Automatic Creation of Team-Control Plans Using an Assignment Branch in Genetic Programming

Walter A. Talbott

Stanford Symbolic Systems Program
Stanford University
Stanford, California 94305
wtalbott@stanford.edu

Abstract. This paper is concerned with the introduction of a method for allowing genetic programming to automatically create team-control plans using an assignment branch. Team-control plans are representations of the composition and behavior of groups of agents and include the definition of one or more roles. Genetic programming is a general problem solving technique, and using genetic programming as a means for creating team plans would allow the user to specify very little, while producing effective results. I propose to show that the use of what I call an assignment branch in genetic programming provides a robust and general way to approach team composition and role definition problems.

1 Introduction

The creation of team-control plans is a problem that has become more pertinent as the power of computing has increased and as the sophistication of artificial intelligence methods has led to the improved ability to design situated agents. A team-control plan defines how a group of these agents act in combination with one another to solve problems and accomplish complex tasks. For most of these team plans, human designers must specifically define the way that the team will act. Because of the high level of human involvement, these teams are not as artificially intelligent as they could be. Genetic programming has recently emerged as a robust, domain-independent method for automatically generating the solution to difficult problems. It seems natural, therefore, to use the power of genetic programming for team organization and design problems.

A suitable problem for the exploration of genetic programming's aptitude for creation of team plans will ideally be impossible to solve without teamwork of some kind, and will require the creation of different roles. I define a role as a set of instructions that a subset of the team will carry out. If the team were intended to play soccer, for example, there would be roles for each position: goalkeeper, defense, and offense. Roles allow for much more elaborate team plans, and hence for the solution of much more elaborate problems, but are proportionately more difficult to create without much human intervention. Luke and Spector (1996), for example, apply genetic programming to multiple role teamwork problems, but require the number of roles to be selected beforehand. This paper will explore a method for automatically creating team plans, number of roles included, with genetic programming, using an artificial agent problem as an illustrative, if simple, problem domain.

K. Deb et al. (Eds.): GECCO 2004, LNCS 3102, pp. 201–212, 2004.
© Springer-Verlag Berlin Heidelberg 2004

1.1 The River Crossing Problem

The problem consists of a nine-by-nine toroidal world, in which there are a number of artificial agents that can move and interact with food pellets. Each agent can face in one of four directions, can be carrying either one or zero pieces of food at a time, and has an assigned role. The exact functions for control of the agents are described in Tables 1-3. The world has one strip that I define as the goal strip, to which the agents must move all the food pellets. The goal strip is surrounded on both sides by two rows of water, and this water is deadly to the agents. When an agent moves onto a square in the water, the agent dies, and can no longer gather food or move in any way. The dead agents, however, form a bridge for the remaining agents, such that if an agent moves onto a water square that has already been visited by another agent, the current visitor can continue to move as if it were on land. The agents are given a limited number of time steps to move the food, both so that the genetic programming run will eventually stop, and so that solutions are reasonably quick. Figure 1 shows an example world after an intermediate number of time steps.

The problem is designed to promote the need for teamwork, since no single agent could move even one pellet to the goal strip. Also, the problem is designed to facilitate a solution involving more than one role, but this facet of a solution is not strictly necessary. A solution, for instance, might benefit from designating a certain number of agents to the creation of a bridge, and the remaining agents to the collection of food. Because the creation of roles is not strictly necessary, I can also use this problem to compare solutions involving multiple roles to solutions with only one role. The success of the method in creating multiple roles is described further in the Results section below.

This paper is organized as follows. It first presents the methods used for representing the problem appropriately for genetic programming. Then, it presents the results gathered from four different experiments, and discusses these results. Finally, it suggests areas of possible future work and areas to which the results might be applied.

Fig. 1. Shows a world with four agents and five pieces of food. Two food pellets have successfully been placed on the goal strip.

2 Methods

Approaching the river-crossing problem and team control problems in general, with genetic programming methods offers several benefits. If biology is any example, it is

clear that the basic principles of genetic programming have already been successful in nature; look at any pack of predators, or any school of fish. Genetic programming is an algorithm that uses the selection pressures found in natural systems to improve a randomly generated population of programs. These programs are built up from sets of terminals and functions into trees, as described by Koza (1992). Expressing the control of artificial agents with the population of tree-like programs for which genetic programming calls is relatively straightforward. Following the example of using zero-argument functions as terminals set by Koza's work with artificial ants (1992), I describe the movement of agents with the terminals outlined in Table 1.

Table 1. Description of zero-argument functions used as terminals.

Terminal Name	Description
MoveForward	Moves the agent one square in the direction that it is facing.
TurnLeft	Rotates the agent 90 degrees counter-clockwise.
TurnRight	Rotates the agent 90 degrees clockwise.
PickUpFood	Picks up food if the agent is currently on a square with food. Each agent can only carry one pellet at a time.
DropFood	Drops food on the current square if the agent is carrying food.
NoOp	The agent remains exactly how it is.

Table 2. Description of the functions used in the program tree

Function Name	Description
Prog2	Executes the two arguments in succession.
Prog3	Executes the three arguments in succession.
IfAtFood	Executes the first argument if the agent is currently on a square with food, and the second otherwise.
IfWaterAhead	Executes the first argument if the agent is facing a square with water and no agent bridge, and the second otherwise.
IfCarryingFood	Executes the first argument if the agent is carrying food, and the second otherwise.
IfGoalStrip	Executes the first argument if the agent is currently on the goal strip, and the second otherwise.
IfBridgeAhead	Executes the first argument if the agent is currently facing a square with an agent bridge, and the second otherwise.
ADF0	Executes an automatically defined function (Koza, 1992) that can use all previously listed functions.
ADF1	Executes an automatically defined function that can use all previously listed functions, including ADF0.
ADF2	Executes an automatically defined function that can use all previously listed functions, including ADF0 & ADF1.

With these terminals defined, the programs can use the functions in Table 2 to further control the action of the agents under certain conditions.

Using these terminals and functions, the experiments described in this paper define an individual in two different ways. One group of experiments uses a tree structure where only one role is allowed. The result-producing branch of the tree represents this role, and is executed once for each agent at every time step. The result-producing branch can call any of the automatically defined functions, which are each in turn represented by their own branch of the tree.

It seems, however, that at least the ability to support more than one role would expand the range of possible problems to which genetic programming could be applied, and make more powerful the solutions to the problems it can already create. Towards that end, the other group of experiments involves the use of an entirely separate branch, whose function set is entirely different than the result-producing branches. This is the assignment branch, and allows the genetic programming run to assign different numbers of agents to different roles. For the purposes of these experiments, up to three roles were allowed, and were represented by three result-producing branches. Proposed methods for automating the number of roles entirely are discussed in the Future Work section below. The functions of the assignment branch are defined in Table 3. During evaluation of the program, the assignment branch is run first, to set up the role of each agent in the world, and also how many total agents are present. These experiments cap the number of agents due to time constraints on each run. The assignment branch removes the necessity for team plans to rely on brute-force repetition of the same set of instructions for each agent. Simultaneously, it allows for efficiency of specification, since agents can be grouped together rather than defined independently and with a high chance of redundancy.

Table 3. The Functions and Terminal of the Assignment Branch

Function Name	Description
Role1	This function takes one integer argument, and inserts that number of agents into the world, all of which are assigned to role 1. If the maximum number of agents has already been assigned, this function does nothing.
Role2	This function takes one integer argument, and inserts that number of agents into the world, all of which are assigned to role 2. If the maximum number of agents has already been assigned, this function does nothing.
Role3	This function takes one integer argument, and inserts that number of agents into the world, all of which are assigned to role 3. If the maximum number of agents has already been assigned, this function does nothing.
Prog2	Executes the two arguments, and arbitrarily returns the value of the first argument.
Random Constant	The only terminal for the assignment branch, the random constant is an integer that is used to specify how many agents of each role are present.

2.1 Sample Branches

The following are illustrations of the structure of the branches used to represent each agent. They have been simplified greatly, and do not come close to solving the river-crossing problem, but serve to highlight the difference in structure between the result producing branches and the assignment branch. The automatically defined functions have been left out for simplicity, since they are nearly identical to a result producing branch.

Result-producing Branch:

```
(Prog2 (IfGoalStrip DropFood MoveForward)
       (Prog2 (IfAtFood PickUpFood MoveForward)
TurnLeft))
```

This branch represents an agent that, at each time step, will detect whether or not it is on the goal. If it is, it will attempt to drop food, the success of which depends on whether it is carrying food. If it is not, it moves one square forward. Then, regardless of the outcome of the goal test, the agent will test whether it is at food, pick it up if it is, and move forward otherwise. To end the time step, the agent will always turn left. The result producing branches are limited to a depth of 17, which did not seem to eliminate possible solutions. A depth of 17 allows a program with up to 3^{17} = 129,140,163 nodes, which should be far more than sufficient for this problem.

Assignment Branch:

```
(Prog2 (Role1 (Role2 5)) (Role3 (Role3 2))
```

The assignment branch listed above assigns 5 agents to roles one and two, and 4 to role three, for a total of 14 agents. Each Role function returns the same integer that it takes as an argument, so the assignment branch chains backwards from the last node to the first, stopping when the maximum number of allowed agents has been reached. Note that, because constants are terminals, it is possible to have an expression such as:

```
(Prog2  4  5)
```

In such a case, no agents are assigned, and the individual would receive the worst possible fitness since none of the food can move itself. Because the assignment branch is only run once per individual, and because individuals such as this are quickly eliminated from the population, this quirk is acceptable.

2.2 Fitness Evaluation

Since it is desirable to generate a solution that is general enough to solve the river-crossing problem independent of where the food is located, the fitness is evaluated based on two fitness cases. The first case scatters the food across the squares of the board, and the second places all of the food in one square. The raw fitness of each program is just the number of food pellets that ended up on the goal strip, which is also the number of hits. A run terminates successfully when an individual has a raw fitness equal to the number of food pellets that exist in the world, over all fitness cases. Another experiment also evaluated fitness based on the number of steps it took to complete; raw fitness was the total number of food pellets present over all fitness cases minus the amount of food placed on the goal strip, plus the total number of steps

it took. If a program did not get all the food pellets of a particular fitness case onto the goal strip, it could not complete that case in less than the maximum number of steps allowed, and so only those individuals who were relatively fit to begin with could gain from being faster. Because of this, the fitness evaluation with steps included can distinguish between individuals who would otherwise seem similar. This discrimination allows the comparison between successful individuals of the two types described earlier: those limited to one role, and those with the assignment branch.

2.3 Breeding and Run Parameters

These experiments use the breeding phases that were provided as default with lil-GP, the genetic programming software package in which the experiments were carried out. Each phase consists of tournament-selection for crossover 90% of the time and reproduction 10% of the time. Crossover could happen only between similar branches of the two parent trees. This is to isolate the evolution of the individual roles in the assignment branch individuals. Since they are somewhat structurally complex, inter-branch crossover might slow the progress of the evolution because it would introduce more noise into the process, which is not necessary for a problem of this magnitude. Mutation was not included, though it arguably should have been. Especially in the assignment branch, mutation could have helped fine-tune the random constants that were present, rather than forcing the program to rely on those constants generated at initialization.

Table 4. Tableau for the river crossing problem

Objective:	To produce a solution to the river crossing problem that uses the assignment branch, and compare with a solution that only uses one role.
Terminal set:	MoveForward, TurnLeft, TurnRight, PckUpFood, DropFood, NoOp, as defined earlier.
Function set:	One Role Individuals, defined in table 2. Assignment Branch Individuals, defined in table 3.
Fitness cases:	Two, one in which the food is scattered across the world, and one in which the food is stacked in one square
Raw fitness:	In some runs, raw fitness is the number of food pellets deposited on the goal strip. In others, it is the number of pellets on the goal strip weighted by the steps it took to get them there.
Hits:	Each pellet that ends up on the goal strip is counted as a hit.
Wrapper:	None.
Parameters:	M = 13,000 G = 95
Success predicate:	When an individual placed all food pellets on the goal strip, the run was terminated successfully, except on the runs where time mattered in the fitness, in which case the runs continued until all generations completed.

These runs were done with population size M = 13,000 and allowed to run until G = 95 generations. The population size, in some of the early runs, was set as high as 75,000, but each of these runs took up to about 48 hours. The reduced population size

decreased each run to about 10 hours, and seemed sufficient for the problem. Because the structure of the programs is fairly cumbersome, with seven branches total in the individuals with the assignment branch, the number of generations has to be high enough to allow the single-node crossover operator to rearrange enough of the branches to form a good solution. In fact, 95 may be too low, but it was sufficient for these experiments. Table 4 presents the tableau summary of the method just described.

3 Results

The experiments lend themselves to a categorization into eight different groups. Each group will be presented in turn, and the results will be discussed in section 4 of the paper, below. The individuals limited to one role will be called simply one-role individuals, and those with the assignment branch will be called branch individuals. The fitness evaluation that takes only hits into account will be called basic fitness, whereas the evaluation that includes time steps will be called step fitness. The final difference between runs involved what happened when an agent that was carrying food hit the water. Originally, the agents would not be allowed to drop their food before they died and could no longer act. This required any fully successful plan to specify that no agent could ever hit the water if it was carrying food, which proved to be fairly difficult. And intuitively, this requirement seems to suggest the use of multiple roles, one for solely forming the bridge and leaving food alone, and one for gathering food. When this requirement was relaxed, agents that hit the water automatically dropped their food so that others could resume in their place. Runs where the requirement was in place will be called no-drop runs, and those without the requirement will be called drop runs. In every experiment, two fitness cases were used, with 25 food pellets each, which makes the highest possible hit score 50. In every experiment with a step score included, 150 steps were the maximum for each fitness case, so at most 300 is added to the difference between 50 and the number of hits to give the raw fitness score. A lower fitness score is better, with 0 being the best.

3.1 Basic Fitness with Drop

In all experiments conducted with basic fitness and drop, a solution was found before the 20th generation. Both one role and branch individuals solved the problem easily, and there were about as many single-role branch individuals as not. This group's results bear mentioning only for the sake of completeness.

3.2 Step Fitness with Drop

Given that the drop problem can be solved easily, I present the results of step fitness runs that allowed drowning agents to drop food in table 5.

Table 5. Results for Step Fitness Runs With Drop

	Generation of Best of Run	Nodes	Hits	Fitness
One Role	64.75	567	43.75	107.75
Branch	86	637	46.67	80.6

The numbers in the table are averages over all the runs conducted in this group. The generation of the best of run does not necessarily imply that a solution was found at the listed generation. The number of nodes is a good measure of structural complexity. It should be noted that in these runs, hits is not the same as fitness, because fitness is defined as the number of steps it takes the individual to complete its task. It is somewhat unfair to define solution as 50 hits in runs where steps contribute to the fitness function, because genetic programming is blind to hits, and only selects based on fitness. However, because individuals that get all the food to the goal strip in both fitness cases have such an advantage in the fitness measure, I are fairly safe defining solution as 50 hits. Out of the runs that did solve the problem, all but two resulted in fitness ratings below 50, and one of the one-role runs produced an individual with a fitness of 16. Of the branch individuals that solved the problem, 2 presented solutions with two distinct roles, and 2 presented solutions with only one.

3.3 Basic Fitness with No Drop

The problem changes drastically when no drowning agent is allowed to drop the food that it may be carrying. Of all the runs performed, only one produced a solution, though most came close. An assignment branch run generated the single solution, and the individual is partially presented below. All assignment branch runs resulted in the production of two roles. Table 6 outlines the results of these runs.

Table 6. Results for Basic Fitness Runs With No Drop

	Generation of Best of Run	Nodes	Hits	Fitness
One Role	35.25	354	41.5	8.5
Branch	55.75	424.5	43.5	6.5

This case is the most difficult for the process to solve, so the runs that produced the best results deserve closer inspection. Figure 2 and Figure 3 present a generation-by-generation breakdown of the hits from the best runs of the One Role and Branch tests. Figure 2 shows the highest number of hits achieved by any individual at each generation, and Figure 3 shows the mean number of hits over all individuals in each generation. Notice that the assignment branch run ended at generation 26, after which it terminated since the success predicate was achieved. The one role run stopped at generation 95, after failing to fulfill the success predicate. Both figures stop at generation 48, because the best individual from the entire one-role run was created in that generation.
Because only one individual managed to solve this problem, I will examine its assignment branch, printed in its entirety below.

```
ASSIGN:
 (role1 (prog2 (role1 (prog2 (role1 3) 4))
               (role1 (role3 (prog2 (role1 (role1 1))
3)))))
```

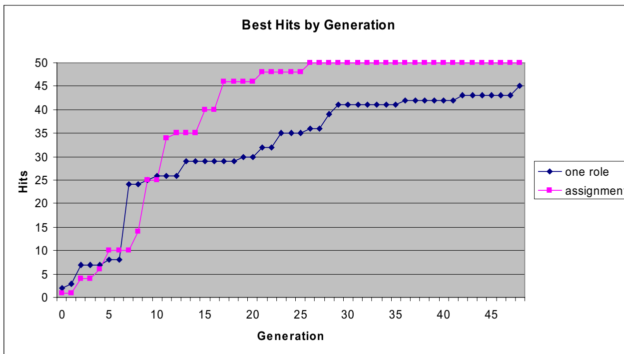

Fig. 2. Shows the highest hits achieved by an individual at each generation in the run.

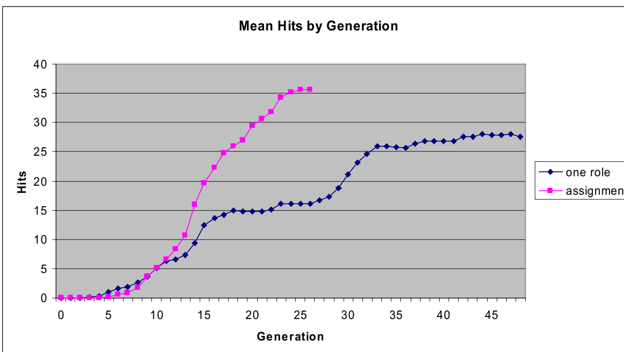

Fig. 3. Shows the mean hits achieved at each generation in the run. Notice that the assignment branch run stops at generation 26, when a solution was found.

The assignment branch has the effect of introducing 13 agents into the world, interestingly 7 less than the maximum of 20. 12 of these are role one agents, and 1 is a role three agent. Because the rest of the program is so complex, I will forego an analysis of the functionality of the program, letting it suffice to say that both role one and role three are substantive, and proscribe different courses of action for the agents they control. Because these roles are generated genetically, they do not define roles as humans might, with an explicit purpose for each, but instead piece together ideas that work to form something that is not intuitive to a human observer.

3.4 Step Fitness with No Drop

Though it is clear that a solution to the no drop restriction is very hard to produce, this next group defines its fitness by how quickly the agents can complete their task. Therefore, the hits are not as important as the overall fitness, since there is no selection pressure generated based on the number of hits. Although hits increase the fitness, they contribute only a small amount in comparison to the number of time steps taken. The results from these experiments are presented below in Table 7.

Table 7. Results from Step Fitness runs with No Drop

	Generation of Best of Run	Nodes	Hits	Fitness
One Role	59	406	35	270
Branch	56	342	29	251

Here I again run into the unfairness of defining a solution as 50 hits, because the runs that do not allow agents to drop food before they die are difficult to solve. This difficulty in getting individuals who can solve both fitness cases drives the population towards individuals who can quickly solve one of the fitness cases, and who do not necessarily perform well on the other. One possible, but untested solution to this is to alter the fitness function so that it only subtracts steps from the total fitness if all fifty hits have been achieved.

4 Discussion

These results show that, in all meaningful categories, assignment branch runs outperformed runs with only one role. The one category in which there was no improvement was not a difficult problem at all, and both solved it easily.

In general, the ratio of fitness of the assignment branch best-of-run individuals to one-role individuals was 1.2. In such a small problem, this increase is significant. Given that solutions can be found with only a single role, it is satisfying to note that an approach with less human involvement than the explicit creation of one role can provide such an increase in performance. Intuitively, note that solutions that allow more than one role are more likely to solve the problem since they can represent more complex tasks. The results presented in Figure 3 show the improvement over the one role method most clearly. The mean number of hits is much higher in the assignment branch runs. It seems reasonable to assume that had the run continued, the mean would have continued to increase. This is a good sign, because a higher mean suggests a more likely solution, and suggests that the assignment branch runs outperform one role runs. Perhaps it would have been better to observe the difference between a run that explicitly enforced the existence of two roles and the assignment branch runs. After running a few preliminary trials in explicitly enforcing more than one role, there still seems to be an improvement in fitness with the assignment branch runs over the hard-coded runs. This is most likely due to the fact that with the assignment branch, the genetic programming run can tailor a solution to the problem at hand, including both the number of agents in each role, and the total number of agents needed. Runs where the user must specify these parameters are confined to what is not necessarily the optimal configuration, or even a configuration that would allow a solution at all. The more general approach of the assignment branch is an attractive benefit. However, since the ultimate goal of this paper is to provide a method for applying genetic programming to automatically generate team plans, the increase in performance over even the simpler representation is a surprising bonus, and I would have been happy with comparability in performance.

It is interesting to observe that the generation at which the best of run individual was created was lower, with few exceptions, such as in the examples of figure 2 and 3, in individuals from the runs limited to one role. This suggests that one role individuals more quickly converge on their best answer. Runs with the assignment branch, since they are much more structurally complex, take longer for the crossover operation to shuffle meaningful parts of the tree around. This increase in the number of generations needed for the distribution of genetic information in the assignment branch runs does not seem a drastic setback to the assignment branch's utility in the creation of team plans, especially when weighed against the clean-hands approach and the benefit in fitness that the assignment branch affords. In fact, it could be the relative slowness of the assignment branch runs to converge that gives them an advantage in fitness. The one role runs might prematurely converge on a suboptimal result, and have no way to recover. However, most of the evidence seems to point to the fact that the assignment branch runs perform better because they allow for the automatic configuration of both the roles and the total number of agents present in each role.

5 Conclusions

I have shown that using an assignment branch to create team plans in the domain of artificial agents not only works as a method of removing responsibility from the human user of genetic programming, but also outperforms methods where the number of roles and number of agents in each role is set by the user before the run begins. My method leverages the innate power of genetic programming to tailor solutions closely to the problem, and though it may require longer runs to generate solutions, the tradeoff of having the computer automatically generate all aspects of the solution is a tempting one. Also, the assignment branch seems to have potential as a method for harnessing the power of genetic programming for the specific domain of multi-agent and teamwork problems.

6 Future Work

Though I have already shown some benefit to the assignment branch approach, it would be valuable to subject it to a more rigorous test. First, it would be beneficial to try the experiment using architecture altering operations (Koza, 1994) to allow the run to create the branches for each role as needed. This would require dynamic alteration of the function set of the assignment branch, and also an incorporation of mutation for at least the assignment branch so that the new role could have a chance to be incorporated into the solution. This would, of course, increase the time necessary for each run, since it might get stuck waiting for a mutation to occur, and would introduce difficulties if crossover occurred between individuals that had different roles defined. However, the benefit seems worth the potential difficulties.

Further work should be done in expanding this approach to more complex and demanding teamwork problems, such as the control of search and rescue teams, or of robot soccer teams. Also, introduction of support for genetically-defined communication between members of the team would be an interesting endeavor.

References

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

Koza, John R. 1994d. *Architecture-Altering Operations for Evolving the Architecture of a Multi-Part Program in Genetic Programming*. Stanford University Computer Science Department technical report STAN-CS-TR-94-1528. October 21, 1994.

Luke, Sean and L. Spector. 1996. "Evolving Teamwork and Coordination with Genetic Programming". In *Genetic Programming 1996: Proceedings of the First Annual Conference*. 141-149.