Genetic Programming for Robot Soccer

Vic Ciesielski, Dylan Mawhinney, and Peter Wilson

RMIT Department of Computer Science
GPO Box 2476V, Melbourne 3001
Tel: (03) 9925-2926, Fax: (03) 9662-1617
vc@cs.rmit.edu.au

Abstract. RoboCup is a complex simulated environment in which a team of players must cooperate to overcome their opposition in a game of soccer. This paper describes three experiments in the use of genetic programming to develop teams for RoboCup. The experiments used different combinations of low level and high level functions. The teams generated in experiment 2 were clearly better than the teams in experiment 1, and reached the level of 'school boy soccer' where the players follow the ball and try to kick it. The teams generated in experiment 3 were quite good, however they were not as good as the teams evolved in experiment 2. The results suggest that genetic programming could be used to develop viable teams for the competition, however, much more work is needed on the higher level functions, fitness measures and fitness evaluation.

1 Introduction

Genetic programming is a process of generating programs by simulated evolution[3, 4]. This paper considers the question of whether genetic programming would be useful in a complex, changing, uncertain environment such as simulated robot soccer.

Genetic programming has previously been used to develop RoboCup teams [1, 5]. The team evolved in [5] won two games in the 1997 tournament. Both these attempts have been somewhat competitive. Our focus is on finding a good set of high level functions and associated fitness measures prior to working on a team for the competition. Other machine learning techniques have been used previously for RoboCup including agent oriented programming [6], neural networks [7], and decision trees [8].

Strongly typed genetic programming (STGP) is an extension of standard genetic programming. All functions and terminals in a STGP system must return a particular data type, all function arguments must also be restricted to a specific type. The main benefit of STGP is the reduced search space it offers. We have developed a strongly typed genetic programming system which can evolve players for the RoboCup competition, full details can be found in [9].

[1] Mpeg files of some games can usually be found at http://www.cs.rmit.edu.au/∼vc/robocup

A. Birk, S. Coradeschi, and S. Tadokoro (Eds.): RoboCup 2001, LNAI 2377, pp. 319–324, 2002.

The overall goal of this work is to determine whether a competitive RoboCup team can be developed by genetic programming. In this investigation we are interested in determining if a team can be generated using just the basic soccer simulator primitives (kick, turn etc), and what fitness measures are useful.

2 Experiment 1 – Basic Robocup Functions

Experiment 1 was the basic starting point for evolving genetic programs to play soccer. Each chromosome represents a control program for a soccer player. A team consists of 11 copies of this player. The only action functions available to programs were those already provided by the soccer server (kick, dash, turn). It was hoped that programs would use these very basic functions and terminals to evolve some useful soccer-playing behaviours.

The program elements (terminals and functions) available to programs in experiment 1 can be found in table 1. Terminals are inputs to a conventional program[4] and will always be leaves of a program tree. Functions form the interior nodes of a program tree.

Table 1. Functions and terminals used in experiment 1. In the type columns, `f` denotes a floating point value and `v` a vector.

Terminal Name	Type	Description
Random Number	f	A random number n where $0 <= n < 360$.
NearTeam	v	Returns the location of the player's nearest teammate.
2ndNearTeam	v	Returns the player's 2nd nearest teammate.
3rdNearTeam	v	The player's 3rd nearest teammate.
4thNearTeam	v	The player's 4th nearest teammate.
NearEnemy	v	The player's nearest opponent.
2ndNearEnemy	v	The player's 2nd nearest opponent.
3rdNearEnemy	v	The player's 3rd nearest opponent.
4thNearEnemy	v	The player's 4th nearest opponent.
BallPos	v	The position of the ball.
MyGoalPos	v	The position of a player's goal (the goal a player must defend).
TheirGoalPos	v	The position of a player's opponent's goal (the goal a player needs to kick the ball through to score).
Function Name	**Returns**	**Description**
$iflte f(a, b, c, d)$	f	If a is less than or equal to b, return c, else return d.
$kick(pow, dir)$	a	Send a kick message to the soccer server with the parameters pow power and dir direction.
$turn(moment)$	a	Turn the player within a range of -180 to 180 degrees.
$dash(pow)$	a	Cause the player to dash in the direction they are facing.
$min(a, b)$ $max(a, b)$	f	Return the minimum or maximum of a and b.
$+, -$	f	Standard arithmetic operators.
$getDist(v)$	f	Return the distance element of a vector.
$getDir(v)$	f	Return the direction element of a vector.
$getCDist(v)$	f	Return the change in distance element of a vector.
$getCDir(v)$	f	Return the change in direction element of a vector.
$execTwo(a, b)$	a	Unconditionally execute both actions and return an action.

2.1 Genetic Operators

The basic genetic operators are crossover and mutation. These operators work on the tree representations of the evolving programs. The crossover operator

selects two parents and swaps randomly selected subtrees to get the children, as shown in figure 1. The mutation operator selects a subtree at random and replaces it with a new, randomly generated tree.

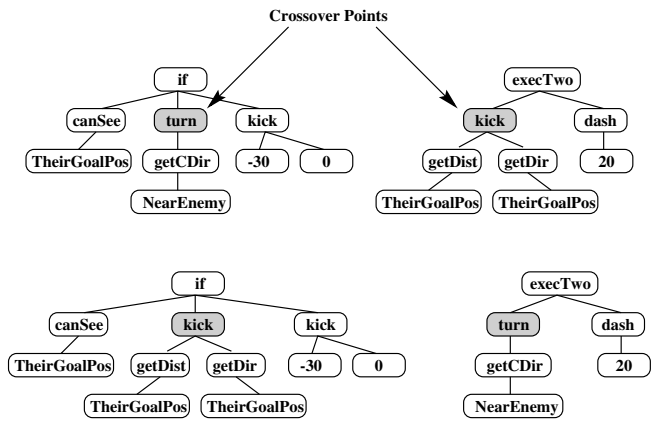

Fig. 1. The genetic crossover operation. The top two trees are the parents, selected at random from the population. The bottom two trees are the children. The chosen subtrees have been swapped.

2.2 Fitness

For this experiment, each generation proceeded in an elimination-round style tournament [2]. Programs were randomly paired up and played off. The winner of the meeting was then advanced to the next round. The fitness of each program was determined by the round in which it was eliminated. The best individuals of each generation were then played off in a best-of tournament.

2.3 Results

The results for this experiment were unexpectedly disappointing. Not only did every program fail to score a proper goal, most failed to even kick the ball. Even less encouraging was the fact that some teams even managed to score goals against themselves. Since very few teams scored goals or executed successful kicks or passes, the winner of a tournament was often determined at random.

One winner was a program with 77 nodes. This program provides its player with one action — kick, it contains no `dash` or `turn` commands and as such cannot cause the player to move at all. The quality of this program is typical of the winners from experiment 1.

Using just low level functions (as was the case in this experiment) did not allow the genetic programming process to evolve players which could display any basic soccer playing behaviours.

3 Experiment 2 - Higher Level Functions

The terminals, genetic operators and fitness evaluation were the same as in experiment 1. The non terminal functions used in experiment 1 (table 1) were replaced by those shown in table 2. These functions correspond to more complex soccer actions than just turning, dashing and kicking. A number of additional, lower level, support functions were also needed. Full details are in [9].

Table 2. Experiment 2, main non terminal functions

Function Name	Returns	Args	Description
$if(a, b, c)$	a	b, a, a	If a is true, execute b, else execute c.
$canSee(v)$	b	v	Return true if the player can see its argument, otherwise return false.
$turnTo(v)$	a	v	If the percept vector is visible, turns the player to face it and runs towards it. If it isn't visible, the player scans for it by turning 90° and waiting for a percept update.
$moveTo(v)$	a	v	This function causes the player to turn towards its argument in the same fashion as $turnTo$ (including the scanning if it isn't visible). If the percept vector is visible, the player then runs towards it.
$kickTo(v)$	a	v	If the player can see the ball and is within kicking distance, it will kick it in the direction and distance of its argument, else this function acts like $moveTo$ with the ball's position as its argument.
$shorten(v)$	v	v	Shorten the distance v is away from the player by 20%.

3.1 Results

The higher level representations used in this experiment allowed the generated programs to exhibit basic soccer-playing behaviour. Like early programs described in Luke *et al.* [5], the programs evolved exhibited behaviour which might be expected at a children's soccer game (ie. all players believe that they alone can win the game). All programs which made it into the final best-of-the-best elimination-round tournament flocked to the ball, irrespective of the location of teammates and opponents on the field. Most programs could kick the ball, some could not. Those that could not kick the ball surrounded it and typically blocked their opponents' efforts to get near the ball. Few goals were scored in this best-of tournament, mainly due to the ball-smothering behaviours of the players.

While many of the programs evolved in experiment 2 were able to play basic games of soccer, the code which had been evolved was usually not very sophisticated. The majority of the programs had simply made use of the `kickTo` function to run to the ball and kick it in some direction (usually towards the goal). In fact 27 of the 32 best programs played off in the tournament had the `kickTo` function as their root node.

4 Experiment 3 - Augmented Basic Robocup Functions

In experiment 3 the terminals and low level functions in experiment 1 were used along with some additional terminals and functions. The new terminals

and functions added for this experiment can be found in table 3. Experiment 3 used the roulette wheel selection method [4], each team played one game against another team in the population. Fitness was calculated as a weighted sum of goals scored and kicks made.

Table 3. Experiment 3, added terminals and functions

Terminal Name	Returns	Description
$GameState$	f	Returns the current state of the game (play_on, kick_off etc) as a number.
$PlayNum$	f	Returns the shirt number of the current player.
$IsLeft$	b	Returns true if the player is on the left hand side team, false otherwise.
$IsRight$	b	Returns true if the player is on the right hand side team, false otherwise.
$ClosestToBall$	b	Returns true if the player is the closest to the ball out of the players it can see.
Function Name	**Returns**	**Description**
$equal(f, f)$	b	Returns true if the two numbers passed to it are equal.
$or(b, b)$	b	Returns true if either of it's arguments are true.
$and(b, b)$	b	Returns true if both it's arguments are true.
$*, \%(f, f)$	f	Multiplication and protected division.
$if(a, b, c)$	a	If a is true return b, else return c.

4.1 Results

The programs generated during this experiment were able to get to the ball, and then kick it, however the ball did not always go towards the goal. Most of the teams which were evolved did not run directly towards the ball, they instead developed a "swirling" behaviour where the team would run around in small circles as a group, the group of swirling players would move towards the ball eventually kicking in what seemed to be a random direction. Strangely this behaviour was common, it was observed in many independent evolutionary runs which had each started from random populations. Teams which used this strategy were able to kick goals some of the time, however this was only the result of kicking the ball hard in random directions, most teams scored just a many own goals as they did goals. We are currently looking at modifying the fitness function to encourage more goals. Another interesting behaviour which emerged from some of the evolutionary runs, was that of goalie behaviour. Some of the teams had a player which would stand in front of the goals and not move around or follow the ball.

The teams evolved in experiment 3 were not as good as those evolved in experiment 2. However the actual "learning" that took place in experiment 3, was much more advanced than in experiment 2. In experiment 2 the majority of the programs which were evolved simply made use of the `kickTo` function which was available. In experiment 3 however the programs evolved were more complex and contained many different functions and terminals.

5 Conclusions

Experiment 1 showed that very low-level behaviours by themselves are not enough to evolve a set of high-level and complex soccer-playing strategies. Players all but lost the ability to dash by generation 7. No goals were scored, some own-goals were scored, but most players failed to even kick the ball.

Experiment 2 utilised higher-level functions and resulted in players which exhibited 'school boy soccer' abilities, that is, each player followed the ball around as it was kicked back and forth.

Experiment 3 utilised low-level functions, however it had quite a few different functions than in experiment 1. The players exhibited reasonable abilities, however they were not up to the level of the players from experiment 2.

The form of the fitness function and the method fitness evaluation remain unresolved issues. It seems clear that different fitness functions are needed early in the evolutionary process when the teams are not very good and later, when the teams have improved. Further work is under way in this area.

References

[1] David Andre and Astro Teller. Evolving team Darwin United. In Minoru Asada, editor, *Robocup-98: Robot Soccer World Cup II. Lecture Notes in Com puter Science*. Springer-Verlag, 1999.

[2] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[3] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic programming: An introduction on the automatic evolution of computer programs and its applications*. San Francisco, Calif. : Morgan Kaufmann Publishers, 1998. Subject: Genetic programming (Computer science); ISBN: 1-55860-510-X.

[4] John Koza. *Genetic Programming: on the Programming of Computers by means of Natural Selection*. The MIT Press, 1992.

[5] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[6] Itsuki Noda. Team gamma: Agent programming on gaea. In H. Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, 1998.

[7] Itsuki Noda, Hitoshi Matsubara, Kazuo Hiraki, and Ian Frank. Soccer server: a tool for research on multi-agent systems. *Applied Artificial Intelligence*, 12(2-3), 1998.

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

[9] Peter Wilson. *Development of a Team of Soccer Playing Robots by Genetic Programming*. Honours Thesis, RMIT, Department of Computer Science, 1998 http://www.cs.rmit.edu.au/~vc/papers/wilson-hons.ps.Z.