The Three-Headed Monster: An Evolutionary Program

Charles Guse and **Dana Vrajitoru**
Computer and Information Sciences Department
Indiana University South Bend
South Bend, IN 46617

Abstract

In this paper we present our experiments with developing an intelligent agent using genetic programming, with the goal of exploring an environment with limited information in search for resources. The challenge consisted of evolving an agent browsing a world knowing only the altitude in a local neighborhood, looking for a balance food diet of game and grain. We used genetic programming to evolve our agent, using both a simple exploratory approach and more sophisticated search techniques. Our results show that in this particular context, the simple approaches seem to work the best.

Introduction

In this paper we present some experiments with evolving an agent capable of exploring an environment given as a height map in search for food, with the purpose of achieving a balanced diet. The agent was developed for the GECCO 2008 Contest: Finding a Balanced Diet in Fractal World (Keijzer 2008). The challenge of this particular application is that the agent has no prior knowledge of the location of the food and can only 'see' a limited local neighborhood around its current position. We decided to evolve this agent using only genetic programming and virtually no static memory that the agent can use.

The problem we are discussing in this paper is specific to the GECCO 2008 contest, but it fits within the larger context of developing intelligent agents capable of finding resources in their environment using limited information. These agents are necessary for the future of artificial life. There is a good number of potential applications, going from robots exploring new worlds, to autonomous devices providing common domestic services while being able to refill the vital resources with little human intervention.

The environment consisted of a discrete two dimensional world containing food of two types: grain and game. The world is specified as a height map, and the agent only knows the altitude in a small neighborhood around its location. The agent will not know if a cell contains food until it moves to it. There is an implicit probabilistic correlation between the height of the terrain and the sort of food that can be found there. The test would drop the agent in a random position on the map and the agent was allowed a given number of moves to find food. The score was the amount of both grain and game that it could find within those steps, meaning the value of the smaller of the two quantities.

We chose genetic programming as our basic model because it seems the most appropriate for the type of problem we tried to solve. The specifics of the problem definition were limiting in the choice of algorithms. For example, usual search algorithms like the A* or depth-first rely on complete knowledge of the map, which was not part of our constraints. The knowledge was limited to the small neighborhood and more knowledge was not available until a move was made. Greedy algorithms would also not be an option because the agent had no knowledge of the food position until it made a move to the cell. Genetic programming gave us the flexibility of not having to pre-define a model for the agent's behavior, as we would have to do with genetic algorithms or neural networks.

We developed the agent in three phases. In the first phase, we aimed to evolve a function exploring as wide a region as it could without repetition and without memory of where it has been before. In the second phase, we evolved the agent in the real contest conditions, and using the best function evolved during the first phase. In the third phase we introduced a cooperative coevolutionary approach where three populations of functions were evolved at the same time in parallel, serving as a test, a positive action, and a negative action in a conditional.

Overall our best agent managed to eat an average of 8.77% of the expected food on the map in a fairly balanced diet, and more sophisticated approaches didn't seem to yield the expected improvement in performance.

Genetic programming (GP) (Koza 1992) is a popular evolutionary paradigm that is used in many cases to evolve complex behavior (Altenberg 1994). Many studies have applied it to developing agents capable of exploring a given terrain with reduced information or even in hostile environment (Haynes and Wainwright 1995). A related topic is evolving robot behavior (Tanev, Ray, and Buller 2005), (Andersson et al. 1999), (Lee and Zhang 2000) or articulated organisms capable of moving efficiently in a given environment (L. Gritz 2006), (Shim and Kim 2003).

The paper is structured as follows. We start by describing the problem and test conditions. Next, we introduce our evolutionary models and the solutions we retained. Third, we compare the scores of the agents in the real test conditions.

The Problem and Contest Rules

The goal of the competition was to evolve an agent to search a two dimensional landscape and find as much as possible of two types of food. The landscape consisted of a 256x256 grid. Each cell had an 'elevation' between 0 and 255. Figure 1 shows an example of a region taken from such a map, where the cells are shaded by elevation, brighter levels of gray meaning lower values. Two types of food are in this landscape: grain, marked by white dots, and game, marked by black dots. In addition, large dark blots represent impassible terrain. We were provided 10 such sample maps, and we assumed that for the contest the agent would be tested on a new set of maps.

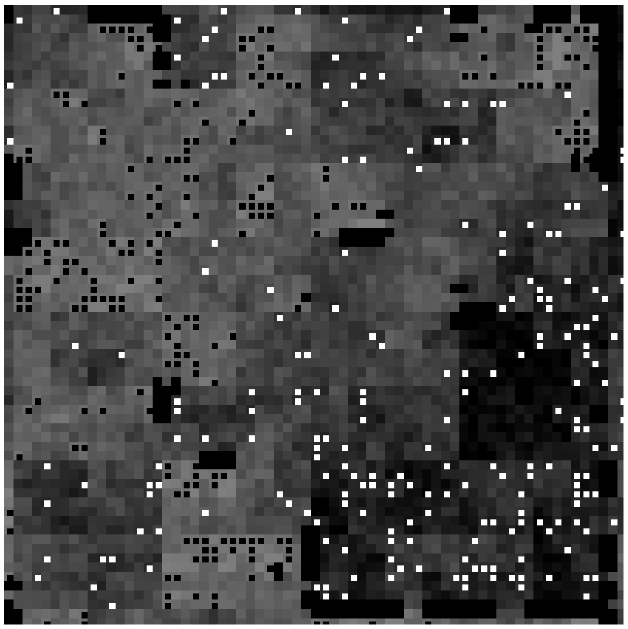

Figure 1: Sample area taken from a map, brightness defines elevation, white dots represent grain, black dots represent game, large dark areas represent impassable terrain.

The agent's goal was to find a balanced mixture of grain and game. Specifically, for the competition an individual's fitness was evaluated as the amount of whichever food it found less of. For example, if an agent found 35 grain and 29 game, its score was 29. Individuals were allowed 13,107 moves, representing 20% of the surface of the terrain. Thus, exhaustive search algorithms were not expected to be particularly effective.

The agent had information only about the elevation of the terrain in a vicinity of the current position. There was an implicit probabilistic correlation though between the type of food more likely to be present at a particular location. The average elevation and the standard deviation on the provided maps are shown in Table 1 below. From this table we can see that game can be found mostly at low altitude elevations, while grain can be found mostly at high altitude elevations.

Table 1: Distribution of game and grain based on elevation

	Game		Grain	
Map	Ave	StdDev	Ave	StdDev
1	79.81	28.3	223.92	10.49
2	75.41	27.48	220.88	17.75
3	77.24	25.98	221.17	17.61
4	73.37	26.83	221.17	18.29
5	74.56	25.93	198.48	46.68
6	72.42	26.71	223.26	15.64
7	73.32	25.97	176.27	62.39
8	66.8	26.98	177.66	65.35
9	74.17	25.84	222.76	15.29
10	69.83	26.36	221.54	22.99
Average	73.69	26.64	210.71	29.25

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Figure 2: Visible neighborhood of the agent and its cell numbering. The current cell is represented by the number 12.

Individuals could 'see' elevations in a 5 by 5 grid around their location, numbered as shown in Figure 2. They automatically picked up food if they entered a square containing it, but could not see it. Thus, the agents had to learn where to look for food based on elevation. At each move, the current position of the agent is in cell 12, in the middle. The agents started on a random cell of unknown location to them.

The ultimate goal of the contest was to develop an ANSI C function with the prototype
```
int Next_Move(int *grid, int grain, int
game, int last, int time);
```
returning the next relative position of the agent as a number between 0 and 24, as stated in Figure 2. The agent received the local neighborhood for the next move as input for this function, as well as the last move it had executed before. Moves resulting in an impassible position, remaining in place, or attempting to move further away than the 5x5 grid resulted in the agent's position not changing, but still counted towards their total number of moves.

Our Solution and Implementation Details

We developed our agent in several phases based on the functionality we aimed to accomplish at each stage. In addition, we wrote some test functions designed to represent a baseline for comparison of the performance of the evolved agents.

Our initial plan was to divide the process in two phases: for the first half of the given steps we would develop a greedy algorithm trying to get as much food of any sort as it could. In the second half we would employ another algorithm attempting to improve on whichever food had been collected less so far.

For the implementation of the agents, we focused on genetic programming (GP) (Koza 1992) attempting to allow for as little input from the human as we could. We used a small free Genetic Program written in Python by Paras Chopra (Chopra 2008) and modified for our purposes. We converted the resulting trees to C by hand.

Benchmark Functions

We used two functions to give us an idea of how well our agent was performing.

The first one is a completely random exploration of the terrain, with no consideration for the elevation or for the current performance in terms of acquired food. This function chooses a random position within the 5x5 grid, taking care of not staying in place (position number 12) and of not falling on a block (marked in the elevation map by -1). This first function gives us an idea of the probability of finding food on the map with a completely random exploration. We called this function `random_walk`.

The second benchmark function is a small utility that we wrote by hand, considering the amount of game and grain that the agent acquired so far, and attempting to improve on the least of the two numbers. For that, the agent is looking at the elevation of the terrain in the 5x5 grid around the current position and moving towards cells of elevation most likely to match the average height of the kind of food it needs. For example, if the amount of grain is lower than the amount of grain, the agent tries to move toward cells of lower elevation, ideally around the value 73.69, which is the overall average of the game positions. This second function would give us an idea of how the agents evolved by genetic programming compare with one written by a human. We called this function `human_walk`.

First Phase

For the first phase we started with a function evolved with a classic GP with the aim of exploring a square area trying to avoid walking over the same cells more than once. The chromosomes are trees where nodes can be functions, variables, or constants. Possible nodes are described below, where the functions have the usual meaning in most programming languages, wrapped in some tests for mathematical soundness of the arguments:

- *Functions:* +, -, *, /, %, cos, sqrt, pow
- *Variables:* x and y
- *Constants:* the range of possible moves, [0, 24], plus the size of the table and some randomly generated constants
- *Output:* a value capped to the range [0, 24] representing the next move.
- *Other parameters:* population size: 100, number of generations: 500, maximum tree depth: 30, selection: fitness-proportionate and elitist, probability of crossover: 0.8, probability of mutation: 0.1

The first four functions we chose as potential nodes for our trees define the basic arithmetic operations. The function square root was chosen because the size of the known neighborhood of the agent was of the order of magnitude of the square root of the size of the table. The cosine function if often used in genetic programming and in combination with the function power it can be used to derive the whole set of trigonometric functions.

These other parameters were used through all the experiments, unless otherwise specified. We used the classic GP crossover and mutation. We initially tried a larger number of generations but the population seemed to converge rather quickly and it wasn't of much use letting it evolve a lot longer.

Fitness: we started with a table of 25x25 which was initially composed of unmarked cells. We tested each program by making 150 moves based on their output, updating the variables x and y after each move accordingly, and marking the visited cell each time. The fitness consisted of the number of marked cells at the end of the run.

Figure 3 shows an example of the output from the program, corresponding to the tree shown in Figure 4.

```
+--ADD
|   +--SQRT
|   |   +--ADD
|   |   |   +--[X]
|   |   |   +--SQRT
|   |   |   |   +--ADD
|   |   |   |   |   +--[Y]
|   |   |   |   |   +--[9]
|   +--SQRT
|   |   +--DIV
|   |   |   +--[X]
|   |   |   +--[20]
```

Figure 3: Example of output of the program

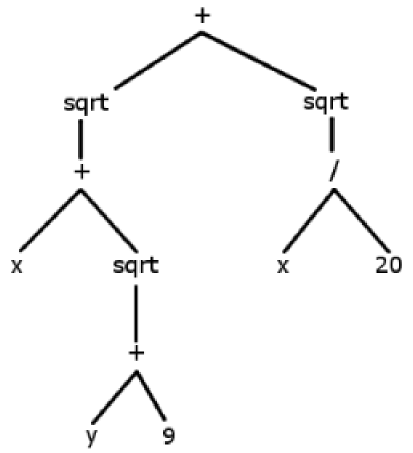

Figure 4: Tree corresponding to the function in Figure 3

The best program we obtained achieved a fitness of 86 and was produced at generation number 162. We translated this output into a C function called `walk_move` (the full code is available in (Vrajitoru and Guse 2008)). The C code for this

```
int walk_move(int x, int y)
{
  float a, b, c;
  c = pow(y - (y+x%y)/y, sqrt(8));
  a = sqrt(15 + y + x%c);
  b = cos((x*20)%y)/11.0;
  return x - (a + b);
}
```

Figure 5: The function `walk_move`

function is shown in Figure 5. Interestingly enough, many of the functions we obtained this way seem to want to go constantly in one given direction.

We tested the function `walk_move` by itself in the real test conditions. As we didn't know the real position of the agent on the map, we started with random values for the variables x and y for which we used static variables, and updated these values based on the moves made. We treated the map as being circular, which means that 256 = 0 for both coordinates. This test revealed that the function has an eigenvalue and the agent eventually converges to that position on the map sooner or later. A solution to avoid this convergence is to re-scramble the variables x and y from time to time.

Second Phase

For the second phase we introduced the real test conditions into the fitness function. The size of the table for the fitness was 256x256 and the food and elevation tables were imported from one of the files that were provided. The fitness function ran the agent 150 steps on one of the real tables and the game and grain points were counted. The fitness returned the least of game and grain. Even though we only used one specific map in the fitness itself, after the training was done, we tested all the functions on all of the maps and made the selection this way. Thus, the function we submitted was not over-specialized to the training data.

The set of functions used as potential nodes in the trees was extended to include the function `walk_move` evolved in the first phase, and also a function `elev` with one argument that returns the elevation at that position in the 5x5 grid if the argument is in the range [0,24], and -1 otherwise.

We evolved two agents with this new approach. For the first one we added the following set of logical and comparison functions: {IF, EQ, NOT, GT, LT, AND, OR, XOR}. The best function evolved this way is translated into the function `game_walk`. As the logical and comparison functions didn't seem to be well used, we removed them, and the best function evolved this way translated into the function `walk_elev`. The original trees for both these functions can be found in (Vrajitoru and Guse 2008). Neither of these two functions performed any better than just the function `walk_move` by itself, so we decided not to use them. In fact they both seem to return a constant move most of the time.

Coevolutionary Approach

For the third phase we tried a cooperative coevolutionary approach (Paredis 1996) where we evolved three populations in parallel. This is a cooperative approach (Potter and De Jong 2000) where different related components are evolved by separate populations (Wiegand, Liles, and Jong 2001). The first population evolved a condition using the set of logical and comparison functions introduced above. The second population evolved a function to be called if the condition is true. This population was not using the logical and comparison functions, but the set of arithmetic functions. We decided at this point to replace the function `elev` with just a variable containing the elevation at the current position in the grid. The third population evolved a function to be called if the condition is false, and its parameters were identical to those used for the second population.

The fitness function was very similar to the one described for the second phase, except that it used one chromosome from each of the 3 populations together to determine the move. The method for combining the 3 chromosomes was the following: for each generation, the chromosomes of the first population were evaluated using the best chromosome from the second and third populations from the previous generation. The second and third populations were evaluated in a similar way.

The best combined functions that we obtained with this new approach were called `three_headed` and the full code can be found in (Vrajitoru and Guse 2008). The performance of these functions in the real test conditions was quite inferior to the first function that was evolved in phase one.

Final Decision

Given that the more complex functions that we tried to evolve performed below our expectations, we decided to turn in the very first function `walk_move` that was evolved without any knowledge of the elevation and of the accumulated food.

As a wrapper around this function, we used a couple of static variables x and y that we initialize with random values in the range [0, 255]. We updated these variables based on the current move and we enclose it in a loop that makes sure that the move returned is not 12 (stay in place) or a move to a position containing a block. On the average we estimated that the algorithm collects a total amount of food between 5% and 10%, which is relatively well balanced between grain and game.

Numerical Results and Analysis

All of the agents evolved by genetic programming were subjected to a fitness function testing their ability to solve the problem in somewhat limited conditions, as described in the previous section.

For a better reflection of the agents' performance and a unified comparison, we subjected all of them to a unified test set. We ran the agents on each of the 10 sample maps for the number of steps specified in the contest, which is 13107. We repeated the test 1000 times on each map starting

Table 2: Average scores over all the maps in 1000 tests and 13107 steps

Agent	Average	Best
walk_move	28.74 (8.77%)	42.77 (13.06%)
game_walk	0.8 (0.24%)	0.96 (0.29%)
walk_elev	0.89 (0.27%)	1.08 (0.33%)
three_headed	2.79 (0.85%)	2.27 (0.69%)
random_walk	30.68 (9.37%)	46.93 (14.33%)
human_walk	3.99 (1.22%)	3.28 (1.00%)

from a different but random initial position, unknown to the agents themselves. We ran the same test with the benchmark functions. The performance in all the cases was considered to be the lowest number of the acquired game and grain.

Table 2 shows the results of these experiments as follows. The total amount of game or grain in all the maps was 3276. The number of moves allowed the agent to explore at most 20% of the entire map. This led to an expected maximal value for the amount of food of 327.6, because for the agent to achieve a given score, it had to find that number of both game and grain. We present the scores as a simple count, and then as a percentage of the maximal expected value in parenthesis. The second column represents the average over all the maps, while the third column represents the map number 1 on which most agents obtained the best score. We can speculate that even though all the maps contain the exact same amount of food, this map probably has the best distribution of the food, making it easier to find.

The results in Table 2 indicate that the best performance was achieved by the function `random_walk`. Note that the function "random" itself was not part of the set that the evolutionary process could use to construct trees. The second best function is `walk_move`, which is the result of the simplest evolutionary approach that we took. This function visibly outperforms all of the others resulting from evolution, and the one designed by hand, `human_walk`. The function `three_headed`, resulting from the most sophisticated evolutionary process we used, improves on the performance of the previous ones and is comparable to the human designed function.

To show how the different distribution of food can influence the performance of the agents, Table 3 presents the results on the maps where the function `random_walk` achieved the weakest score, map number 10, and the one where the function `walk_move` got the lowest score of all the maps, map 2. This shows that even though one agent might perform better over all, various distributions of the food on the map have a major influence on the performance of the agents. On the whole, the function `walk_move` showed the best score on 3 out of the 10 maps, and the function `random_walk` showed the best score on the 7 other maps.

Conclusions

In this paper we presented an evolutionary approach to developing an agent capable of exploring a two dimensional world in search for food in the specific conditions for the GECCO 2008 contest.

Table 3: Worst scores for the function `random_walk` (map 10) and for the function `walk_move` (map 2) in 1000 iterations

Agent	Map 10	Map 2
walk_move	24.64 (7.52%)	13.91 (4.25%)
game_walk	0.6 (0.18%)	0.38 (0.12%)
walk_elev	0.68 (0.21%)	0.7 (0.21%)
three_headed	2.37 (0.72%)	2.04 (0.62%)
random_walk	16.99 (5.19%)	20.53 (6.27%)
human_walk	2.62 (0.80%)	2.52 (0.77%)

We introduced several models, the first one doing a simple exploration where the goal was to avoid repetition as much as possible, with no knowledge of the specific problem conditions. The second model we developed evolved under the real test conditions, where the fitness function used in the evolutionary process reflected the performance measure used in the contest. The third model introduced a coevolutionary approach where three populations of genetic trees evolved in parallel, depending on each other for the fitness.

The results from the various tests indicate that the approach that performed best was the simplest one, not taking into consideration any information concerning the elevation and the food acquired so far. This agent was capable of acquiring about 4.25% to 13.06% of the expected food in a number of moves covering at most 20% of the terrain, which is an encouraging result. This last agent performs well as compared to the agents written by hand.

Our results indicate that the genetic programming can be a valid approach to difficult problems and can show performance matching or better than the human coded agents. They also suggest that sometimes attempts to introduce extra intelligence into the system can be counterproductive, and can limit the search space in unexpected ways.

References

Altenberg, L. 1994. The evolution of evolvability in genetic programming. In Kinnear, K., ed., *Advances in Genetic Programming*, 47–74. Cambridge, MA: MIT Press.

Andersson, B.; Svensson, P.; Nordin, P.; and M.Nordahl. 1999. *Genetic Programming*. Springer. chapter Reactive and Memory-Based Genetic Programming for Robot Control, 161–172.

Chopra, P. 2008. Genetic programming system in python. URL: http://www.paraschopra.com/sourcecode/GP/index.php.

Haynes, T., and Wainwright, R. 1995. A simulation of adaptive agents in a hostile environment. In *Proceedings of the ACM Symposium on Applied Computing*, 318 – 323. ACM Press.

Keijzer, M. 2008. Contest problems: Finding a balanced diet in a fractal world. URL:http://marvin.cs.uidaho.edu/ heckendo/Gecco08Contest/FractalFoodProblem.

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

L. Gritz, J. K. H. 2006. Genetic programming for articulated figure motion. *The Journal of Visualization and Computer Animation* 6(3):129 – 142.

Lee, K. J., and Zhang, B. T. 2000. Learning robot behaviors by evolving genetic programs. *Proceedings of the 26th International Conference on Industrial Electronics, Control and Instrumentation (IECON-2000)* 2867–2872.

Paredis, J. 1996. Coevolutionary computation. *Artificial Life* 2(4).

Potter, M., and De Jong, K. 2000. Cooperative coevolution: an architecture for evolving coadapted subcomponents. *Evolutionary computation* 8(1):1–29.

Shim, Y.-S., and Kim, C.-H. 2003. Generating flying creatures using body-brain co-evolution. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 276 – 285. San Diego, California: Eurographics Association.

Tanev, I.; Ray, T.; and Buller, A. 2005. Automated evolutionary design, robustness and adaptation of sidewinding locomotion of simulated snake-like robot. *IEEE Transactions on Robotics* 21(4):632–645.

Vrajitoru, D., and Guse, C. 2008. The three-headed monster: An evolutionary program. Technical Report TR-20081217-1, Indiana University South Bend, Computer and Information Sciences Department.

Wiegand, R.; Liles, W.; and Jong, K. D. 2001. An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In Belew, R., and Juillé, H., eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, 1235–1242.