Genetic Programming Produced Competitive Soccer Softbot Teams for RoboCup97

Sean Luke
seanl@cs.umd.edu
http://www.cs.umd.edu/~seanl/

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

ABSTRACT

At RoboCup, teams of autonomous robots or software softbots compete in simulated soccer matches to demonstrate cooperative robotics techniques in a very difficult, real-time, noisy environment. At the IJCAI/RoboCup97 softbot competition, all entries but ours used human-crafted cooperative decision-making behaviors. We instead entered a softbot team whose high-level decision making behaviors had been entirely evolved using genetic programming. Our team won its first two games against human-crafted opponent teams, and received the RoboCup Scientific Challenge Award. This report discusses the issues we faced and the approach we took to use GP to evolve our robot soccer team for this difficult environment.

1 Introduction

RoboCup is a competition which pits teams of robots against each other in a robotic soccer tournament [Kitano *et al* 1995]. To be successful at RoboCup, a team of robotic soccer players must be able to cooperate in real time in a noisy, highly-dynamic environment against an opposing team. In addition to the two "real-robot" leagues at RoboCup, there is a softbot league which competes inside a provided soccer simulator, the RoboCup Soccer Server [Itsuki 1995]. The simulator enforces extremely limited and noisy sensor information, complex physics, real-time dynamics, and limited intercommunication among softbots. The result is a rather challenging real-time domain.

Practically all entrants in the RoboCup simulator league used hand-coded team strategy algorithms; though some fine-tuned a few low-level functions (like ball interception) with backpropagation or decision trees. In contrast, at the University of Maryland a group of undergraduates and I entered a softbot team whose high-level strategies were entirely learned through genetic programming [Luke *et al* 1997].

Unlike other teams, who had refined well-understood robotics techniques in order to win the competition, we saw the RoboCup simulator as a very difficult environment to push the bounds of what was possible to do with existing evolutionary computation techniques. For a variety of reasons detailed later, the soccer simulator is *very* difficult to evolve for. Hence, our goal was relatively modest: to produce a team which played at all. As it turned out, we were pleasantly surprised with the results. Our evolved teams learned to disperse throughout the field, pass, kick to the goal, defend the goal, and coordinate with and defer to other teammates. At the IJCAI/RoboCup97 competition our team managed to win its first two matches against human-coded opponents, and took home the RoboCup97 Scientific Challenge award.

2 The RoboCup Soccer Server Domain

Genetic programming has been successfully applied to multiagent coordination before. [Andre 1995] evolved communication between agents with different skills. [Qureshi 1996] evolved agent-based communication in a cooperative avoidance domain. [Raik and Durnota 1994] used GP to evolve cooperative sporting strategies, and [Luke and Spector 1996], [Haynes *et al* 1995] used GP to develop cooperation in predator-prey environments. [Iba 1996] applied a similar approach to cooperative behavior in the TileWorld domain. Even so, evolutionary computation is rarely applied to a problem domain of this difficulty. The RoboCup Soccer Server (http://ci.etl.go.jp/~ noda/soccer/server.html) was not designed with GP in mind. To realize why the Soccer Server presents such a challenge for evolutionary computation (and GP in specific), it's important to understand how the domain works.

In a full match, the Soccer Server admits eleven separate player programs per team, each controlling a different virtual soccer player in its simulation model. By regulation rules, these player programs must be separate processes which are not permitted to communicate with each other except through the limited facilities provided by the Soccer Server. Each player on the team makes a separate socket connection to the simulator. Once connected, a player program receives UDP datagrams once every 300 milliseconds, providing it with sensor information and messages "yelled" by other players on the field. The player issues commands to the server by sending it UDP datagram messages no faster than once every 100 milliseconds. Commands are not queued: if the player issues commands faster than this, they are simply ignored by the server. The server updates its internal world model every ten milliseconds; this places a real-time restriction on the speed of play.

The simulator maintains a virtual soccer field 105 units long by 68 units wide, with goals 14 units wide. Sensor information relays game status and the relative positions of viewable objects on the field. The only useful sensor option (and the one used by all players in the competition) gives both the direction and distance of objects the player can "see". Players may choose narrow (45 degrees), medium (90 degrees — we and most competitors chose this), or wide (180 degrees) fields of view, but the wider the range, the more slowly sensor updates are received. Sensor information includes only:

- The ball position and relative movement.
- Positions and relative movement of other players. If players are far enough away (over 20 units), their jersey numbers cannot be ascertained. If players are very far away (over 40 units), the *team* they're on cannot be determined.
- Goal positions.
- The position of flags placed at corners of the field, and on the edges of the field at its midpoint.
- The distance to and perceived angle of the soccer field boundary line crossing the player's field of vision.
- The state of the ball in play, including free, goal, side, and corner kicks, out-of-bounds, pre-game and mid-game setup, kickoffs, etc.

Sensing is further complicated in three ways. First, the coordinate positions of objects are given *relative to the player's position and the direction he is facing*, but the server does not tell the player any information about his own whereabouts. Second, the server gives information only about object closer than 3 units, or within the player's field of view (the widest is only 180 degrees). Third, the simulator adds to the sensor positional information a heavy dose of gaussian noise, and noise increases as the distance to an object increases.

Player movement is nonholonomic (players can either turn or dash but not both at the same time), which makes things messy. And like sensor information, movement is also subject to a great deal of noise. Each movement cycle, a player may issue one of several commands:

- Rotate n degrees from his current facing direction.
- Dash in the direction he is facing with n power. A player must repeatedly dash to keep up forward movement. Dashing also decreases stamina; players that dash with high power will soon start running much slower than they realize. A player is not told his current stamina, nor how fast he is currently running. Players may also dash backwards, but at most a third of maximum power.
- Kick the ball in a certain direction (relative to the direction the player is facing) and with a certain power. Players can only kick a ball when it is under 1.8 units away.
- Yell a message up to 512 bytes long. There are strong limitations on this. Messages may be yelled only very infrequently, and fellow players can hear only so many messages each sensor cycle. Further, players aren't told where a yell came from; hence yells can be (and often are) faked by opponents.
- Move the player to a specific (X,Y) coordinate position and facing a specific angle. This is only permitted while the ball is out of play.

Play happens in real time. If a player cannot process sensor information or make moves fast enough, he will fall behind. The Soccer Server also maintains complex dynamics among moving objects. Balls and players have acceleration and momentum, and cannot immediately stop, change direction, or move at a certain velocity on command. Players and the ball have different mass, hence can move at different rates (the ball can move much faster). Players take up space and collide with the ball and other players inelastically. When a team is given possession of the ball (for a goal kick, perhaps), the server "bumps" opposing players from the general ball area. Though not used in the competition, the simulator can also provide wind and other complicating conditions.

The simulator enforces standard soccer rules with one very large exception: as the robot players have no hands, there is no goalie, and no goalie area. When a ball is kicked out-of-bounds, ball control is transferred to opponents and the ball is moved to the appropriate kick-in position per regulation rules. Goals are scored when the ball passes through the goal line. Finally, the server allows a human referee to make foul calls for ungentlemanly play (for example, the entire team lining up to block the goal).

3 The Challenge for Evolutionary Computation

As should be obvious from the above description, the Soccer Server domain is very complex, with a large number of options and controls, and a correspondingly large number of boundary conditions and special cases that must be accounted for. This alone makes it a tough problem to tackle with GP.

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. Because of the enforced ten-millisecond delay between world model updates, a full game takes ten minutes to play. Game play can be sped up by hacking the server and players into a unified program (removing UDP) and eliminating the ten-millisecond delay. However, we found that for many 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 to optimize over).

The reason all this is such a problem is that evolving a computer program to work successfully in this domain would likely require a very large 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 could require on order of 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 in several ways:

- Brute force. We sped up play by performing up to 32 full-team game evaluations in parallel on a supercomputer cluster. We also cut down game time from 10 minutes to between 20 seconds and one minute.
- We attempted to cut down the population size and number of generations necessary to produce a reasonable team.
- We developed an additional layer of software which simplified and orthogonalized the domain, eliminating many of the boundary conditions the GP programs would have to account for. We also spent much time designing a function set and evaluation criteria to promote better evolution in the domain.
- We performed parallel runs with different genome structures to give us more options as competition time neared.

4 Using Genetic Programming to Evolve Soccer Behaviors

As the Soccer Simulator dynamics were quite complex, we began by hand-coding a multithreaded socket library which abstracted away some of the oddities of the domain. The library received all incoming sensor information and boiled it down into the *absolute* position of all visible teammates and opponents (and the player himself), the ball, and the goals. We included a simple state-estimation mechanism that interpolated teammate and opponent positions in-between sensor cycles and maintained estimates of the player's stamina. The boiled-down domain provided information about whose ball it was during free-kicks, goal-kicks, etc., but this information was largely unused as the simulator would keep players out of the kick area anyway. As we decided to ignore the complexities of intercommunication, our domain also eliminated the ability to yell or listen.

We also 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. The first 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. The second 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. At evaluation-time, the program executing the player's moves would follow the instructions of one or the other tree based 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.

This state mechanism eliminated a great many troublesome boundary conditions. First, by combining "movement" into turn-dash pairs, we allowed the GP tree function set to assume its player had holonomic movement, that is, the ability to move immediately in any direction. Second, by doing everything reasonable to keep the ball in view, we were able to eliminate many of the boundary conditions which occur when ball suddenly disappears due to arbitrary player movement (a big problem in our early tests).

Before evolving a team, we had to create the set of low-level "basic" behavior functions to be used by its players. This required some compromise. Ideally, we would have liked to produce soccer players out of a variety of very low-level, generic vector functions. This would have allowed us to say that in no way did we bias the function set to produce certain kinds of strategies. But our early tests suggested that the domain was so complex that there was little hope of evolving a team with this kind of function set. Instead, we designed a large set of functions (some generic, some specialized) we thought would have particular utility in the soccer domain. In doing so, we tried to stay as general as possible but still come up with a function set that we thought stood a chance of evolving successfully.

Function Syntax	Returns	Description
(home)	v	A vector to my home (my starting position).
(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 $\sum_{m \in \{\textit{vectors to teammates}\}} \frac{max - \lvert\lvert m \rvert\rvert}{\lvert\lvert m \rvert\rvert} m$
(away-opps)	v	A vector away from known opponents, computed as the inverse of $\sum_{o \in \{\textit{vectors to opponents}\}} \frac{max - \lvert\lvert o \rvert\rvert}{\lvert\lvert o \rvert\rvert} o$
(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*.
(inv *v*)	v	*v* rotated 180 degrees.
(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: GP functions used in the soccer evaluation runs. Other functions tried (but not used in the final runs) 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. Vectors (for either kicking or moving) are a pair of floating-point values.

To achieve this, 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, restricting tree formation to conform to these type rules. This allowed us to include a large, rich set of GP functions (allowing for many more player options), but still constrain the possible permutations of function combinations.

Table 1 gives 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, Charles Hohn used *symbolic regression* to evolve functions determining the probability of a successful goal-kick or pass to a teammate, given opponents in various positions [Hohn 1997]. Our symbolic regression data points were generated by playing actual trials in the Soccer Server (with a kicker, receiver, and opponent for teammate-pass trials, or just a kicker and opponent for goal-kick trials). We used these evolved algorithms as the basic probabilistic mechanism behind the decision-making functions (kick-goal ...), (mate-m ...), and (far-mate ...).

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, widely used in GP, which we had tried before in [Luke and Spector 1996]: the genetic programming genome is an entire team; all the players in a team stay together through evaluations and breeding.

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, and so a GP genome would be a single kick-move tree pair used by all teammates during play. 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 can take far less time to evolve, since they require evolving only a single algorithm rather than (in this case) eleven algorithms.

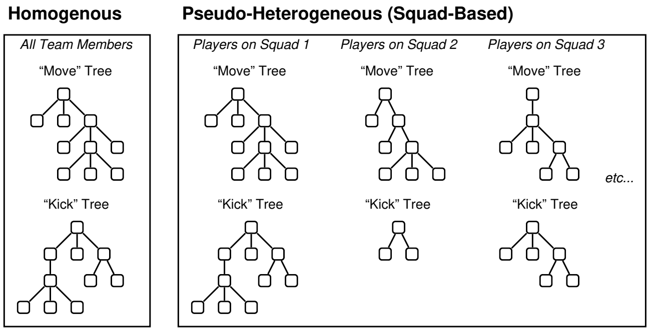

Figure 1: Homogeneous and Pseudo-Heterogeneous (Squad-Based) genome encodings.

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 we felt could reasonably evolve in the time available. Instead, we ran separate runs for homogeneous teams and for hybrid *pseudo-heterogeneous* teams (see Figure 1). 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 function set included functions which let each player distinguish himself from his squadmates. We ran separate runs with these two different approaches to increase our chance of having something to show at RoboCup (as discussed later, our concern was justified).

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 at the root instead of swapping subtrees. This let teams effectively "trade players", which we hoped would spread good strategies through the population more rapidly.

To reduce run time, we used population sizes between 100 and 400 (in the final run, 128). We felt these small populations (given the problem complexity) necessitated a somewhat unusual mix of breeding operators. As a consequence of findings in [Luke and Spector 1997], we decided to use a large dose of mutation (30% in the final run) to produce higher overall fitness with the small population, and to stave off premature convergence. The rest consisted of 70% subtree crossover (choosing internal-nodes 30% of the time,

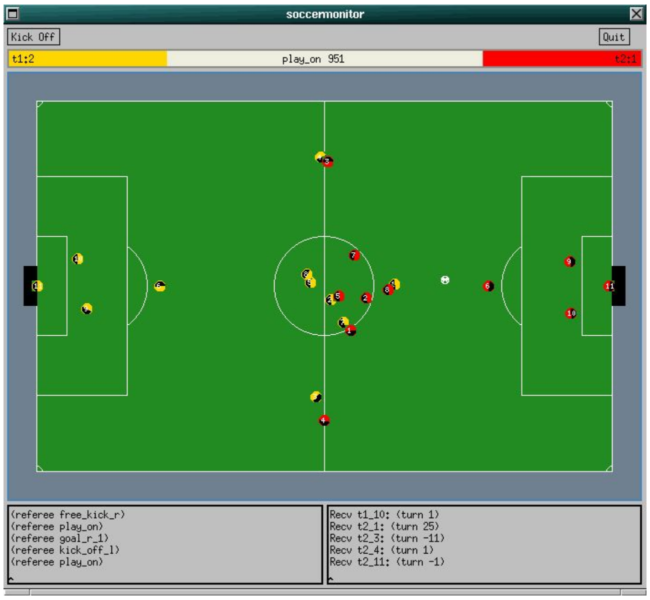

Figure 2: A competition between two initial random (and randomly-moving) teams.

Figure 3: "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.

leaf-nodes 10% of the time, and performing root crossover 60% of the time). Finally, we used tournament selection with a tournament size of 7.

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, from our experience, evolutionary computation strategies often 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 difficult to gauge. Second, unless there are several opponents at any particular difficulty level, one runs the common risk of evolving a team optimized to beat that *particular set* of hand-made opponents, instead of generalizing to play "good" soccer.

We opted instead for evolving our teams in a *competitive fitness* environmnent [1]: teams' fitnesses were assessed based on competition with peers in the population (for a survey of such environments, see [Angeline and Pollack 1996]. There are a variety of approaches to creating such competitions. One approach is to create a "round-robin" tournament where every team in the population squares off at least once against every other team. This approach is very expensive: $(n^2 - n)/2$ evaluations for a population of size n. Another approach is to use a traditional single- or double-elimination tournament ($n - 1$ evaluations at best). Because of the extreme cost on evaluations, we opted instead to randomly pair up teams, basing team fitness on the single game each pair played ($n/2$ evaluations).

Competitive fitness functions are chaotic and so can occasionally have undesirable effects on the population, but we found them a useful fit for a naturally competitive domain such as robotic soccer. Competitive fitness functions also naturally ramp problem difficulty because teams in the population play against peers of approximately similar ability. Such functions can also promote generalization, because the set of possible "opponents" an individual might face is 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 goal difference alone, the population avoided such convergence. At first glance, such a simplistic fitness assessment would seem an overly crude 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 and quite variable: 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 defen-

[1] This might all fit under the "co-evolution" umbrella. In theoretical biology, the definition "co-evolution" (along with the term "species") has become rather fuzzy of late. But while I originally used "co-evolution" to describe this environment, I think the term carries just too much inter-species (or multi-population) baggage. "Competitive fitness" is more precise.

Figure 4: Some players begin to hang back and protect the goal, while others chase after the ball.

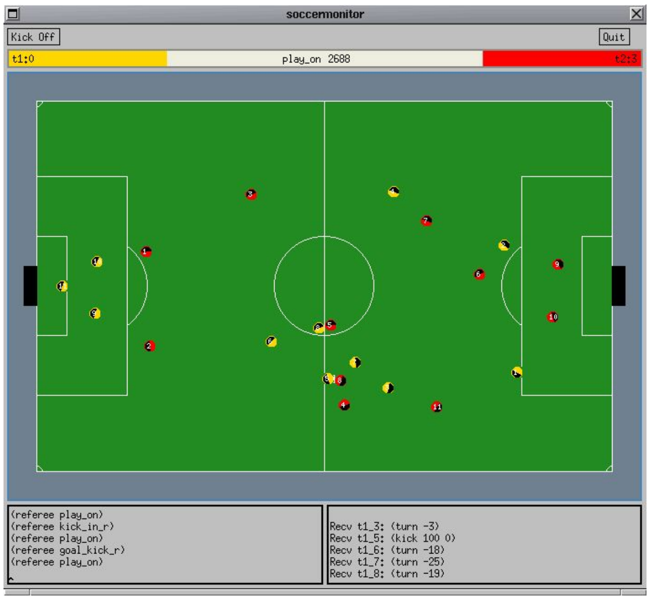

Figure 5: Teams eventually learn to disperse themselves throughout the field.

sive 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 supercomputer. At evaluation time, the system paired off competitors, formed the pairs into groups, and assigned each group to a separate evaluation thread. In parallel, these evaluation threads would work with the socket communication library to pair off teams and play competitions in separate Soccer Server processes.

At some point I felt we would need the population to stop global searches and start narrowly tweaking its best strategies to date in preparation for the competition. As such, we ran the final runs for forty generations, at which time we reintroduced into the population high-fitness individuals from past generations, and added 10% reproduction (30% mutation, 60% crossover). The intent of this unusual step was to force the population to rapidly converge to a narrow set of suboptima. 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 I 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.

5 A History of Evolution

One of the fun parts of working with this domain is watching the population learn. In a typical GP 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 helpless opponents. Figure 2 shows two random teams playing.

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.

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 3. 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 common approach we noted was to have a few players hang back near the goal when not close to the ball (Figure 4). 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.

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 5. 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. Although 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 better 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 long run 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, and they beat teams crafted by hand by real human experts (who weren't us). As such, I think the experiment was a success.

Still, we had to compromise in order to make the project a reality. The function set we provided was heavy on if-then statements and functional operation, with little internal state. In the future I hope to try more computationally sophisticated algorithms. The competition mechanism (only one game per individual) was also very restrictive: I think a single-elimination tournament might yield better results. Finally, the population sizes were very small; enlarging these could make a significant impact on evolutionary progress.

It is unfortunate that the our pseudo-heterogeneous teams could not outperform our homogeneous teams by RoboCup competition-time. Much of this is due to the excessive size of the pseudo-heterogeneous genomes (which were *still* much smaller than full-heterogeneous genomes). Hindsight is 20-20. In the future I would definitely pick larger squad sizes, perhaps three squads of three or four each, or two squads of five or six each. This would bring the total number of trees in the genome down to six or four, which might evolve more rapidly.

In the past, genetic programming has been surprisingly successful in a variety of areas, but real-time robotics is not one of them. The complex dynamics of the field, plus the long time necessary to perform evaluations, makes robotics (and realistic simulator robotics) a difficult problem for evolutionary computation to crack. But not an impossible problem. I hope this paper puts to rest the myth that genetic programming can compare favorably to human-crafted solutions only in toy domains, or for problems tailor-made for the constraints of evolution.

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.

My thanks to the Maryland RoboCup team for their help in the development of this project: Charles Hohn, Jonathan Farris, Gary Jackson, Daniel Wigglesworth, John Peterson, Shaun Gittens, Shu Chiun Cheah, and Tanveer Choudhury. Thanks also to Jim Hendler, Lee Spector, Kilian Stoffel, Bob Kohout, Hiroaki Kitano, Minoru Asada, and to the UMIACS system staff for turning their heads while we played soccer games on their supercomputers.

References

Andre, D. 1995. The Automatic Programming of Agents that Learn Mental Models and Create Simple Plans of Action. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, C. S. Mellish, ed. 741–747. Morgan Kaufmann, San Mateo CA.

Angeline, P. and J. Pollack. Competitive Environments Evolve Better Solutionf for Complex Tasks. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, S. Forrest, ed. 264–270. Morgan Kaufmann, San Mateo CA.

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

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

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

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

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

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

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

Luke, S. *et al*. 1997. Co-evolving Soccer Softbot Team Coordination with Genetic Programming. In *Proceedings of the RoboCup-97 Workshop at the 15th International Joint Conference on Artificial Intelligence (IJCAI97).* H. Kitano, ed. 115–118. IJCAI.

Luke, S. and L. Spector. 1997. A Comparison of Crossover and Mutation in Genetic Programming. *In Genetic Programming 1997: Proceedings of the Second Annual Conference (GP97).* J. Koza et al, eds. 240–248. San Fransisco: Morgan Kaufmann.

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

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

Qureshi, A. 1996. Evolving Agents. In *Proceedings of the First Annual Conference on Genetic Programming (GP-96)*, J. R. Koza *et al*, eds. 369–374. The MIT Press, Cambridge MA.

Zongker, D. and B. Punch. 1995. *lil-gp 1.0 User's Manual.* Available at http://isl.cps.msu.edu/GA/software/lil-gp