Byoung-Tak Zhang and Dong-Yeon Cho

Genetic programming provides a useful paradigm for developing multiagent systems in the domains where human programming alone is not sufficient to take into account all the details of possible situations. However, existing GP methods attempt to evolve collective behavior immediately from primitive actions. More realistic tasks require several emergent behaviors and a proper coordination of these is essential for success. We have recently proposed a framework, called *fitness switching*, to facilitate learning to coordinate composite emergent behaviors using genetic programming. *Co-evolutionary fitness switching* described in this chapter extends our previous work by introducing the concept of coevolution for more effective implementation of fitness switching. Performance of the presented method is evaluated on the table transport problem and a simple version of simulated robot soccer problem. Simulation results show that coevolutionary fitness switching provides an effective mechanism for learning complex collective behaviors which may not be evolved by simple genetic programming.

18.1 Introduction

Evolving complex collective behaviors is an interesting problem for distributed intelligence and artifi cial life. Some tasks can be done faster or more easily by dividing them up among many agents. Other tasks may not only be solved better by using multiple agents, but can only be effectively solved, by using teams of agents working together [Kube and Zhang, 1993; Mataric, 1996].

Several attempts have been made to use genetic programming to evolve cooperative behavior of a group of simple robotic agents. [Koza, 1992] and [Bennett III, 1996] used genetic programming to evolve a common program that controls foraging for food by ants. [Haynes et al., 1995] showed that programs for solving a predator-prey problem can be generated by genetic programming without any deep domain knowledge. [Luke and Spector, 1996] explored various strategies for evolving teams of agents in the Serengeti world, a simple predator-prey environment. [Iba, 1997] studied three different breeding strategies (homogeneous, heterogeneous, and coevolutionary) for cooperative robot navigation. Genetic programming was also used in agent based computing. [Qureshi, 1996] demonstrated that it is possible to evolve agents that communicate and interact with each other to solve a global problem.

Most of these studies have attempted to evolve emergent collective behavior immediately from primitive actions. However, more realistic complex tasks require more than one emergent behavior and a proper coordination of these is essential for successful accomplishment of the task. In real applications, it is common to use externally imposed structures to reduce the complexity of learning [Digney, 1996]. For example, [Langdon, 1998] and [Bruce, 1995] show that evolving a list took much less effort if broken into 2 tasks, one following the other.

In previous work [Zhang and Cho, 1998] we have introduced a framework, called *fitness switching*, that facilitates evolution of composite emergent behaviors using genetic programming. In fitness switching, different parts of a genetic tree are responsible for different behaviors and for each of the subtrees a basis fitness function is defined. The complex behavior is produced by dynamically changing fitness types from a pool of fitness functions. *Coevolutionary fitness switching* described in this chapter is an extension of fitness switching in which multiple subtrees are coevolved in a single GP run.

Our approach is different from other heterogeneous breeding schemes [Luke and Spector, 1996; Iba, 1997] in which different subtrees represent different *agents*. In coevolutionary fitness switching, different subtrees represent different *behaviors* of a single agent which need to be coordinated. The basic idea behind this approach is that fitness functions are a fundamental mechanism that guides the evolutionary process. It is motivated by progressive learning, i.e. learning easier tasks first and then harder tasks, which is a well-proven method in pedagogy [Zhang and Hong, 1997]. As will be discussed later, coevolutionary fitness switching can be considered as a method that enhances scalability of genetic programming by enabling the designer to incorporate problem structure while preserving essential explorative and automatic programming capability of genetic programming.

The chapter is organized as follows. Section 18.2 describes the general framework for coevolutionary fitness switching. Section 18.3 evaluates the effectiveness of the framework on the table transport task, a multiagent cooperation task. Section 18.4 shows application results on simulated robotic soccer. Section 18.5 discusses our results and further work.

18.2 Genetic Programming with Coevolutionary Fitness Switching

Genetic programming is an automatic programming method that finds the most fit computer programs by means of natural selection and genetics [Koza, 1992; Langdon, 1998; Banzhaf et al., 1998]. A population of computer programs are generated at random. They are evolved to better programs using genetic operators. The ability of the program to solve the problem is measured as its fitness value.

In genetic programming, the computer programs are usually represented as *trees* or LISP S-expressions. The tree consists of elements from the function set and the terminal set. Typically, terminal symbols provide values to the GP program while function symbols perform operation on their input, which are either terminals or output from other functions. Therefore, terminals should be evaluated earlier than functions; that is, bottom-up evaluations are performed. For multiagent control, functions denote sensing of environments and terminals denotes actions to be taken. Thus functions first should determine the state of environment and then actions described by terminals are taken; that is, top-down evaluations are performed. An illustrative example of the genetic program for multiagent control is shown in Figure 18.1.

Fitness switching is a method designed for evolving complex group behaviors using genetic programming [Zhang and Cho, 1998]. The procedure for applying fitness switching to a specific problem can be summarized as follows:

1. Define the primitive actions for the problem domain. These are the terminal set, i.e. the actions executed by the agents to solve the problem.

2. Define a small number of micro-behaviors

$$\mathcal{B} = \{B_1, B_2, \cdots, B_n\},$$

that constitute the original problem-solving behavior.

3. Define a fitness function for each micro-behavior. Together they make the pool of fitness functions

$$\mathcal{F} = \{f_1, f_2, \cdots, f_n\}.$$

4. Design a sequence of micro-behaviors or their combinations to achieve the target behavior:

$$S_t = S_{t-1} \oplus B_t,$$

where $\oplus$ denotes the append operator and S_0 is the empty sequence, i.e. $S_0 = \langle\rangle$. The corresponding sequence of fitness functions are defined as

$$F_t = F_{t-1} \oplus f_t,$$

where $F_0 = \langle\rangle$.

5. Define the structure of a genetic program A as having n subtrees, A^i $(i = 1, \cdots, n)$, immediately under the root node.

$$A = (A^1, A^2, \cdots, A^n).$$

6. Apply genetic programming to evolve S_t, $t = 1, \cdots, n$ in sequence. The first subtree A^1 is executed on the given problem and the fitness of this subtree is evaluated using its fitness function f_1. Then the second subtree A^2 is executed and evaluated. Likewise, other subtrees are executed and evaluated sequentially. After that, the fitness of the program A, $F(A)$, is calculated (Figure 18.2).

The method is called fitness switching since evolution is guided by fitness functions switched from simpler ones to more complex ones. If multiple subtrees are coevolved, then the method is referred to as *coevolutionary fitness switching*.

In coevolutionary fitness switching, each subtree A^i is responsible for one micro-behavior B_i and fitness measures are switched within a single generation. Fitness of programs is measured at each generation as follows:

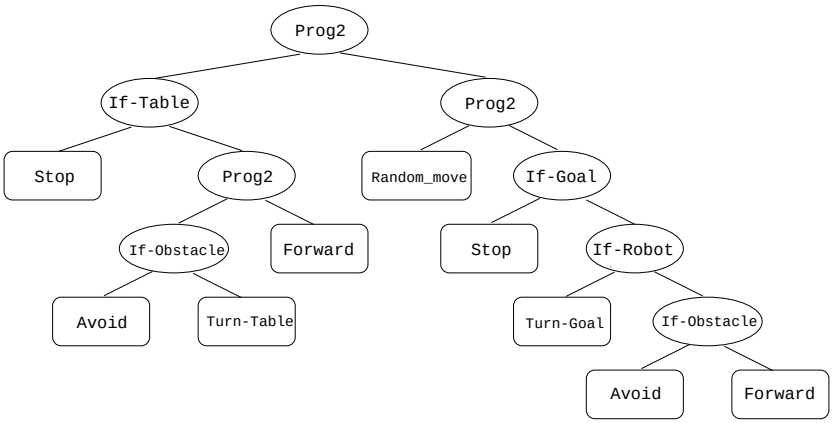

Figure 18.1
An example of genetic program for controlling the behavior of a robot. The left subtree means "`If` the `table` is nearby then `stop`, else do [`if` an `obstacle` is nearby then `avoid` it, otherwise `turn` to the `table`] and move `forward`." Likewise, the right subtree encodes a control program for the robot which is executed after the left subtree.

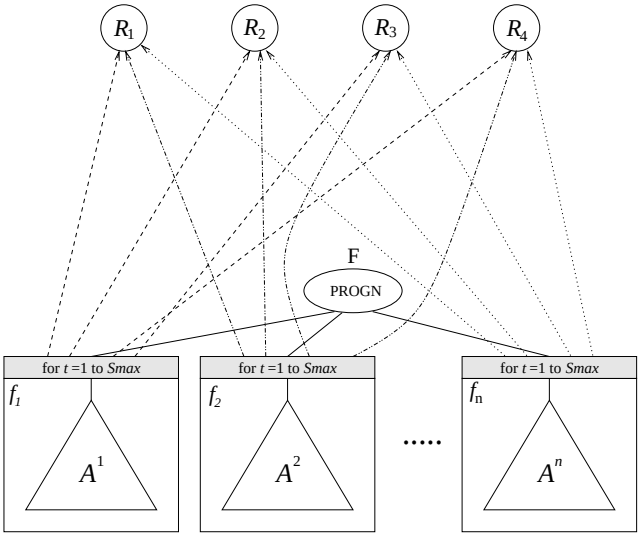

Figure 18.2
Schematic diagram for genetic programming with coevolutionary fitness switching. For multiagent control, each subtree A^i represents a control program for a micro-behavior of each robot R_j. The fitness of subtree A^i is assigned by executing it S_{max} times and measuring the goodness of its behavior by fitness function f_i.

```
Procedure CoevolutionaryFitnessSwitching()
  Pop = Initialize()
  For each generation g = 1 to G
      For each individual A ∈ Pop
          F(A) = 0
          For each subtree t = 1 to n
              For i = 1 to S_max
                  Execute(A^t)
              EndFor
              Calculate f_t
              F(A) = F(A) + f_t
          EndFor
      EndFor
      Pop = Select(Pop, |Pop|/2)
      NewPop = { }
      For each selection step s = 1 to |Pop|/2
          Select A1 and A2 from Pop based on F
          (C1, C2) = CrossoverMutation(A1, A2)
          NewPop = NewPop ∪ {C1, C2}
      EndFor
      Pop = NewPop
  EndFor
  Return(A_best = Best(Pop))
EndProcedure
```

Figure 18.3
Procedure for genentic programming with coevolutionary fitness switching. Each subtree A^i is responsible for a single micro-behavior and fitness measures are switched within a single generation to coordinate a sequence of micro-behaviors to evolve macro-behaviors. (See text for more details.)

1. Execute the first subtree A^1 for S_{max} times and measure its fitness by f_1.

2. Execute the second subtree A^2 for S_{max} times and measure its fitness by f_2.

3. Execute other subtrees and measure their fitness sequentially.

4. The fitness of the program $A = (A^1, A^2, \cdots, A^n)$ is defined as $F(A) = \sum_{i=1}^{n} f_i$.

The advantage of this method is the ability of concurrent evolution of primitive cooperative behaviors and their coordination.

A more formal description of this procedure is given in Figure 18.3. The initial population is created with random individuals. Then, the fitness values of individual A_i at generation t for the training set D, $F_t^{(i)} := E(D|A_i)$, are evaluated as described above. Based on the adaptive Occam method [Zhang et al., 1997] a complexity term was used in

all experiments to penalize large trees:

$$F(i) = E(D|A_i) + \beta C(A_i),$$

where $C(A_i)$ is the complexity measured in the number of nodes in tree A_i and β is a small constant. This measure is based on the minimum description length (MDL) principle, i.e. it encourages to induce models that have the minimal total code length:

$$A_{best} = \arg\min_{A_i}\{L(D|A_i) + L(A_i)\},$$

where $L(A_i)$ is the code length for model description and $L(D|A_i)$ is the code length for data description using the model.

We used (μ, λ) uniform ranking selection, i.e., the best μ individual assigned a selection probability of $\frac{1}{\mu}$, while the rest are discarded:

$$p_s(A_i) = \begin{cases} \frac{1}{\mu}, & 1 \leq i \leq \mu \\ 0, & \mu < i \leq \lambda. \end{cases}$$

In experiments $\frac{\mu}{\lambda} = 0.5$, i.e. the best 50% of the population are deterministically selected into the mating pool. Two parents in the mating pool are selected at random to generate two offspring by crossover and mutation.

The crossover operator selects a subtree from A_i^k of the first parent A_i and swap it with a subtree selected from the A_j^k in the second parent A_j. If the depth of the offspring tree exceeds the depth limit, crossover is performed again. Then, the mutation operator changes a node symbol according to the mutation rate. A terminal node is replaced by another terminal node and a nonterminal node by a different nonterminal.

The genetic operators are applied repeatedly until offspring of population size are produced. After generating all offspring, the best two of the parents population replace two individuals selected at random from the offspring population (2-elitism). This completes one generation and the process is repeated for G generations.

There are two simple implementational variants of fitness switching. One is *naive evolution* in which each micro-behavior is performed using the whole genetic program tree. Naive evolution is one extreme on which most existing GP studies are based. This method is very efficient in memory usage since the same tree is shared by multiple behaviors. A disadvantage is that this representation is difficult to coordinate multiple cooperative behaviors. An alternative method for implementing fitness switching is *sequential evolution*. Here each subtree A^i is responsible for one micro-behavior B_i. Subtree A^i is evolved by a GP run and then the best program for this run is used to evolve the next GP run for evolving the subtree A^{i+1}. This is another extreme in which the coordination is hard-coded both in representation and in evolutionary process. Sequential evolution approach seems the most practical in solving tasks which can be clearly decomposed into a sequence of independent subtasks. But most of the interesting problems that need emergent computations do not belong to this class of problems.

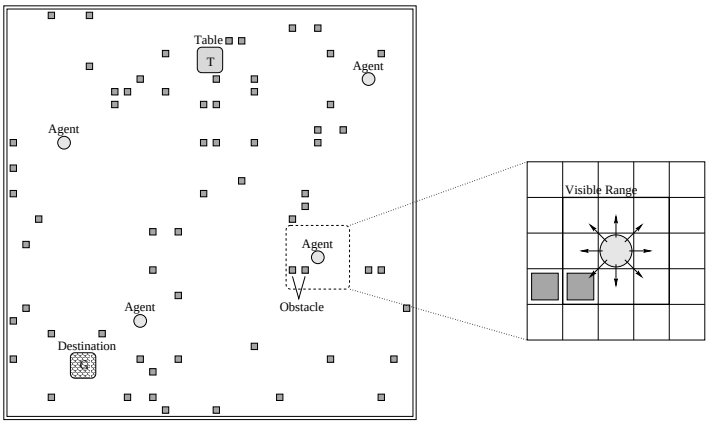

Figure 18.4
Table transport problem. There is one table, one destination, four robotic agents and a fixed number of obstacles. The robots can move in 8 directions and have a limited visual field of range 1 in all 8 directions. The table and destination are assumed to be observable by the agents from any position.

18.3 Results on Table Transport

The different GP approaches have been experimentally compared on table transport problem. This problem consists of an $n \times m$ grid world, a single table T and four robotic agents as shown in Figure 18.4. A specific location G is designated as the destination. All 6 initial locations are chosen randomly. The goal of the robots is to transport the table to the destination G. The robots need to move in a herd since the table is too heavy and large to be transported by single robots.

This problem requires at least two emergent behaviors, i.e. homing and herding, to be executed in sequence. The four robots need first to get together around the object to transport, i.e. homing, and then transport it in team to the destination, i.e. herding. Though in a different context, [Werner and Dyer, 1993] have studied the herding behavior. In this task, a group of robot agents must cooperate to achieve the goal.

The fitness switching method is applied to the table transport problem. This task can be considered as a composition of two following cooperative behaviors: homing and herding. Thus the sequence of micro-behaviors is $\mathcal{B} = \langle B_1, B_2 \rangle$, where $B_1 = homing$ and $B_2 = herding$. The sequence of fitness functions is $\mathcal{F} = \langle f_1, f_2 \rangle$, where f_1 measures the fitness of the homing behavior to the table and f_2 measures the fitness of the herding behavior for transporting the table to the goal. Better programs are defined to have lower

Table 18.1
Symbols used for fitness definition.

Symbol	Description
X_r	x-axis distance between target and robot r
Y_r	y–axis distance between target and robot r
S_r	number of steps moved by robot r
C_r	number of collisions made by robot r
M_r	distance between starting and final position of r
A_r	penalty for moving away from other robots
c_i	coefficient for factor i
K	positive constant ($K = 40$)

fitness values as follows:

$$f_1 = \sum_{r=1}^{4}\{c_1 \max(X_r, Y_r) + c_2 S_r + c_3 C_r - c_4 M_r + K\} \quad (18.1)$$

$$f_2 = \sum_{r=1}^{4}\{c_1 \max(X_r, Y_r) + c_2 S_r + c_3 C_r - c_4 M_r + c_5 A_r + K\} \quad (18.2)$$

where the subscript r denotes the index for robots. Typical values of c_i are $c_1 = 5.0$, $c_2 = 0.25$, $c_3 = 1.0$, $c_4 = 3.0$, and $c_5 = 0.25$. The definitions of the symbols used in these equations are summarized in Table 18.1. The constant *K* is used to normalize the fitness values to be positive (this does not change the selective chance of the individuals.) The target position for homing behavior is the initial position of the table T while the target position for herding behavior is the destination G of the table.

The objective of a GP run is to find a multi-robot algorithm that, when executed by each robot in a group of 4 robots, causes efficient table transport behavior in group. The terminals and functions used for GP are listed in Table 18.2. The function set consists of six primitives: `IF-OBSTACLE`, `IF-ROBOT`, `IF-TABLE`, `IF-GOAL`, `PROG2` and `PROG3`. The terminal set consists of six primitive actions: `FORWARD`, `AVOID`, `RANDOM-MOVE`, `TURN-TABLE`, `TURN-GOAL` and `STOP`. If `FORWARD` or `RANDOM-MOVE` cause a robot to run into obstacles, other robots or edges of the world, then the robot remains the current position. We assume that all primitive actions take the same time for execution and the robots have a mixture of local and global sensors. Local sensors (e.g., infra-red sensors) are used for `IF-OBSTACLE` and `IF-ROBOT`. Global sensors are used for `IF-TABLE` and `IF-GOAL`. An example of genetic program is shown in Figure 18.1. Table 18.3 summarizes the experimental setup for genetic programs. We enforce a maximum tree depth of 10 to avoid generating overly large structures.

Each fitness case represents a world of 32 by 32 grid on which there are four robots, 64 obstacles, and the table to be transported. A total of 20 different, randomly generated training cases were used. A total of 20 different worlds were also used for evaluating the

Table 18.2
Terminals and functions of GP-trees for the table transport problem.

	Symbol	Description
Terminals	FORWARD	Move one step forward in the current direction.
	AVOID	Check clockwise and make one step in the first direction that avoids collision.
	RANDOM-MOVE	Move one step in the random direction.
	TURN-TABLE	Make a clockwise turn to the nearest direction of the table.
	TURN-GOAL	Make a clockwise turn to the nearest direction of the goal.
	STOP	Stay at the same position.
Functions	IF-OBSTACLE	Check collision with obstacles.
	IF-ROBOT	Check collision with other robots.
	IF-TABLE	Check if the table is nearby (within the range of one cell.)
	IF-GOAL	Check if the destination is nearby (within the range of one cell.)
	PROG2, PROG3	Evaluate two (or three) subtrees in sequence.

Table 18.3
Parameters used in the experiments for the table transport problem.

Parameter	Value
Population size	100
Max generation	200
Crossover rate	0.9
Mutation rate	0.1
Max tree depth	10

generalization performance of evolved programs. The training and test cases differ in the initial positions of the robots and the locations of the table, destination and obstacles.

All the robots use the same control program. In evaluating the fitness of robots we made a complete run of the program for one robot, before the fitness of another is measured. This is an efficient way of measuring the fitness, but is different from the real situation in which robots are moving at the same time. This is an advantage of subgoal evolution. Sequential execution of the program can detect various behavior patterns of parallel execution if a sufficient number of training cases are used.

Figure 18.5 shows the change in fitness values during a GP run with coevolutionary fitness switching: The fitness of a tree A is measured by $F(A) = f_1(A) + f_2(A)$, where $f_1(A)$ is the fitness for homing and $f_2(A)$ is the fitness for herding. A rapid decrease in fitness indicates the fast improvement in cooperative behavior.

We examined the evolution process of cooperative behavior by analyzing the performance of best programs at each generation. Shown in Figure 18.6 are the performance at generations $g = 1, 10, 14, 71$. At $g = 1$, a program was evolved that successfully moves some of the robots toward the table. But, no herding behavior was achieved. At $g = 10$ the robots tend to move toward the goal position but no group behavior is observed. At $g = 14$, three of the robots learned the herding behavior but one failed. This is possible

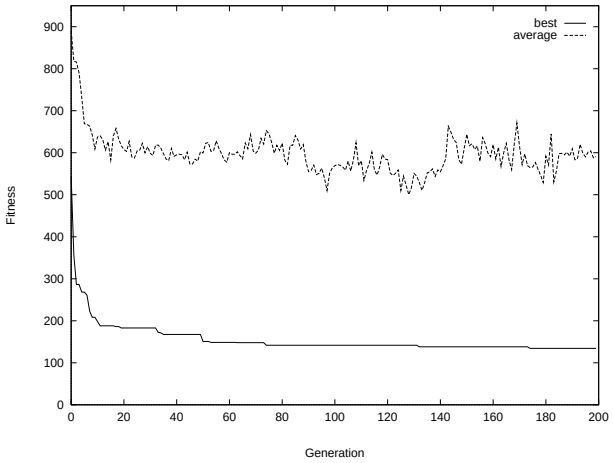

Figure 18.5
Evolution of fitness values during a GP run: (solid line) best fitness for each generation, (dotted line) average fitness for the individuals in each generation. Lower fitness means better individuals. A rapid decrease in fitness indicates fast improvement in cooperative behavior for table transport.

since, though having the same control program, the robots see different local environments. It took the robots 71 generations to learn perfect homing (i.e., every robot arrives at the table) and herding behaviors to transport the table to the destination.

Genetic programming with coevolutionary fitness switching was able to learn to solve the transport problem for more than one environments. The generality of the evolved programs was verified by running them on test environments. Figure 18.7 shows the behaviors of the robots for the test cases. Shown is the performance for four test cases out of 20. The composite cooperative behaviors can be observed: The robots start at their initial positions, getting together to the table (homing), moving in a herd to the goal (herding), and finally arriving at the destination (transporting).

The performance of genetic programs was measured by the number of hits: the number of times all the robots reached the destination. Figure 18.8 shows the change in the number of hits during the run.

Table 18.4 compares the number of hits for the three fitness switching methods described in the previous section. We made 10 runs for each method and measures the average values and their standard deviations. It is interesting to note that the naive approach only succeeded on 0.4 fitness cases in average out of 20 test cases in each run. As expected the fitness switching with sequential evolution, the most engineered version, was the best in number of hits for training and test. The coevolutionary switching method was competitive to the sequential switching in number of hits for both training and test performance. Relatively small values of standard deviation suggest that the results are statistically significant.

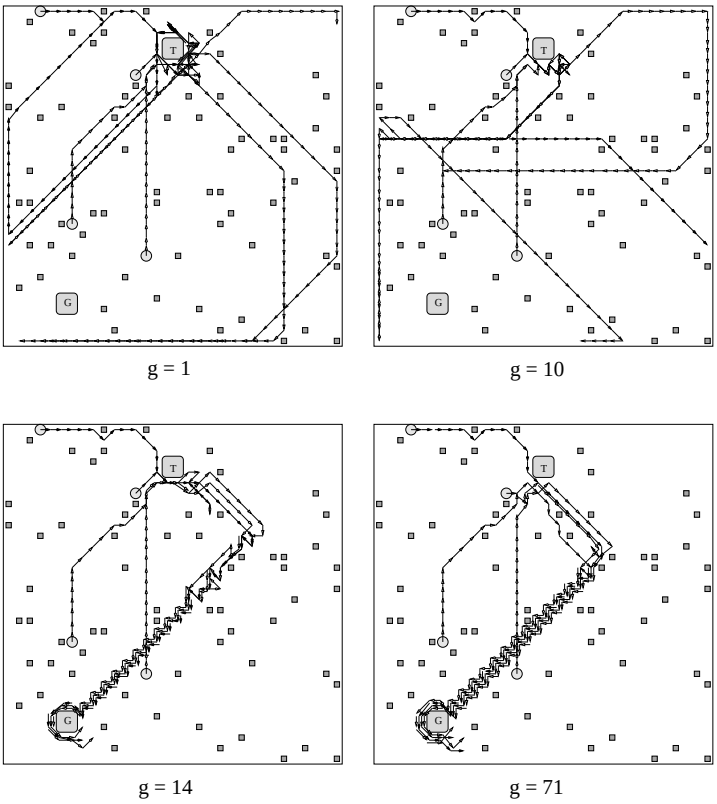

Figure 18.6
Trajectories of robots controlled by the best programs in the population. At generation one, three out of four robots succeeded in homing to the table (T), but they failed to arrive at the goal (G) position. At generation 10, all the robots succeeded in homing but still failed in herding. As generation goes on ($g = 14$, $g = 71$), they show successful homing and herding behaviors.

Table 18.4
Comparison of different evolution methods in terms of the number of hits for table transport. Experiments have been performed on 20 fitness cases for training and test, respectively. The values are averaged over ten runs. Also shown are the standard deviation. Coevolutionary fitness switching is competitive to sequential evolution, the most problem-specific approach, while the naive evolution method fails to solve this problem.

Method	Number of Hits	
	Training	Test
Naive	0.3 ± 0.458	0.4 ± 0.663
Sequential	15.1 ± 2.982	13.0 ± 1.949
Coevolution	13.9 ± 2.737	11.9 ± 2.546

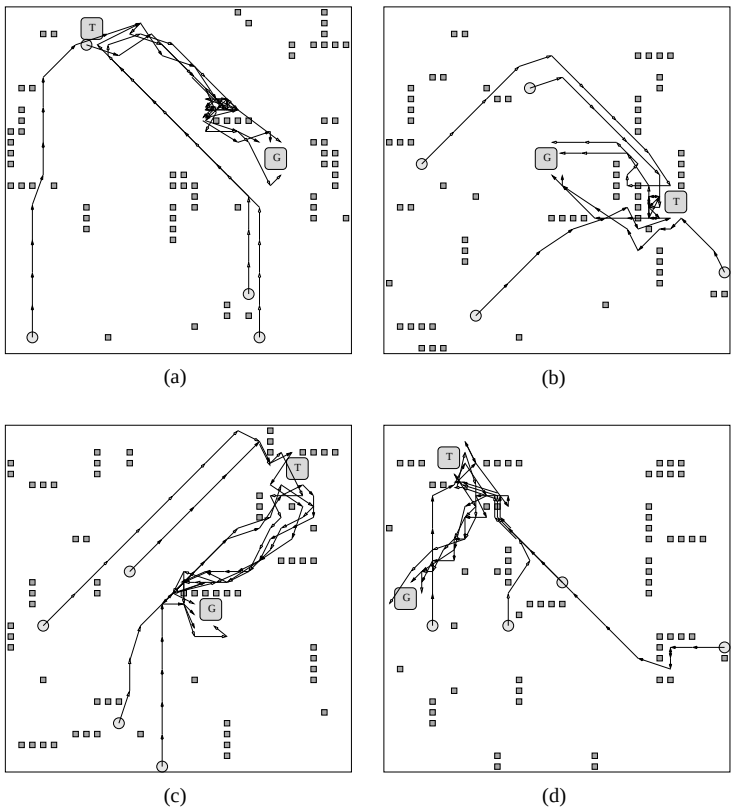

Figure 18.7
Trajectory of robots running the best program for the 4 test cases out of 20. Though the environments are different, robots have successfully performed homing and herding behaviors to transport the table to the destination. Here the environments have wall-like obstacles whose position and length were generated randomly.

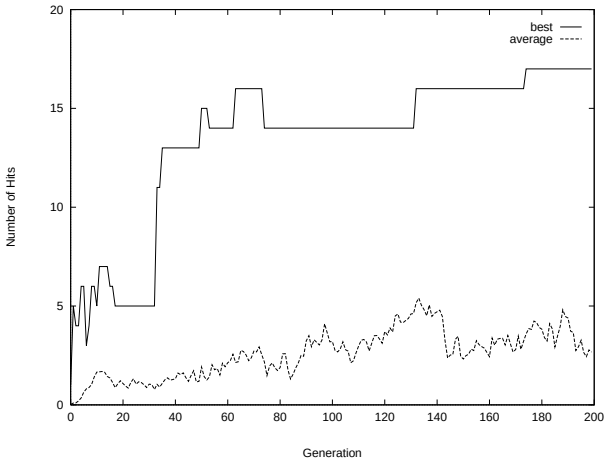

Figure 18.8
Number of hits vs. generation for a typical run. Despite the elitism, best fitness (and the number of hits) can decrease since the control programs have random moves as their primitive actions.

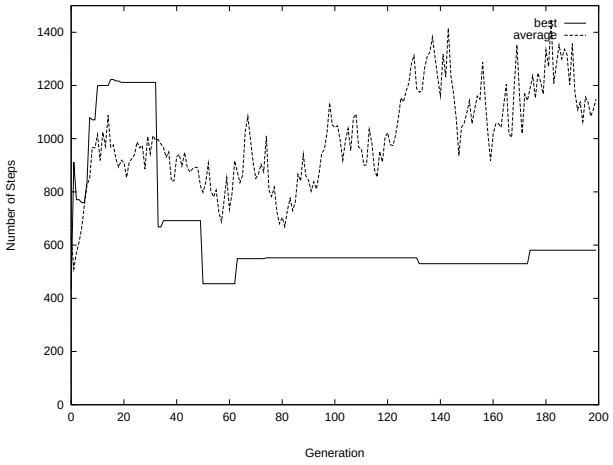

Figure 18.9
Average number of steps vs. generation. The solid line shows the number of steps taken by a group of 4 robots using the best program. The dotted line shows the average number of steps for a group of 4 robots, average taken over all the programs in the population.

Table 18.5
Comparison of the number of steps taken by a group of four robots for table transport. Shown are average values with standard deviation for the training set and test set, respectively. Also shown are the tree size in terms of the number of nodes. Coevolutionary fitness switching achieved competitive performance to the sequential evolution using more smaller programs.

Method	Average Number of Steps		Tree Size
	Training	Test	(#Nodes)
Naive	327.75 ± 156.11	697.18 ± 200.19	36.9 ± 15.34
Sequential	671.69 ± 140.01	523.39 ± 132.39	60.5 ± 21.95
Coevolution	553.72 ± 122.88	538.47 ± 107.75	37.8 ± 11.90

Figure 18.9 shows the evolution of the average number of steps made by four robots for 20 different training environments. The number of steps for a robot is defined as the number of primitive actions taken during the execution of the program. Shown are the best-of-generation and population-average values. Table 18.5 compares the performance of three different methods for fitness switching. The values given are the average number of steps made by a group of four robots for 20 different environments for training and test. Since genetic programs are shared for two micro-behaviors in naive evolution, robots have tried to remain around the table or destination. Thus, the average number of steps for training cases was very small, but they did not achieve the goal. In contrast, the most engineered sequential evolution method obtained the largest average number of steps because the control programs were fully evolved to fit the training cases. In the coevolutionary method, the average number of steps for training and test cases was moderate, indicating relatively reliable fitting to both the training data and the test data. It should be mentioned that, compared to the number of hits (Table 18.4), the standard deviation of the performance is relatively large. The table also shows the size of programs evolved by each method. The program evolved by coevolutionary fitness switching was the most sparse consisting of 37.8 nodes in average.

18.4 Application to Robotic Soccer

Robot soccer is an interesting challenge for artificial intelligence research on autonomous agents and multiagent learning [Kitano et al., 1997]. It involves two adversary teams of cooperative agents. To achieve the ultimate goal, each team needs to cooperate with teammates but its cooperation is hindered by the opponent team. Agents also need to generate subgoals, such as passing or intercepting the ball, and have to collaborate to achieve them in harmony with others. As the teammates and opponents are moving around in the field, the environment is dynamically changing. It is almost impossible for humans to develop a soccer program that takes into account every possible situation [Kitano et al., 1997].

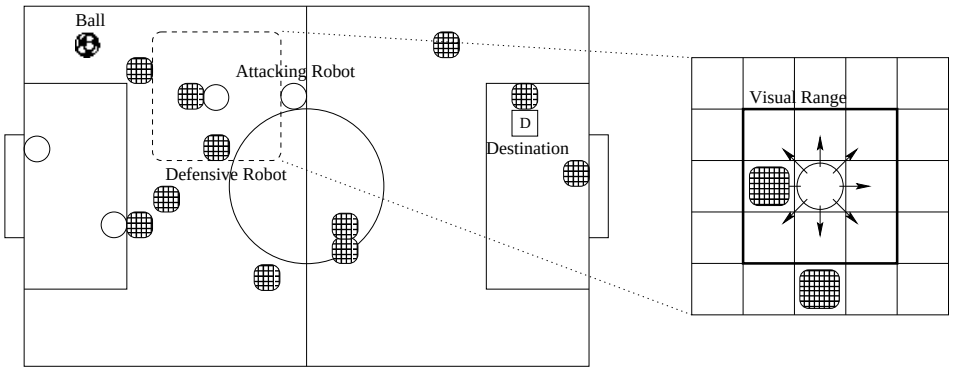

Figure 18.10
The environment for the dash-and-dribble problem. There are two teams and a ball in the 22×14 grid field. Attacking robots can move in 8 directions and have a limited visual field of range 1 to each movable direction.

We demonstrate the effectiveness of coevolutionary fitness switching in the context of a particular aspect of soccer game in the RoboCup domain [Cho and Zhang, 1998]. A fuller application of genetic programming to RoboCup Simulation League is reported in [Luke et al., 1997]. We focus on the dash-and-dribble behavior: given the ball position and its destination, a team of robot players is to approach to the ball and dribble it to the destination. In this task, a group of robot agents should cooperate to accomplish the task, otherwise they will get the ball intercepted by the opponent team.

We consider a soccer field of 22×14 grid points (Figure 18.10). There are two teams of robots and a ball in the field. The target position of the ball is given as D. Each team consists of 11 players. The objective of the attacking team, T_A, is to dash to and dribble the ball to the destination. The objective of the opponent team, T_B, is to hinder this attack.

The dash-and-dribble behavior is useful in several situations. The dashing behavior is used in a defensive mode; team T_A needs to dash to the ball to intercept the ball of the opponent team. The dribble behavior is required to pass the ball to the destination. In both cases group behaviors are required so that the respective behavior is performed effectively, that is, not to be intercepted by the opponents. Since not all players of team T_A need to take part in the attacking dash-and-dribble, we consider in the experiments the case of four attacking players. In contrast, all of the 11 opponent players are considered as obstacles.

The attacking robots move forward in the current direction (N, E, S, W, NE, SE, SW, NW) or remain in the current position. The defensive robots do not move, that is, remain in the current position (they are regarded as obstacles). The attacking robots are all have the same sensors and actuators (homogeneous agents) and have a limited visual field of range 1 to each movable direction.

Table 18.6
The terminal and function set used in the dash-and-dribble problem. This is similar to Table 18.2.

	Symbol	Description
Terminals	FORWARD	Move one step forward in the current direction.
	AVOID	Check the surrounding region and make one step in the first direction possible.
	RANDOM-MOVE	Make a movement in the random direction.
	TURN-BALL	Make a clockwise turn to the nearest direction of the ball.
	TURN-DESTINATION	Make a clockwise turn to the nearest direction of the destination.
	STOP	Make no step and remain in the current position.
Functions	IF-OPPONENT	Return true if an opponent is adjacent to the current block, else false.
	IF-MATE	Return true if a mate is adjacent, else false.
	IF-BALL	Return true if a ball is adjacent, else false.
	IF-Destination	Return true if the destination is adjacent, else false.
	PROG2	Execute two subtrees in sequence.
	PROG3	Evaluate three subtrees in sequence.

The attacking robots activate their control program to run a team trial. At the beginning of the trial, they have different positions and orientations which are chosen randomly. During a trial they should approach and move the ball to the destination in cooperation with others.

This problem has similarities to the table transport problem (TTP) in several aspects. Dashing to the ball can be considered as a homing behavior in TTP, and dribbling the ball to the destination can be regarded as a herding behavior in TTP. In our experiments there are four robots which perform the given task and eleven defensive robots. The terminals and functions used in the dash-and-dribble problem are shown in Table 18.6. Fitness functions for dash and dribble behaviors are similar to Equation (18.1) and (18.2) except for the target positions. A total of 20 training cases are used for evolving programs. A total of 20 different worlds are used for evaluating the generalization performance of the evolved programs. Other parameter values are the same as in the previous section (Table 18.3).

Figure 18.11 shows the evolution of fitness values during a GP run. A rapid decrease in fitness during early generations indicates a fast improvement in cooperative behavior. The training resulted in a program tree whose depth is 8 and the number of nodes is 37 in the 187th generation. Figure 18.12 shows the change in the number of hits during the run. The number of steps needed for dash-and-dribble behavior is shown in Figure 18.13.

The generality of the programs evolved was verified by running them on test environments. Figure 18.14 shows the behavior of the robots to some test cases. Though the environments are different, robots successfully perform the dash-and-dribble behavior. Table 18.7 compares the performance of three methods for the training and test cases. The naive evolution failed to find the appropriate control program. The sequential evolution showed the best results in number of hits for both cases. The coevolutionary fitness switching was competitive to the sequential method.

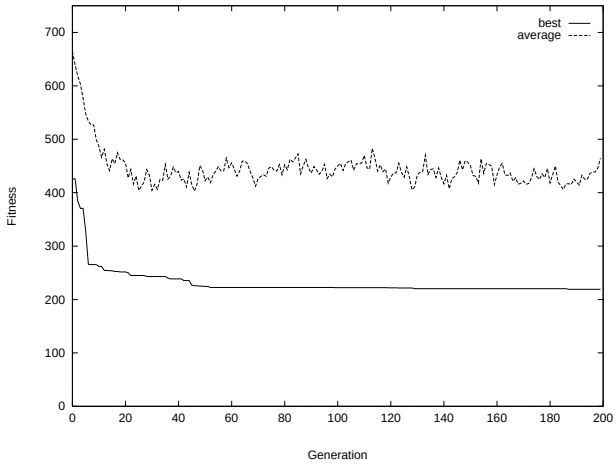

Figure 18.11
Evolution of fitness values during a GP run for the dash-and-dribble behavior: (solid line) best fitness for each generation, (dotted line) average fitness for the individuals in each generation. Lower fitness means better individuals.

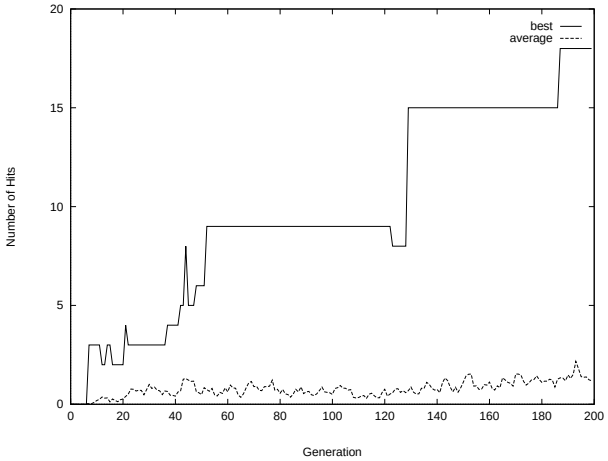

Figure 18.12
Number of hits vs. generation for the dash-and-dribble behavior. Despite the elitism, the number of hits can decrease the control programs have random moves as their primitive actions.

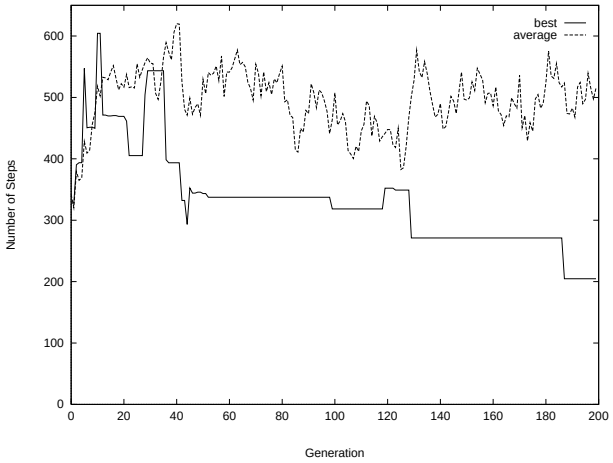

Figure 18.13
Number of steps vs. generation for the dash-and-dribble behavior. The solid line shows the number of steps taken by a group of 4 robots using the best program. The dotted line shown the average number of steps for a group of 4 robots, average taken over all the programs in the population.

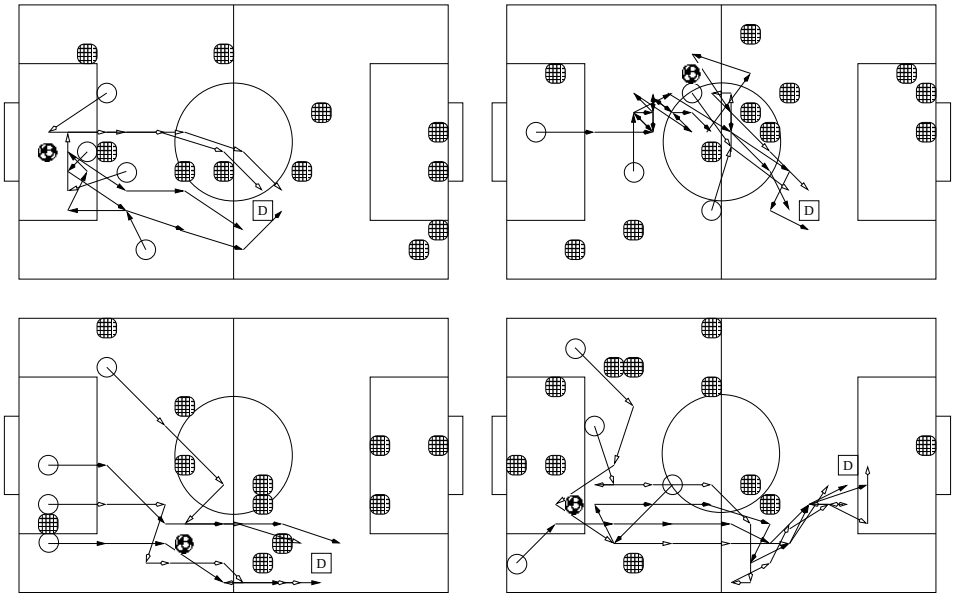

Figure 18.14
Trajectories of robots running the evolved programs on the 4 test cases. Robots successfully dashed to the ball and dribbled it to the destination D.

Table 18.7
Results for the training and test cases on the dash-and-dribble problem. The values are averaged over ten runs. Also shown are the standard deviation.

Method	Number of Hits	
	Training	Test
Naive	0.0 ± 0.000	0.0 ± 0.000
Sequential	17.0 ± 2.366	13.9 ± 1.700
Coevolution	13.0 ± 3.162	10.9 ± 3.673

18.5 Conclusions

We described a method for evolving composite cooperative behaviors of multiple robotic agents. This method, called coevolutionary fitness switching, is based on a pool of fitness functions which are defined to reflect the problem structure but without too much need for domain knowledge. The method was motivated by the observation that, while GP is able to evolve emergent behaviors, the evolution can be more efficient if the program structure and sometimes the evolution strategy is constrained to match the problem structure.

In the context of the table transport problem we have experimentally shown that coevolution with fitness switching can solve a class of tasks which we were not able to efficiently solve using simple genetic programming. Simulation results also show that, compared with the carefully designed sequential evolution method, the coevolutionary fitness switching is competitive in training and test performance. We also observed that coevolutionary fitness switching generally produced more smaller solution trees. It seems that coevolutionary fitness switching has the effect of avoiding overfitting to one specific behavior, thus resulting in small trees, while sequential evolution tends to find large trees overfit to specific subgoals.

The method was also applied to learn the dash-and-dribble behavior for soccer robots. Here each genetic program consists of several subroutines which are coordinated to perform the entire task. Each subroutine is responsible for a group behavior and it is evolved by a specific fitness function. The coevolutionary fitness switching method is then used to coordinate the evolution of the complex macro behaviors from the primitive micro-behaviors. Our experimental results demonstrate that it is feasible to evolve by genetic programming some complex group behaviors which are useful for solving specific aspects of robotic soccer games.

Acknowledgements

This research was supported in part by the Korea Science and Engineering Foundation (KOSEF) under grants 96-0102-13-01-3 and 981-0920-350-2. Thanks to William B. Langdon and Peter J. Angeline for comments on the previous versions of this paper.

Bibliography

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998), *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann.

Bennett III, F. H. (1996), "Automatic creation of an effi cient multi-agent architecture using genetic programming with architecture-altering operations," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 30–38, Stanford University, CA, USA: MIT Press.

Bruce, W. S. (1995), *The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs*, PhD thesis, School of Computer and Information Sciences, Nova Southeastern University, 3100 SW 9th Avenue, Fort Lauderdale, Florida 33315, USA.

Cho, D.-Y. and Zhang, B.-T. (1998), "Learning soccer-robot cooperation strategies using genetic programming," Technical Report SCAI-98-009, Artifi cial Intelligence Lab (SCAI), Dept. of Computer Engineering, Seoul National University.

Digney, B. L. (1996), "Emergent hierarchical control structures: Learning reactive/hierarchical relationships in reinforcement environments," in *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior: From animals to animats 4*, P. Maes, M. J. Mataric, J.-A. Meyer, J. Pollack, and S. W. Wilson (Eds.), pp 363–372, Cape Code, USA: MIT Press.

Haynes, T., Sen, S., Schoenefeld, D., and Wainwright, R. (1995), "Evolving a team," in *Working Notes for the AAAI Symposium on Genetic Programming*, E. V. Siegel and J. R. Koza (Eds.), pp 23–30, MIT, Cambridge, MA, USA: AAAI.

Iba, H. (1997), "Multiple-agent learning for a robot navigation task by genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo (Eds.), pp 195–200, Stanford University, CA, USA: Morgan Kaufmann.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E., and Matsubara, H. (1997), "Robocup: A challenge problem for ai," *AI Magazine*, 18(1):73–85.

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

Kube, C. R. and Zhang, H. (1993), "Collective robotics: From social insects to robots," *Adaptive Behavior*, 2(2):189–218.

Langdon, W. B. (1998), *Data Structures and Genetic Programming: Genetic Programming + Data Structures = Automatic Programming!*, Boston: Kluwer.

Luke, S., Hohn, C., Farris, J., Jackson, G., and Hendler, J. (1997), "Co-evolving soccer softbot team coordination with genetic programming," in *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artifi cial Intelligence*, H. Kitano (Ed.), pp 115–118.

Luke, S. and Spector, L. (1996), "Evolving teamwork and coordination with genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 150–156, Stanford University, CA, USA: MIT Press.

Mataric, M. J. (1996), "Learning in multi-robot systems," in *Adaptation and Learning in Multi-Agent Systems*, G. Weiss and S. Sen (Eds.), volume 1042 of *LNCS*, pp 152–163, Springer-Verlag.

Qureshi, A. (1996), "Evolving agents," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds.), pp 369–374, Stanford University, CA, USA: MIT Press.

Werner, G. M. and Dyer, M. G. (1993), "Evolution of herding behavior in artifi cial animals," in *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, J. A. Meyer, H. L. Roitblat, and S. W. Wilson (Eds.), pp 393–399, Honolulu, Hawaii, USA: MIT Press.

Zhang, B.-T. and Cho, D.-Y. (1998), "Fitness switching: Evolving complex group behaviors using genetic programming," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (Eds.), pp 431–438, University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann.

Zhang, B.-T. and Hong, Y.-J. (1997), "A multiple neural architecture for evolving collective robotic intelligence," in *Proceedings of 1997 International Conference on Neural Information Processing*, N. Kasabov, R. Kozma, K. Ko, R. O'Shea, G. Coghill, and T. Gedeon (Eds.), pp 971–974, University of Otago, Dunedin, New Zealand: Springer.

Zhang, B.-T., Ohm, P., and Mühlenbein, H. (1997), "Evolutionary induction of sparse neural trees," *Evolutionary Computation*, 5(2):213–236.