeding operators are customized to deal with GP's tree-structured genomes. The three most common operators are *subtree crossover, point mutation,* and *reproduction*. GP's mutation operator (shown in Figure 2) takes a single fit genome, replaces an arbitrary subtree in this tree with a new, randomly-generated subtree, and adds the resultant tree to the next generation. GP's crossover operator swaps random subtrees in two fit trees to produce two new trees for the next generation, as shown in Figure 3. GP's reproduction operator simply takes a fit tree and adds it to the next generation. Our implementation used crossover and mutation, but not reproduction.

3 The Challenge of Evolutionary Computation

As if the soccer server's complex dynamics didn't make evolving a robot team hard enough, the server also adds one enormously problematic issue: time. As provided, the soccer server runs in real-time, and all twenty-two players connect to it via separate UDP sockets. The server dispenses sensor information and receives control commands every ten milliseconds; as a result, games can last from many seconds to several minutes. Game play can be sped up by hacking the server and players into a unified program (removing UDP) and eliminating the ten-millisecond delay. Unfortunately, for a variety of reasons this does not increase speed as dramatically as might be imagined, and if not carefully done, runs the risk of changing the game dynamics (and hence "changing" the game the players would optimize over).

The reason all this is such a problem is that evolutionary computation, and genetic programming in specific, typically requires a huge number of evaluations, and each new evaluation is another soccer simulator trial. In previous experiments with considerably simpler cooperation domains [Luke and Spector, 1996], we have found that genetic programming may require at least 100,000 evaluations to find a reasonable solution. We suspected the soccer domain could be much worse. Consider that just 100,000 5-minute-long evaluations in serial in the soccer server could require up to a full year of evolution time.

Our challenge was to cut this down from years to a few weeks or months, but still produce a relatively good-playing soccer team from only a few evolutionary runs. We accomplished this with a variety of techniques:

- We tried various evolutionary computation techniques (for example, co-evolution) to cut down on the number of evaluations.
- We customized GP in a variety of ways to be able to use smaller population sizes and numbers of generations.
- After initial experimentation, we designed the function set and evaluation criteria to promote better evolution in the domain.
- We performed simultaneous runs with different genome styles to promote diversity.
- We sped up play by performing up to 200 single-player evaluations and up to 32 full-team game evaluations in parallel. We also cut down game time from 10 minutes to between 20 seconds and one minute.

4 Using Genetic Programming to Evolve Soccer Behaviors

The basic function set with which our soccer softbots were built consisted of *terminal functions* of arity 0 which returned sensor information, and *nonterminal functions* which operated on this data, provided flow-control, or modified internal state variables. We used Strongly-Typed GP [Montana, 1995] to provide for a variety of different types of data (booleans, vectors, etc.) accepted and returned by GP functions. Strongly-Typed GP allows the user to assign a *type* to the arguments and the return value of each function. It then forms GP trees with the constraint that types of child and parent nodes must match appropriately. This allowed us to include a rich set of GP operators, but still constrain the possible permutations of function combinations.

Function Syntax	Returns	Description
(home)	v	A vector to my home.
(ball)	v	A vector to the ball.
(findball)	v	A zero-length vector to the ball.
(block-goal)	v	A vector to the closest point on the line segment between the ball and the goal I defend.
(away-mates)	v	A vector away from known teammates, computed as the inverse of $\sum_{m \in \{vectorstoteammates\}} \frac{max-\|\|m\|\|}{\|\|m\|\|} m$
(away-opps)	v	A vector away from known opponents, computed as the inverse of $\sum_{o \in \{vectorstoopponents\}} \frac{max-\|\|o\|\|}{\|\|o\|\|} o$
(squad1)	b	t if I am first in my squad, else nil.
(opp-closer)	b	t if an opponent is closer to the ball than I am, else nil.
(mate-closer)	b	t if a teammate is closer to the ball than I am, else nil.
(home-of *i*)	v	A vector to the home of teammate *i*.
(block-near-opp *v*)	v	A vector to the closest point on the line segment between the ball and the nearest known opponent to me. If there is no known opponent, return *v*.
(mate *i v*)	v	A vector to teammate *i*. If I can't see him, return *v*.
(if-v *b v1 v2*)	v	If *b* is t, return *v1*, else return *v2*.
(sight *v*)	v	Rotate *v* just enough to keep the ball in sight.
(ofme *i*)	b	Return t if the ball is within $\frac{i}{max}$ units of me, else nil.
(ofhome *i*)	b	Return t if the ball is within $\frac{i}{max}$ units of my home, else nil.
(ofgoal *i*)	b	Return t if the ball is within $\frac{i}{max}$ units of the goal, else nil.
(weight-+ *i v1 v2*)	v	Return $\frac{v1(i)+v2(9-i)}{9}$.
(far-mate *i k*)	k	A vector to the most offensive-positioned teammate who can receive the ball with at least $\frac{i+1}{10}$ probability. If none, return *k*.
(mate-m *i1 i2 k*)	k	A vector to teammate *i1* if his position is known and he can receive the ball with at least $\frac{i2+1}{10}$ probability. If not, return *k*.
(kick-goal *i k*)	k	A vector to the goal if the kick will be successful with at least $\frac{i+1}{10}$ probability. If not, return *k*.
(dribble *i k*)	k	A "dribble" kick of size $\frac{i}{20}(max)$ in the direction of *k*.
(kick-goal!)	k	Kick to the goal.
(far-mate!)	k	Kick to the most offensive-positioned teammate. If there is none, kick to the goal.
(kick-clear)	k	Kick out of the goal area. Namely, kick away from opponents as computed with (away-opps), but adjust the direction so that it is at least 135 degrees from the goal I defend.
(kick-if *b k1 k2*)	k	If *b* is t, return *k1*, else return *k2*.
(opponent-close *i*)	b	Return t if an opponent is within $\frac{max}{(1.5)i}$ of me.
0,1,2,3,4,5,6,7,8,9	i	Constant integer values.

Table 1. Some GP functions used in the soccer evaluation runs. Other functions included internal state, magnitude and cross-product comparison, angle rotation, boolean operators, move history, etc. *max* is the approximate maximum distance of kicking, set to 35. *k* is a kick-vector, *v* is a move-vector, *i* is an integer, and *b* is a boolean.

Before evolving a team, we had to create the set of low-level "basic" behavior functions to be used by its players. Table 1 gives a sampling of the basic functions we provided our GP system with which to build GP trees. We decided early on to enrich a basic function set of vector operators and if-then control constructs with some relatively domain-specific behavior functions. Some of these behaviors could be derived directly from the Soccer Server's sensor information. This included vector functions like (kick-goal!), or (home). Others behaviors were important to include but were hand-coded because we found evolving them unsuccessful, at least within our limited time constraints. These included good ball interception (a surprisingly complex task), which was formed into (ball), or moving optimally to a point between two objects (forming (block-near-opp), for example).

We used genetic programming to evolve other low-level behaviors. Most notably, we used the GP technique of *symbolic regression* to evolve symbolic functions determining the probability of a successful goal-kick or pass to a teammate, given opponents in various positions [Hohn, 1997]. Symbolic regression evolves a symbolic mathematical expression which best fits a set of data points, using only basic mathematical operators such as (+ ...) or (sin ...). Our symbolic regression data points were generated by playing trials in the soccer server. The evolved results formed the mechanism behind the decision-making functions (kick-goal ...), (mate-m ...), and (far-mate ...).

To meet the needs of the soccer softbot domain, we made some significant changes to the traditional GP genome. Instead of a player algorithm consisting of a single tree, our players consisted of two algorithm trees. One tree was responsible for making kicks, and when evaluated would output a vector which gave the direction and power with which to kick the ball. The other tree was responsible for moving the player, and when evaluated would output a vector which gave the direction and speed with which to turn and dash. At evaluation-time, the program executing the player's moves would follow the instructions of one or the other tree depending on the following simplifying state-rules:

– If the player can see the ball and is close enough to kick the ball, call the kick tree. Kick the ball as told, moving the player slightly out-of-the-way if necessary. Turn in the direction the ball was kicked.
– If the player can see the ball but isn't close enough to kick it, call the move tree. Turn and dash as told; if the player can continue to watch the ball by doing so, dash instead by moving in reverse.
– If the player cannot see the ball, turn in the direction last turned until the player can see it.

Given a basic function set, 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 genome is an entire team; all the players in a team stay together through evaluations, breeding, and death.

Given that the GP genome is the team itself, this raises the question of a *homogenous* or *heterogeneous* team approach. With a homogenous team approach, each soccer player

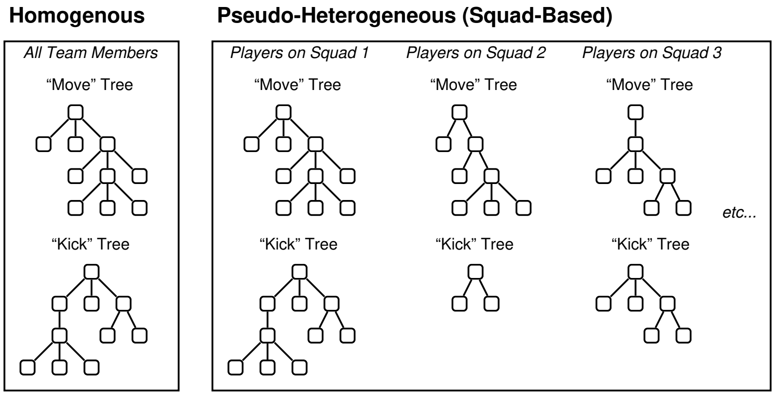

Fig. 4. Homogeneous and Pseudo-Heterogeneous (Squad-Based) genome encodings

would follow effectively the same algorithm, and so a GP genome would be one kick-move tree pair. With a heterogeneous approach, each soccer player would develop and follow its own unique algorithm, so a GP genome would be not just a kick-move tree pair, but a forest of such pairs, one pair per player. In a domain where heterogeneity is useful, the heterogeneous approach provides considerably more flexibility and the promise of specialized 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.

To implement a fully heterogeneous approach in the soccer domain would necessitate evolving a genome consisting of twenty-two separate GP trees, far more than could reasonably evolve in the time available. Instead, we ran separate runs for homogeneous teams and for hybrid *pseudo-heterogeneous* teams (see Figure 4). The hybrid teams were divided into six squads of one or two players each; each squad evolved a separate algorithm used by all players in the squad. This way, pseudo-heterogeneous teams had genomes of twelve trees. Each player could still develop his own unique behavior, because the primordial soup included functions which let each player distinguish himself from his squadmates.

Because our genomes consisted of forests of trees, we adapted the GP crossover and mutation operators to accommodate this. In our runs, crossover and mutation would apply only to a single tree in the genome. For both homogeneous and pseudo-heterogeneous approaches, we disallowed crossover between a kick tree and a move tree. For pseudo-heterogeneous approaches, we allowed trees to cross over only if they were from the same squad: this "restricted breeding" has in previous experience proven useful in promoting specialization [Luke and Spector, 1996]. We also introduced a special crossover operator, *root crossover*, which swapped whole trees instead of subtrees. This allowed teams to effectively "trade players", which we hoped would spread strategies through the population more rapidly.

Another issue was the evaluation function needed to assess a genome's fitness. One way to assess a team would be to play the team against one or more hand-created opponent teams of known difficulty. There are two problems with this approach. First, evolutionary computation strategies typically work more efficiently when the difficulty of the problem ramps up as evolution progresses, that is, as the population gets better, the problem gets harder. A good ramping with a suite of pre-created opponents is very difficult to gauge. Second, unless there are several opponents at any particular difficulty level, one runs the very common risk of evolving a team optimized to beat that *particular set* of hand-made opponents, instead of generalizing to playing "good" soccer.

We opted instead for evolving our teams with *co-evolution*: teams' fitnesses were assessed based on competition with peers. In our implementation, to evaluate the fitness of all the teams in the population, the GP system first paired off teams in the population, then played matches with each pair. Team fitness was based on the game score and other game criteria.

Co-evolution is chaotic and so can occasionally have undesirable effects on the population, but more often than not we have found it natural for many "competitive" domains (such as robot soccer). Co-evolution naturally ramps problem difficulty because teams in the population play against peers of approximate ability. Co-evolution can also promote generalization, because the number of "opponents" a population faces is the size of the population itself.

We initially based fitness on a variety of game factors including the number of goals, time in possession of the ball, average position of the ball, number of successful passes, etc. However, we found that in our early runs, the entire population would converge to very poor solutions. Ultimately, we found that by simply basing fitness on the number of goals alone, the population avoided such convergence. At first glance, such a simplistic fitness assessment would seem an overly crude a measure, as many early games might end with 0–0 scores. Luckily, this turned out to be untrue. We discovered that initial game scores were in fact very high: vectors to the ball and to the goal were fundamental parts of the function set, so teams did simple offense well, but defense poorly. Only later, as teams evolved better defensive strategies, would scores come down to more reasonable levels.

We performed our GP runs in parallel using a custom strongly-typed, multithreaded version of *lil-gp 1.1* [Zongker and Punch, 1995], running on a 40-node DEC Alpha cluster. To speed up evolution run time, we used population sizes between 200 and 400 (small for GP), which required a large amount of mutation (up to 70%) to prevent premature convergence.

We ran the final runs for forty generations, at which time we re-introduced into the population high-fitness individuals from past generations. We then continued runs up to the time of the RoboCup-97 competition (for twelve generations). Just prior to the competition, we held a "tournament of champions" among the twenty highest-performing teams at that point, and submitted the winner. While we feel that, given enough evolution time, the learned strategies of the pseudo-heterogeneous teams might ultimately outperform the homogeneous teams, the best teams at competition time (including the one we submitted) were homogeneous.

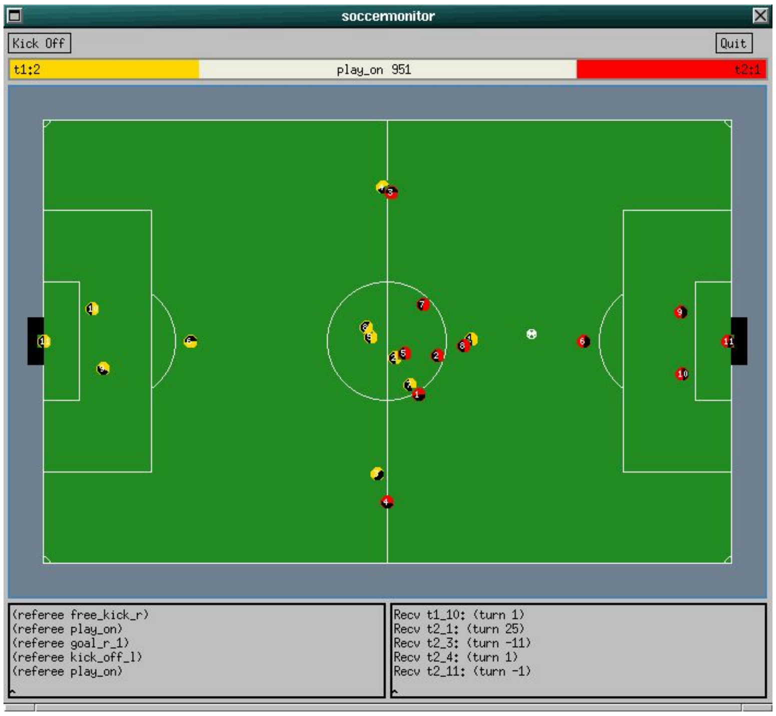

Fig. 5. A competition between two initial random teams.

5 A History of Evolution

One of the benefits of working with evolutionary computation is being able to watch the population learn. In a typical evolutionary computation experiment one would conduct a large number of runs, which provides a statistically meaningful analysis of population growth and change. For obvious reasons, this was not possible for us to do. Given the one-shot nature of our RoboCup runs (the final run took several months' time), our observations of population development are therefore admittedly anecdotal. Still, we observed some very interesting trends.

Our initial random teams consisted primarily of players which wandered aimlessly, spun in place, stared at the ball, or chased after teammates. Because (ball) and (kick-goal!) were basic functions, there were occasional players which would go to the ball and kick it to the goal. These players helped their teams rack up stratospheric scores against completely helpless opponents. Figure 5 shows two random teams in competition.

Early populations produced all sorts of bizarre strategies. One particular favorite was a (homogeneous) competition of one team programmed to move away from the ball, against another team programmed to move away from the first team. Thankfully, such strategies didn't last for many generations.

Fig. 6. "Kiddie-Soccer", a problematic early suboptimal strategy, where everyone on the team would go after the ball and try to kick it into the goal. Without careful tuning, many populations would not escape this suboptima.

One suboptimal strategy, however, was particularly troublesome: "everyone chase after the ball and kick it into the goal", otherwise known as "kiddie-soccer", shown in Figure 6. This strategy gained dominance because early teams had effectively no defensive ability. Kiddie-soccer proved to be a major obstacle to evolving better strategies. The overwhelming tendency to converge to kiddie-soccer and similar strategies was the chief reason behind our simplification of the evaluation function (to be based only on goals scored). After we simplified the evaluation function, the population eventually found its way out of the kiddie-soccer suboptima and on to better strategies.

After a number of generations, the population as a whole began to develop rudimentary defensive ability. One approach we noted was to have a few players hang back near the goal when not close to the ball (Figure 7). Most teams still had many players which clumped around the ball, kiddie-soccer-style, but such simple defensive moves effectively eliminated the long-distance goal shots which had created such high scores in the past.

Fig. 7. Some players begin to hang back and protect the goal, while others chase after the ball.

Eventually teams began to disperse players throughout the field and to pass to teammates when appropriate instead of kicking straight to the goal, as shown in Figure 8. Homogeneous teams did this usually by using players' home positions and information about nearby teammates and ball position. But some pseudo-heterogeneous teams appeared to be forming separate offensive and defensive squad algorithms. It was unfortunate that the pseudo-heterogeneous teams were not sufficiently fit by the time RoboCup arrived; we suspect that given more time, this approach could have ultimately yielded some very good strategies.

6 Conclusions and Future Work

This project was begun to see if it was even possible to successfully evolve a team for such a challenging domain as the RoboCup Soccer Server. Given the richness of the Soccer Server environment and the very large amount of evolution time required to get reasonable results, we are very pleased with the outcome. Our evolved softbots learned to play rather well given the constraints we had to place on their evolution. As such, we think the experiment was a success.

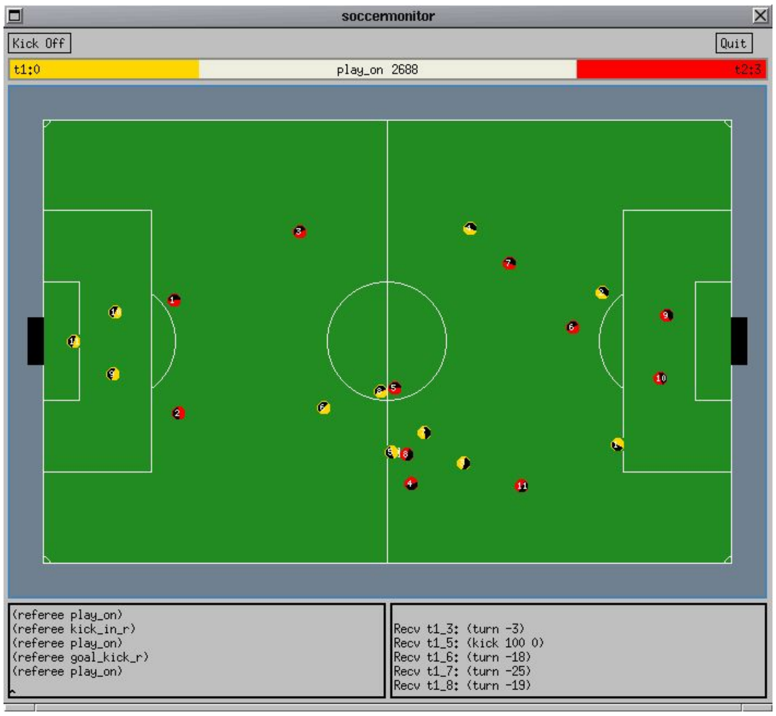

Fig. 8. Teams eventually learn to disperse themselves throughout the field.

Still, we had to compromise in order to make the project a reality. The function set we provided was heavy on flow-control and functional operation, with little internal state. In the future we hope to try more computationally sophisticated algorithms. We also hope to perform longer runs with significantly larger population sizes. Finally, we were disappointed that the pseudo-heterogeneous teams could not outperform our homogeneous teams by RoboCup competition-time. In the future we hope to try again with heterogeneous teams, which we suspect might yield better results.

While genetic programming has been very successful in some domains, it is surprisingly difficult to adapt it to others, especially domains where evaluations are expensive. Still, we think our work shows that considerably more can be done with this technique than is usually thought. As the price of computer brawn continues to drop, we feel evolutionary computation may become attractive in a variety of areas which are currently the purview of human programmers only.

7 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, and to the UMIACS system staff for turning their heads while we played soccer games on their supercomputers.

References

[Haynes *et al*, 1995]. Haynes, T., S. Sen, D. Schoenefeld and R. Wainwright. 1995. Evolving a Team. In *Working Notes of the AAAI-95 Fall Symposium on Genetic Programming*. E. V. Siegel and J. R. Koza, editors. 23–30. AAAI Press.

[Hohn, 1997] C. Hohn. Evolving Predictive Functions from Observed Data for Simulated Robots. Senior Honor's Thesis. Department of Computer Science, University of Maryland at College Park. 1997.

[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 J.-A. 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 at http://isl.cps.msu.edu/GA/software/lil-gp 1995.