Evolving Programs for Distributed Multi-Agent Configuration in Two Dimensions

Rob P. DeConde
Stanford Biomedical Informatics Program
Stanford University
344 Olmsted Road, Apt. 431
Stanford, California 94305
650-804-1342
rdeconde@smi.stanford.edu

ABSTRACT

This paper demonstrates how using a minimal function set and relying only on local data, programs can direct groups of agents to configure themselves into 2-dimensional configurations. Incorporating a branching node in the function set provides the otherwise homogeneous agents a form of state and memory which allows for the emergence of complex behavior from the combination of elementary elements. The strategies learned by the genetic program are complex and in some cases not readily accessible to human intuition.

1. Introduction

Genetic programming (GP) has proven to be a widely applicable method for solving problems in a broad range of fields (see Koza, 1992), producing human-competitive results using a minimum of prior information. While in some instances more specialized or domain-specific methods may perform better or faster, GP provides a very flexible and general approach to almost any problem with results competitive—in many cases—to the best of specialized algorithms (Koza, 1992). Beyond the algorithm's flexibility as a general method, one of GP's greatest strengths is its ability to solve problems without the user needing specific knowledge of how to solve or even how to approach the problem. In this way, GP can be applied to problem domains where humans have difficulty formulating solutions.

One domain that poses difficulty for humans is distributive intelligence. Problems that require a direct, serialized solution are more approachable by human problem-solving techniques, whereas distributed and parallel tasks require the human to think not as an individual agent but as a group. Multi-agent problem solving that results in emergent behavior poses a significant intellectual hurdle for human minds, since examination of the each agent may not be enough to explain the behavior of all the agents as a group. While this type of problem solving is difficult for humans, there is ample evidence that it is leveraged to great effect throughout nature. The immune system, for example, accomplishes a very complex task, that of protecting the body from outside infection, using an intricate system of agents acting using relatively local information. Similarly, ant colonies have been shown to exhibit complex hive behavior, yet they lack central oversight, and thus must accomplish this behavior through a distributed system of multiple agents working with local information.

The ant example has been studied using genetic programming (Koza, 1992), and the results have shown that emergent behavior can arise from these distributed systems. In some cases, examination of the individual rules followed by each agent reveals logical and intuitive decisions that can explain, in part or whole, the behavior of the group. However, it can happen that solutions may arise in which the local rules do not directly imply (by a human's reading at least) the resulting group behavior. It is in these instances that GP can be of particular use. Not only does it provide a means to attack problems that are awkward for human, single-agent problem solving, but the solutions they provide can potentially offer insight into how such emergent behavior comes about.

This paper explores the application of GP to the area of two-dimensional modular configuration. The problem of two- or three-dimensional configuration has been addressed before in other fields (such as robotics, see Casal, 1999) with some degree of success. In the traditional approach, the researcher designs the rules for the individual agents with goal of the agents exhibiting a certain predetermined behavior. The approach taken in this paper uses genetic programming to try to learn a good set of rules that, when given to all the agents in the world, result in a desired collective behavior.

2. Methods

This problem was implemented using ECJ 10, a program written by Sean Luke. A number of different conditions, parameters, and even function sets were used (although they differed in only small amounts) in this study, but the following description outlines the core configuration for the problem. Deviations from this core are noted where they are discussed in the Results section. The standard genetic programming tableau for this problem is given in Table 1.

Table 1 Tableau for Multi-Agent Two-Dimensional Configuration problem.

Objective:	Find a program that when each agent in a group follows that program the group can configure themselves to match and arbitrary pattern.
Terminal set:	Move, Left, Right
Function set:	IfCanMove, IfOnPattern, Progn2 (branch into 2 subtrees)
Fitness cases:	A number of worlds with different patterns
Raw fitness:	The number of agents standing on `PATTERN` squares at the last time-step (across all worlds)
Standardized fitness:	The number of remaining unoccupied `PATTERN` squares at the last time-step (across all worlds)
Hits:	Same as raw fitness
Wrapper:	None
Parameters:	Population size between 50 and 200 Generations between 50 and 100
Success predicate:	An s-expression that results in all `PATTERN` squares being occupied

A number of design choices were available for configuring the problem beyond the standard parameters involved in genetic programming. During the course of this research, several options were explored, however the core design choices can be summarized in the list to follow. Deviations from this core are detailed in the sections describing each aspect of the problem.

- Each individual in the genetic programming population is cloned to make the population of agents for that individual's evaluation. In other words, the population of agents is homogeneous.
- The world allows wrapping, so individuals can move from one edge to the opposite, i.e. there are no walls.
- Multi-objective fitness measurement is used to prevent solving the problem of over-fitting to one objective.
- Only the most minimum of information was provided the agents, so there is no communication between agents, and agents can see no further than the square they are on (although they are prevented from moving into another already occupied square).

2.1. The World

The world is defined as an m by n two-dimensional array, containing two types of values: `EMPTY` and `PATTERN`. The agents are placed randomly throughout this world, with just enough agents to equal the number of squares with `PATTERN`. The goal is for the agents to arrange themselves so as to cover the pattern completely. Figure 1 displays some example worlds used in some of the simulations. '.'s indicate empty squares, while '#'s indicate pattern squares. The first world, 'square', is of particular note.

Given the restriction that every agent must be the same, and that no communication is permitted among agents, an immediate possible solution to this problem would be to design an agent that does some sort of efficient search of the world, and stops when it finds a marked square. While this does get descent results, the square world shows where it fails. With this pattern, the proposed solution would only fill in the perimeter of the square. After that, since there is no communication allowing one agent to tell another to "move over", the inner parts of the square will be completely blocked, and will only be occupied if an agent randomly started there.

As noted previously, the world wraps, allowing passage of an agent out one side and into the other. This option was not included in the original experiments, but the poor fitness of any movement (since doing just about anything other than standing still moved the entire population to one wall) caused most runs to converge with "do nothing" as the optimal individual, since that at least allowed some agents to stay on patterns that they were randomly assigned to in the first place.

2.2. Functions and Terminals

The choice of functions and terminals was designed to be minimal. The goal was to give the agents as little help as possible, since doing so made the problem harder for a human to solve (give a human complete communication and a universal controller and the problem becomes almost trivial). The three terminals are nearly the bare minimum (either the

```
           1         2
  12345678901234567890
 1....................
 2....................
 3....................
 4....................
 5....................
 6....................
 7....................
 8.......######.......
 9.......######.......
10.......######.......
 1.......######.......
 2.......######.......
 3.......######.......
 4....................
 5....................
 6....................
 7....................
 8....................
 9....................
20....................
```

```
           1         2
  1234567890123456789012345
 1.........................
 2.........................
 3...###...................
 4......#............###...
 5.....#............#......
 6....#............#.......
 7.....#..........#........
 8.....###.....####........
 9........#...#............
10........##..#............
 1.....###..##.............
 2..###.....#..............
 3...........#.....#.......
 4...........#....#.#......
 5........####...##..#.....
 6.......#....#..##..#.....
 7......#.....#.#....#.....
 8.....#.......##.###......
 9......#..................
20.....#...................
 1................##.......
 2.........#......##.......
 3...............##.#......
 4.................#.......
 5.........................
```

```
           1         2
  12345678901234567890
 1....................
 2....................
 3....................
 4....................
 5....................
 6........####........
 7.......######.......
 8......##....##......
 9.....##......##.....
10.....##......##.....
 1.....##......##.....
 2.....##......##.....
 3......##....##......
 4.......######.......
 5........####........
 6....................
 7....................
 8....................
 9....................
20....................
```

Figure 1 Three example worlds used for some of the GP runs: square, tentacles, and donut, respectively.

`Left` or `Right` could be removed without limiting functionality). `IfOnPattern` tells the agent if it is standing on a square with a pattern and is necessary as a manner for the agent to interect with the world to gain the pattern information. `IfCanMove` is possibly the least necessary, because executing a `Move` into a blocked square simply does not work, with no other consequence. However, considering the limit in the number of time-steps each group has to find the pattern, where a time-step is measured as allowing each agent to make one Move or complete one tree, whichever happens first, it was reasoned that the added control could make the evolved program more time efficient, although no significant studies have yet been conducted to quantify the hypothesized heightened efficiency.

`Progn2` is a branching node, serving no purpose other than to allow a program to spawn two child trees at this node. Inclusion of `Progn2` showed a marked improvement over earlier runs without the function (see the Example and Results sections below). More detail on the importance of `Progn2` follows in The Agents section.

2.3. The Agents

The agents each represent a cloned copy of the individual evolving in the genetic program. Agents are not themselves part of the evolution structure; they are transient and only serve as a means of evaluating the fitness of a give individual from the genetic programming population. When a program needs evaluation, it copies itself into a population of agents and then the agents are set work on a set of worlds, trying to match the patterns in each world using the same program.

The agents simply run through a loop, executing the program (an s-expression) they have been given continually until some maximum number of time-steps. For each time-step, each agent is given the opportunity to execute a number of nodes in its program tree until either it finishes the tree (and would begin back at the root) or until it executes a `Move` node once (whether the move is successful or not). `Move` is a terminal, so without `Progn2`, an agent would simply execute one pass of the program tree for each time-step. This means that at every time-step, the agent would have no memory from the previous time-step, making any sort of advanced searching behavior impossible. Progn2 provides an agent with a form of memory, since it allows the execution of leaf nodes without requiring that the agent have completed execution of the entire tree. This way, and agent can start a new time-step somewhere in the middle of eats program tree, thus allowing it to have different states, effectively. This allows for much more complicated behavior.

In order to accommodate agents stopping execution midway through the recursion of a tree and to resume in that same state later, the agents were implemented using multithreading, giving each agent its own thread. This allowed the thread stack to act as indirect memory for the agents.

2.4. Example

This example is drawn from early results that did not include `Progn2`, thus the behavior is relatively simple. This also shows how important the inclusion of `Progn2` is when these results are compared to those in section 3.

The evaluation worlds are donut and square, shown again in Figure 2, only this time they also depict the population of agents distributed randomly about each world. ‘o’ represents an agent on an empty square, while ‘H’ represents and agent situated atop a pattern square. As the program evolves, it gets better and better results, but it plateaus at about 37. Figure 3

```
o..ooo.....o........      ........o..o....o.oo
.o....o.o...........      o.......o...........
.................o.o      .....o..............
...................o      ....................
.o.....oo...........      ........o....o......
....o...H###.......o      .................oo.
.......######.....o.      ....o......o........
..o...##....##ooo..o      .......####H#.......
o....##......##.....      ..o....######o......
.....#H...o..##.o...      .......####H#....o..
.....##......##.....      ...o...######.......
.o...##.....o##...o.      .......######.......
......##.o..#H..o...      .......##H###.......
.o.....######.......      .......o.o........o.
...o....H###........      .....o...........oo.
....................      .......o...o....o...
....................      ................o...
..................o.      ...........o...o....
..o......o.o........      .........o..o...o...
o............ooo....      ....................
```

Figure 2 Donut and square populated with agents at random starting positions. ‘o’ represents agnets, ‘#’ pattern squares, ‘.’ empty squares, and ‘H’ pattern squares with agents.

shows one high performing individual, for which the number of hits was 38. The program for this individual is shown below.

```
(ifCanMove (ifOnPattern right move) right)
```

This simple program executes the strategy described earlier. If the agent is on a pattern square, then it simply turns right continually. If it is not on a pattern square, it moves forward, unless it cannot do so, in which case it turns right, giving it a chance to try moving a new direction on its next turn. Predicted pattern of filling the perimeters of patterns is apparent in Figure 3.

```
....o....o..........      ..o.................
....o............o.o      ..............o.....
........o.......oo..      ....................
.o.....o..........o.      .............o......
...o.........o......      .o..............o...
........H#HH.o......      .o..o...o..........o
.......H#H##H.......      ....................
......H#....#H......      .....o.HHH##H.......
.....H#......#H.....      .......#####H.......
.....HH......#H.....      .......##H##Ho......
.....H#......#H.....      .......H####H.......
.....H#......#H.....      ....o..H####H.......
......H#....##......      .......H###HH.......
o......H####H.......      ............o.......
........HHHH....o.o.      ................o...
....................      o...........o.......
.................oo.      ....o..........oo...
....................      ..................o.
....................      ....................
............o.......      .o...........o......
```

Figure 3 After evolution, a simple program has evolved that searches until a pattern square is found and then stops, resulting the in the perimeter-filling phenomenon seen here.

```
....................     o...................
....................     ....................
....................     ....................
....................     ....................
....................     ....................
........HHHH........     ....................
.......HHHHHH.......     ............o.......
..o..oHH....HH......     .......HHHHHH.......
.....HH......#H.....     .......HH#HHH.......
.....HH......HH.....     .......#HHHHH.......
.....HH......HH.....     .......HHH#H#.......
.....HH......HH.....     .......HHHHHH.......
......HH....#H......     .......HHHHHH.......
.......HHHHH#.......     ..................o.
........HHHH........     ....................
....................     ....................
....................     ....................
....................     ....................
.o..................     ....................
....................     ..o.................
```

Figure 4 The inclusion of `Progn2` allows for a much more complete solution, hitting 73 of the 80 targets.

3. Results and Discussion

Some of the results have been included in earlier sections, so they will only be briefly summarized here. Section 2.4 details how running the genetic program on test worlds with enclosed patters (like square and donut) without allowing the branching node `Progn2` results in relatively simple programs that cause agents to search until they find a pattern square and then stop. To compare this performance with what can be achieved using a branch node, examine Figure 4, which depicts a similar run as Example only with `Progn2` included in the function set. After only 30 generations with 50 individuals, Figure 4 depicts a much more complete program that far outperforms the results from the original experiments without `Progn2`. The number of hits for this individual is 73 out of 80, compared to the 38 from the previous section. The program that produced this output is quite long, and much more difficult to interpret than the first example.

```
(progn2 (ifOnPattern (ifOnPattern right (ifOnPattern
     left move)) (progn2 (ifOnPattern (ifOnPattern
     (ifOnPattern right (ifCanMove (ifCanMove
         move (ifOnPattern left move)) (ifOnPattern
         (ifCanMove (ifCanMove move right) (ifOnPattern
             (ifCanMove (ifCanMove move right) (ifOnPattern
                 left move)) (progn2 (ifOnPattern left move)
             (progn2 move left)))) (progn2 move (progn2
         move left))))) (progn2 (ifOnPattern (ifCanMove
     move right) move) (ifCanMove right move)))
     move) (ifCanMove right move))) (ifOnPattern
     (ifCanMove (ifCanMove move right) (ifOnPattern
         (ifOnPattern (ifOnPattern (ifCanMove (ifCanMove
             move right) (ifOnPattern (ifOnPattern (ifCanMove
             (ifCanMove (ifOnPattern left move) (ifOnPattern
                 left move)) (ifOnPattern left move)) (progn2
             (ifCanMove (ifCanMove move right) left) (progn2
             move left))) move)) (progn2 (ifOnPattern
             left move) (ifOnPattern (ifCanMove (ifOnPattern
             right (ifOnPattern left move)) (ifOnPattern
             left move)) (progn2 (ifOnPattern left move)
             (progn2 move left))))) (progn2 (progn2 move
             left) (progn2 move left))) (ifCanMove right
         move))) (progn2 (ifOnPattern left move) (progn2
     move left))))
```

While the exact strategy this program employs may not be readily apparent from this expression, the frequency of the use of Progn2 is. Each occurrence represents an additional state that the agent can be in between time steps, and since these states can have different meaning depending on whether the agent can move, is on a pattern square or empty square, etc., the use of so many branch nodes represents a large combinatoric increase in the memory of the agent, which is the source of the more complex behavior.

Perfect solutions were not achieved on multi-objective runs. Often, combining multiple worlds of differing types (like those in Figure 1) would result in solutions converging to solve only a subset of the worlds. When applied to a single world, near perfect solutions (within 1 hit) are possible (Figure 5). This suggests that better results may be achieved by using layered learning techniques on each type of world and allowing a master program to use the learned programs as subroutines in solving the global problem. However, such fitting could be the result of the GP learning a better program, or over-fitting, although over-fitting is hampered by the homogeneity constraint and the very multi-agent nature of the problem. Unfortunately, programs like the one depicted above (and the much longer ones learned for larger worlds—data not shown) are not easy to interpret.

```
....................
....................
....................
....................
....................
....................
......o.............
.......HHHHH#.......
.......HHHHHH.......
.......HHHHHH.......
.......HHHHHH.......
.......HHHHHH.......
.......HHHHHH.......
....................
....................
....................
....................
....................
....................
....................
```

```
.........................
........................o
...HHH...o...............
......H............HHH...
.....H............H......
....#............H.....o.
.....H..........H........
.....#HH.....HHHH........
........H...H..........o.
.o......#H..H............
.....HHH..HH.............
..HHH.....H..............
...........H.....H.......
...........H....H.H......
........HHHH...HH..H....o
.......H....H..HH..H.....
......#.....H.H....H.....
.....#.......HH.#HH......
......H..................
.....H...................
................HH.......
.........H......HH.......
...............HH.H......
.................H.......
.........................
```

Figure 5 Best-of-run result from a run on only the square world (left), 35 of 36 possible hits, and a run on the square, donut, and tentacle world (only tentacle depicted), right.

4. Conclusions

Genetic programming can be successfully applied to the problem of multi-agent two-dimensional configuration, although the full measure of the success cannot be ascertained without either better interpretation and understanding of the resulting programs or more experimentation to empirically determine that these results are not over-fitting the given configuration worlds.

While not conclusive in and of itself, the complexity of the solutions that inhibits there interpretation implies that the genetic programming algorithm succeeded in learning a complicated yet successful strategy to solve a problem that a human could not solve as well given the same constraints. This conclusion also hinges on the issue of over-fitting, so it must remain tentative until further work can be done.

These results demonstrate the necessity for some means of maintaining state for each agent. Without it, the complex behavior would not evolve, and with it the problem became solvable. There are many other ways to store and represent state, and as with the many other design decisions made in this work, it is likely that a different representation could yield very different results, perhaps even solve some of the clarity problems that are so abundant with using branching. Regardless, this shows that the problem of multi-agent two-dimensional configuration is approachable and perhaps solvable through genetic programming, and it warrants further research and investigation.

5. Further Work

While the results detailed here are promising, there are (as always) still many questions left unanswered. It was possible to get good, converging solutions, but not always possible to understand them. Unless research into this problem is to be relegated to a purely empirical approach, then better methods for interpreting the resulting programs are necessary. A parser that can simplify redundant branches within the tree, or perhaps software for better visualization of each agent evolving and acting in comparison with other, characterized agents.

The results generated here used a very stringent set of constraints. This leaves plenty of room for relaxing the constraints in order to help solve the problem. While the solution under the strict constraints may be of theoretical interest, it does not imply that utilizing other functions such as communication and perhaps sight could not yield other, equally theoretically interesting results. Further work will attempt to achieve better performance, and perhaps more importantly, qualitatively different performance, by adding further means for the agents to interact with the world.

Acknowledgments

I would like to thank John Koza for his consultation in all things GP, and Dave DeConde for technical assistance.

Bibliography

Casal, A., Yim, M. "Self-Reconfiguration Planning for a Class of Modular Robots", SPIE symposium on Intelligent Systems and Advanced Manufacturing, Sept. 1999.

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