Ant Programming: or how to use ants for automatic programming

Conference Paper · January 2000

CITATIONS
52

READS
775

2 authors:

Olivier Roux
Université Catholique de Louvain - UCLouvain
41 PUBLICATIONS **681** CITATIONS
SEE PROFILE

Cyril Fonlupt
Université du Littoral Côte d'Opale (ULCO)
105 PUBLICATIONS **1,112** CITATIONS
SEE PROFILE

Ant Programming:
Or how to use ants for automatic programming

Olivier ROUX, Cyril FONLUPT

Laboratoire d'Informatique du Littoral, Université du Littoral
BP 719, 62228 CALAIS Cedex, FRANCE
{roux,fonlupt}@lil.univ-littoral.fr

Abstract

Ant Programming (*AP*) is a new method which applies the principle of the ants systems to the automatic generation of programs. It is loosely inspired by the work of Koza [9] known as Genetic Programming (*GP*). *AP* is an evolutionary system which generates successively populations of solutions and uses the stigmergy-based communication to improve the solutions (programs). We compare here *AP* with some well known benchmarks of *GP*. The problems studied here are two regression problems and a multiplexer problem.

1 Introduction

For a few decades, natural evolution has been a source of inspiration for the resolution of combinatorial optimization problems. Among the most recent work have emerged the ants systems (AS) which make good solutions emerge from the memorization of partial information found during the search.

Genetic Programming (*GP*) has been introduced by Koza [9, 10, 11]. This method allows computers to solve problems without being explicitly programmed. *PIPE* [12] (Probabilistic Incremental Program Evolution) is a Population-Based Incremental Learning method, it is a variant of Genetic Programming. *PIPE* generates successive populations from a probabilistic prototype tree. This tree contains the probability of elements for all nodes. This probability is directly proportional to the fitness of the generated tree. This strategy leads to think that *AS* could be adapted with some expected success to the generation of programs. In the first part, we will briefly introduce the concepts used in genetic programming and ants systems. After these descriptions, we will present our algorithm, and some comparisons between AP and *GP*.

2 Overview

2.1 The ants systems (AS)

This class of algorithm has been proposed by Dorigo [2]. It is based on the modelization of the natural behavior of the ants. *AS* has been applied to combinatorial optimization problems like *TSP* [6, 8, 5], *QAP* [13, 7, 14], ...

One of the principle of the ants algorithms is the use of a collective memory to build and improve the solutions. With a very simple communication method (use of chemical substance known as pheromone) natural ants are able to solve some complex tasks: for instance finding the shortest way for avoiding an obstacle, or collecting food [1].

For one ant, the path is chosen accordingly to the quantity of pheromone. Furthermore, this chemical substance has a decreasing action over time, and the quantity left by one ant depends on the amount of food found. As illustrated in Fig. 1, when facing an obstacle, there is an equal probability for every ant to choose the left or right path. As the left trail is shorter and so requires less travel time, it will end up with a higher level of pheromone after many ants passing on, thus being more and more attractive. This fact will be increased by pheromone evaporation.

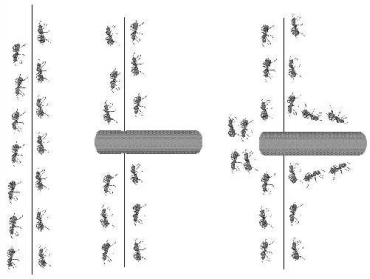

Figure 1: Ants facing an obstacle

DORIGO [4] quite naturally applied this principle to the resolution of the *TSP* (Traveling Salesman Problem). It consists in finding a Hamiltonian tour. The purpose of the pheromone trail is to encourage the ants to choose the edges connecting the cities. Each ant is dropped on a city and has to build tour. The rate of pheromone is proportional to the quality of the solution they build and to the length of the edge. This method has been quite succesfull in solving many difficult TSP instances.

Hybridisation of the ants systems with local search methods improves the performances of the *AS* for many problems [3] like *TSP*, *QAP*, ...

2.2 Genetic Programming (GP)

GP allows computers to solve problems without being explicitly programmed. *GP* automatically generates computer programs. It may sound odd to think that a computer may automatically generate computer programs to solve a given problem. Actually like in nature for evolution, *GP* will generate hundreds or thousands of different computer programs. A population (a set of programs) will evolve by repeatedly selecting the fitter programs, combining them and thus producing new programs.

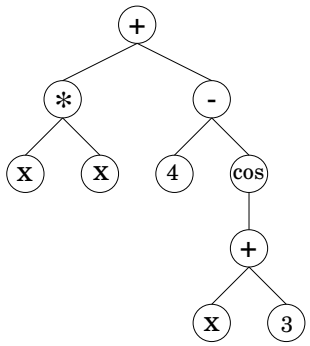

Figure 2: Tree representation of a program

As explained above, in *GP*, individuals in the population are computer programs. These programs are usually represented as trees. For instance, a mathematical expression such as $x^2 + (4 - \cos(x+3))$ is depicted in Fig. 2. Note that this tree form representation is equivalent to the parse tree that is built by most high-level language compilers. This tree form representation is also the most useful way of representing LISP expression (also known as S-expression).

The internal nodes represent the functions and the leaves are the terminal nodes: for example in Fig. 2 functions are the set $\{*, +, -, cos\}$ and terminals are $\{x, 4, 3\}$. The terminal nodes have an arity equal to zero (i.e. variables or numerical constants) and the non-terminal nodes are functions. They are chosen according to the type of the problem (for instance, a typical regression problem will use the functions set $+, -, *, /$ while a boolean problem would use the boolean functions AND, OR, NOT).

The general *GP* method (as introduced by Koza) can be roughly described as follows:

1. Randomly generation of programs (trees)
2. Evaluation of programs (fitness measure)
3. Apply genetic operations to individuals: crossover (Fig. 3), mutation (Fig. 4), copy
4. Selection of individuals based on the Darwinian principle of the survival of the fittest
5. Goto step 2 until some termination criterion is satisfied

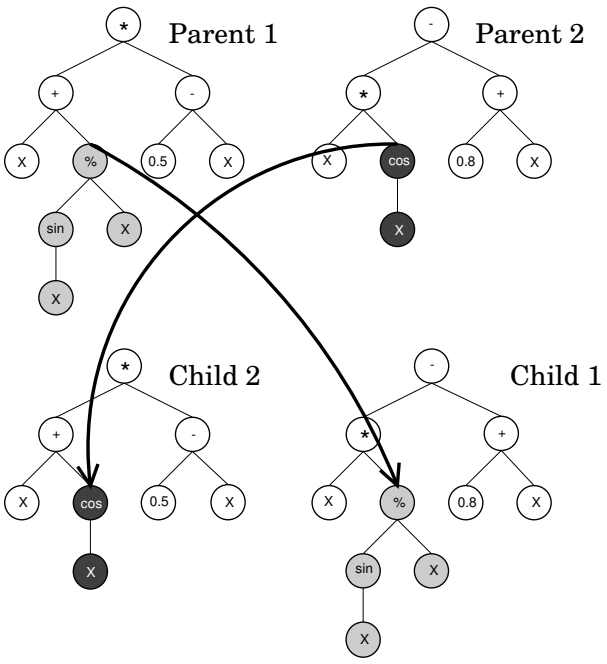

Figure 3: Crossover individuals exchange a part of their genetic materials by swapping a part of one parent with a part of the other parent.

The half-and-half generation method of initial population is frequently chosen *i.e.* half of the population is generated in the form of a complete tree of given size, while the other half is generated in a completely random way.

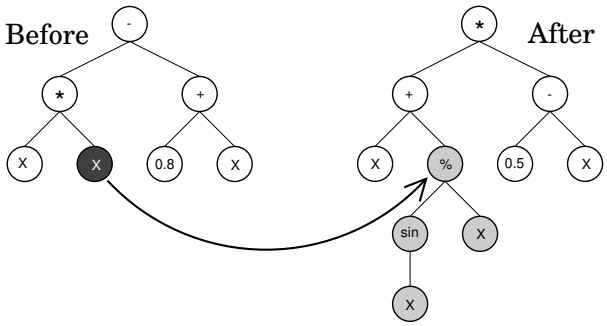

Figure 4: Mutation operates on one individual. A node of the tree is selected, and the associated subtree is replaced by a randomly generated subtree.

3 The *AP* algorithm

In the *AP* algorithm, each ant will build and modify the programs(trees) according to the quantity of pheromone at each node. The pheromone table appears as a tree. Each node owns a table which memorizes the rate of pheromone to various possible elements (Fig. 5).

First, randomly generated programs (trees) are created. The table of pheromone at each node is initialized at 0.5 (meaning that there is an even probability to choose each terminal and function). The higher the rate of pheromone, the higher the probability to be chosen. Each program (individual) is then evaluated. The table of pheromones is updated by two mechanisms:

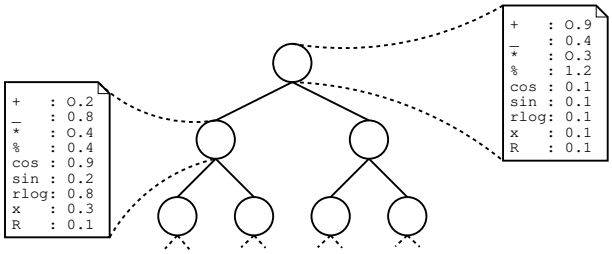

Figure 5: Pheromone tree: in each node a pheromone table holds the quantity of pheromone associated with all possible functions and terminals.

1. evaporation decreases the rate of pheromones for every element (T_g, pheromone value at the generation g) on every node. The formula is :

$$\tau_g = (1 - \alpha)\tau_{g-1}$$

with α constant ($\alpha = 0.1$).

2. for each tree, the components of the tree will be reinforced according to the fitness of the tree. The formula is :

$$\tau_{i,s_i} = \tau_{i,s_i} + \frac{\alpha}{f(s)}$$

s is a solution, $f(s)$ its fitness, s_i the function or the terminal set at node i in this individual, α is a constant ($\alpha = 0.1$), τ_{i,s_i} is the value of the pheromone for the element s_i in the node i.

Fig 6 briefly describes the *AP* algorithm.

```
0 Every component of the pheromone tree is set to an average value.

1 Random generation of program based on the pheromone tree
  (the higher rate, the higher the probability to be chosen).

2 Evaluation of ants.

3 Update of the pheromone table (see below).

4 Goto to step 1 unless some criteria is satisfied.
```

Figure 6: Overview of AP algorithm.

4 Results

The common parameters between the two methods (GP and AP) are identical (size of the solutions, method of generation). We use the number of evaluation as criterion for comparison between GP and AP. Results are given for 40 different runs. Fitness of the solution is given using the $adjusted\ fitness$, the formula is:

$$adjusted\ fitness = \frac{1}{1 + raw\ fitness}$$

where the $raw\ fitness$ depends on the problem.

Adjusted fitness ranges between $[0, 1]$. 1 indicates the best individual, 0 a very poor individual.

4.1 Symbolic regression problems

The function used for the first symbolic regression problem is $f(x) = x^4 + x^3 + x^2 + x$(Fig.7), 200 values in $[-1, 1]$ are selected for the fitness cases.

The fitness function is the sum of all errors between the values provided by the computed function (given by AP or GP) and the expected function:

$$raw\ fitness = \frac{1}{n} \sum_{i=1}^{n} |C_{computed} - C_{expected}|$$

The function set is $[+, -, \%, sin, cos, exp, rlog]$, where $\%$ is the protected division (provides 1 if the denominator is equal to zero), and the function $rlog$ is the protected logarithm function (gives the logarithm of the absolute value and 0 if the value is equal to zero). The terminals set is the variable x and R a random generic constant between -1 and 1.

The parameters are: 50 generations, population of 5000 individuals (25,000 fitness evaluations), the initial solutions are generated by the half-and-half method with a size ranging from 2 to 6, depth limit for execution is set to 6.

Results for this problem are shown in Tab. 1. For this problem AP gives worst results than GP. The GP finds the best results while AP only finds a solution of 0.166 ($adjusted\ fitness$).

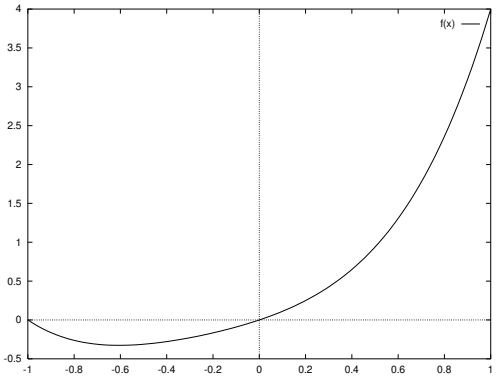

Figure 7: $f(x) = x^4 + x^3 + x^2 + x$

Table 1: Comparison between *GP* and *AP* for the symbolic regression problem corresponding to the function $f(x) = x^4 + x^3 + x^2 + x$

	GP		*AP*	
	nodes number	adjusted fitness	nodes number	adjusted fitness
Best	13	1.0	13	0.1660
Worst	14	0.1086	12	0.0458
Average	21.2	0.33405	13.125	0.0876

But when you look at the distribution of the 40 results, it is obvious that *GP* can find good solutions or bad solutions. *AP* however provides homogeneous solutions. We think that *AP* finds difficulties on this problem because of its weak exploration mechanism.

4.2 Second symbolic regression problem

For this second comparison the studied function is $f(x) = x^3 \cdot e^{-x} \cdot cos(x) \cdot sin(x) \cdot (sin^2(x) \cdot cos(x) - 1)$ (Fig.8).

The parameter setting is identical to the previous subsection The set of 200 fitness case is generated in [0,10].

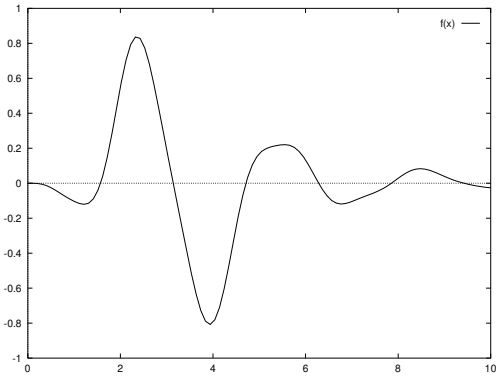

Figure 8: $f(x) = x^3 \cdot e^{-x} \cdot cos(x) \cdot sin(x) \cdot (sin^2(x) \cdot cos(x) - 1)$

Table 2: Comparison between *GP* and *AP* for the symbolic regression problem for the function $f(x) = x^3 \cdot e^{-x} \cdot cos(x) \cdot sin(x) \cdot (sin^2(x) \cdot cos(x) - 1)$

	GP		*AP*	
	nodes number	adjusted fitness	nodes number	adjusted fitness
Best	47	0.0530	15	0.0632
Worst	14	0.0264	31	0.0306
Average	18.85	0.0342	19.225	0.03815

Results (Tab. 2) show that *AP* has on average slightly better results than *GP*. *AP* has the best results with a number of nodes smaller than *GP*. Regression problems are known to be difficult for *GP*.

4.3 11-multiplexer problem

The output of the boolean multiplexer is a boolean value (0 or 1) of the data bit selected by the a addresses bits of the multiplexer (Fig. 9)). The multiplexer inputs are an address on 3 bits a_0 to a_2, and a value on 8 bits (d_j). The goal is to obtain as output the value d_j which corresponds to the address provided by $a_0 ... a_2$. For example, in Fig. 9, data is $(01000000)_2$ and address is $(110)_2$, the multiplexer must provide the bit corresponding to bit d_6.

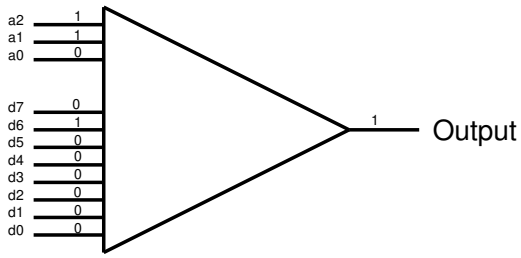

Figure 9: Boolean 11-multiplexer with input of 11001000000 and output of 1

The function has 11 boolean arguments. The fitness function is the number of incorrect outputs for the whole pattern (2048 different inputs).

The terminal set is the various entries { a_0, a_1, a_2, d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7 } without making any distinction between address bit and data bit. The function set is { *and*, *or*, *not*, *if* }, *not* accepts only one argument, *or* and *and* accept two arguments and *if* accepts three arguments, if returns the value of the second argument if the first argument is equal to 1 else the third argument is returned.

The parameters are: 20 generations, population of 50 individuals (size between 2 and 6)

For this problem (Tab. 3) *AP* provides better results than GP. The best result for *AP* has a higher fitness than *GP* and the size of the tree is smaller (7 nodes).

5 Conclusions and future works

This is an on-going study on the automatic generation of program using the ants paradigm. *AP* automatically generates programs like *GP*, and uses some elements of this method. *AP*

Table 3: Comparison between *GP* and *AP* for the 11-multiplexer problem

	GP		*AP*	
	nodes number	adjusted fitness	nodes number	adjusted fitness
Best	44	0.6641	7	0.6875
Worst	9	0.5938	23	0.6104
Average	16.75	0.6256	15.8	0.6295

uses a global memory (pheromone) where *GP* uses only local memory. We have seen that the second regression problem seems harder for *GP* than for AP. This little improvement is in our opinion brought by the collective cooperation of ants. Moreover, it seems that the best solutions provided by the *AP* algorithm are smaller than those provided by the *GP* algorithm. However, the parameters setting used in this first version of *AP* is very restrictive. For instance, the size of the trees is fixed at the beginning of the run. We plan to make the size of individuals evolve during run time. This will allow a fuller exploration of the search space. Future works also include local optimization of the trees generated by *AP*. As quoted by Bonabeau *et* al [1], hybridisation of an *AS* and a local search obtains, in some cases, world-class results.

References

[1] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence, From Natural to Artificial Systems*. A volume in the Santa Fe institute studies in the sciences of complexity. Oxford University Press, 1999.

[2] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In F.Varela and P.Bourgine (Eds.), editors, *ECAL91-European Conference on Artificial Life*, pages 134–142. Elsevier Publishing, 1992.

[3] David Corne, Marco Dorigo, and Fred Glover, editors. *New ideas in Optimization*. Advanced Topics in computer science series. Mc Graw Hill, 1999.

[4] M. Dorigo and L.M. Gambardella. Ant colonies for the traveling salesman problem. Technical Report TR/IRIDIA/1996-3, IRIDIA, Université Libre de Bruxelles, 1996.

[5] M. Dorigo and L.M. Gambardella. A study of some properties of Ant-Q. In *Proceedings of PPSN IV-Fourth -International Conference on Parallel Problem Solving From Nature*, pages 656–665, Berlin, Germany, September 22-27 1996.

[6] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics*, 1-Part B(26):29–41, 1996.

[7] L. M. Gambardella, E. D. Taillard, and M. Dorigo. Ant colonies for the quadratic assignment problem. *Journal of the Operational Research Society*, 50(2):167–176, 1999.

[8] L.M. Gambardella and M. Dorigo. Ant-Q: A reinforcement learning approach to the traveling salesman problem. In A. Prieditis and S. Russell (Eds.), editors, *Proceedings of ML-95 - Twelfth International Conference on Machine Learning*, pages 252–260, Tahoe City, CA, 1995. Morgan Kaufmann.

[9] John Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992.

[10] John Koza. *Genetic Programming II: Automac Discovery of Reusable Programs.* The MIT Press, 1994.

[11] John Koza, Forrest Bennet III, David Andre, and Martin Keane. *Genetic programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann, 1999.

[12] Rafal Salustowicz and Jérgen Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):pp.123–141, 1997.

[13] T. Stétzle. Max-min ant system for quadratic assignment problem. Technical Report AIDA-97-04, AIDA, Darmstadt University of Technology, Computer Science Department, 1997.

[14] E-G. Talbi, O. Roux, C. Fonlupt, and D. Robillard. Parallel ant colonies for combinatorial optimization problems. In Jose Rolim et al., editor, *BioSP3 Workshop on Biologically Inspired Solutions to Parallel Processing Systems, in IEEE IPPS/SPDP'99 (Int. Parallel Processing Symposium / Symposium on Parallel and Distributed Processing*, pages 239–247, San Juan, Puerto Rico, USA, April 1999. Lecture Notes in Computer Science Vol.1586, Springer-Verlag.