A Grid-based Ant Colony System for Automatic Program Synthesis

Sergio A. Rojas[1], Peter J. Bentley[1]

[1] Department of Computer Science
University College London
WC1E 6BT Gower Street, London, UK
{s.rojas, p.bentley}@cs.ucl.ac.uk

Abstract. The Ant Colony Metaheuristic was originally proposed for tackling optimization problems. More recent research has suggested that it can be applied for automatic generation of programs. By allowing the artificial ants to visit functions and terminals nodes, they become able to build pheromone trails that represent computer programs for optimizing a fitness domain-specific function. In this paper a novel approach is addressed using a grid architecture as a more suitable discrete world to be explored by the ants. The resulting system was applied to automatically produce programs to solve Boolean functions.

1 Introduction

Since computers were invented, it has been a dream of human programmers to enable the automatic creation of code out of a set of specifications, without human intervention. The automatic programming research field has been gratefully influenced by the major achievements made by Genetic Programming (GP) in the last decade or so [7, 9]. GP is based on a stochastic process to iteratively modify a pool of program solution candidates by means of some genetic operations including selection, reproduction and mutation. As a result, the average fitness of the population tends to increase in similar fashion to the genetic algorithm (GA) on which GP was inspired. Recently remarkable results have been reported [9] with synthesized programs with increasingly levels of complexity (including loops, functions definition and temporal memory).

Also a decade or so ago the Ant Colony Metaheuristic (ACO) was proposed [3]. This idea, inspired in the foraging behaviour of real ants, state that a colony of agents are able to explore the search-space of a complex problem. By means of a indirect communication mechanism known as stigmergy (pheromone laying), ants can share local and global information that can lead to the construction of shorter paths in that space. Artificial ant colonies have been successfully applied mainly to a number of optimization problems like the traveling salesman problem (TSP) [2]. In order to extrapolate this interesting idea to other domains several issues must be taken into account such those of defining the discrete world (problem-specific states and adjacency), the state transition policy, the stigmergy dynamics, the ants internal state, the pheromone laying timing, the quality measure of the paths found and the local heuristics to be combined.

Although some previous attempts have been made for automatic programming using the ACO metaheuristic, here we address these issues with a novel approach based on a grid definition of the discrete world that the ants have to explore. It seems that this is a more suitable representation for building programs out of pheromone trails followed by the ants.

The paper is organized as follows. The next section is an overview of some related work. Section 3 describes the proposed approach. In section 4 preliminary experiments are shown. The last section is devoted to the conclusions and a discussion of outgoing and future work.

2 Related work

2.1 GP and Variations

In standard GP, programs are represented as trees containing function and terminal nodes. More complex programs may comprise a number of trees to include a main branch, and several loop and function-definition branches. This allows GP to obtain more powerful programs [9]. However some researchers argue that expression trees are not the only or even the most natural way to synthesize programs [5, 12, 14].

A number of alternative variations to the tree representation have been proposed. In Cartesian GP [13] the genome is represented as a linear array of indexes to functions and variables. Partial results can be shared by connecting the outputs of one function gate with the inputs of another located in a posterior loci in the genome. In this way, some of the genes can be activated/deactivated and hence may or may not be present in the expression of the final program. Another approach for linearising the expression tree of the programs is the so-called Gene Expression Programming [4]. Here each individual is represented as a character string of fixed length similar to GA. Each symbol in the genome defines a function or a terminal. The mapping to the phenotype program is carried on by a level-order traversal of the tree (which is equivalent to a breadth-first search) compared to the pre-order used in common GP. As all the strings are of fixed-length the genetic operations for crossover and mutation can be easily implemented, and also the presence of introns is natural [4].

Another interesting variation in the representation of programs is the Parallel Distributed GP (PDGP) [15]. In this approach the tree is represented as a graph with functions and terminals nodes located over a grid. In this way it is possible straightforward to execute several nodes concurrently and even to represent loops and code reusing by the addition of labels in the links of the graph.

Yet another variant of GP is known as Probabilistic Incremental Program Evolution (PIPE) [18]. Even though PIPE adopts the tree representation for a program, it does not maintain a population of individuals. Instead it builds a probabilistic prototype tree in which each node comprise a probability distribution over all the possible elements in the function and terminal sets. Thus the programs are generated using a stochastic process based on that distribution, and the tree is modified by a reinforcement learning technique combined with mutation operators.

It is worth noting that most of these approaches claimed to obtain better results than GP for specific problems.

2.2 ACO and Automatic Programming

One of the best-known instantiations of the ACO algorithm was called the Ant Colony System (ACS) [3] and was used for solving the traveling salesman problem (TSP). In this method ants are randomly located in a graph with numbered nodes that represent cities. Each ant visits the next city as a consequence of a rule that allows for the pheromone concentration and a local heuristic towards the closest neighbor. After all ants have finished up their tours, they are evaluated and the best ant is allowed to deposit pheromones over the edges of the cities in its tour. The Ant System successfully outperformed other methods for medium-size TSP problems, and hence researchers have been encouraged to perform further research in the application of ACO to different optimization problems [2].

The first attempt to apply the ACO algorithm for automatic programming was reported in [17] with the name Ant Programming (AP). They use the ants to navigate the nodes of a prototype tree similar to that of PIPE, but using a pheromone distribution instead of a probability distribution. The pheromone update obeys the ACO dynamics with a reinforcement given by the ant that maximizes a fitness problem-dependant measure. The results obtained by this system did not show remarkable success over some benchmark tasks; nevertheless the authors pointed out that the solutions found by AP tend to be more parsimonious than those given by GP [17].

Recently two more proposals have been reported: the so-called Ant Colony Programming (ACP) [1] and Generalized Ant Programming (GAP) [6]. ACP was used for solving symbolic regression problems with ants that explored a tree with two different versions. In the first version nodes were expressions corresponding to functional and terminal elements. In the second, nodes store assignment instructions working over function, terminal and temporary variables. The authors had to tackle problems such as programs that are not closed or programs with redundant instructions (they need to carry out post-generation program pruning). They state that when using the instructions tree the solution found were more accurate, albeit they also take longer to execute [1]. On the other hand, GAP uses an ant colony approach to build programs specified by a context-free grammar. A financial stocks market application was chosen as test bed. Programs obtained by this method showed approximation results that outperformed other presented in literature [6].

3 A Grid-Based Ant Colony System for Program Synthesis

3.1 Overview

Based on the ideas above, a novel instantiation of the ACO algorithm for automatic programming, the Grid Ant Colony Programming (GACP), is presented in this section. In this approach a population of ants is used to navigate across the nodes of a grid, each one associated with a symbol of the functional or terminal set of instructions. Taken the tours given by the colony in each iteration of the algorithm, the system is provided with a population of fixed-length strings that can be translated to functional programs using an order-level traversal. The search process is guided by the dynamics of laying/evaporation pheromones. A correlation instead of a raw or normalized fitness case is used to evaluate the quality of the paths followed by the ants. The reminder of this section provides an explanation of each component of the system.

3.2 Grid Representation of the World

The reason why using a grid instead of a graph (like in the ACS) or a tree (like in GP, PIPE, AP or ACP) complies with the fact that in order to explore the program search space, an ant is allowed to visit a node several times. Therefore an additional temporal parameter must be considered when choosing the next edge to follow. This will depend on the point of the tour that the ant is following. Without this information it will not be able to discriminate the relevance of the pheromone concentration for that particular point. This is clearly shown in figure 1. Assume that one ant builds a program by jumping from one node to another during five times. The node **X** appears twice in the program **OR(X,AND(X,Y))**. In the graph representation (1a), after visiting the first **OR** and **X** nodes, the ant can not decide between **X->AND** or **X->Y** as the next edge to follow in the third step. On the other hand, the grid representation (1b) holds different edges for each step, so the ant will not get confused. Alternatively the grid can be thought as an array of graphs with a temporal index (1c) to control the step that the ant is jumping (this type of view is particularly useful for implementation purposes).

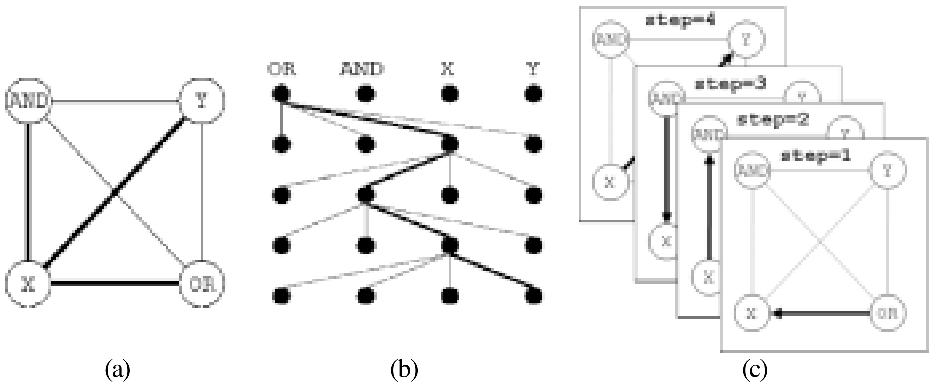


Fig. 1. Different representations for the world to be explored by ants in order to build the program **OR(X, AND(X, Y))**. Chosen edges are shown in bold. (a) Edges in a single graph are not differentiated for every step. (b) A grid maintaining a separate set of edges for each chosen instruction (each step corresponds to a row in the grid). The program can be retrieved reading from top to down, similarly to a feed- forward network. (c) Alternative view of the grid as a matrix of directed graphs.

The grid is a 2D network of nodes similar to that found in PDGP. Each row in the grid contains the same number of columns as the elements in the functional and terminal set. The adjacency of the grid are restricted to connections only between two consecutive layers of nodes (similar to a feedforward neural net). Besides, the number of rows is determined by the depth parameter δ, and the arity parameter γ. Firstly, the parameter δ defines the maximum number of functional nodes that an ant is allowed to visit from the beginning of its tour. Secondly, the γ parameter is the number of arguments of the function with maximal arity in the functional set. With these values it is possible to compute the number of rows of the grid as defined in equation (1), that is, the number of steps that an ant must accomplish to guarantee the closeness of the generated programs. This is true even when one program contains δ times the function with maximum arity γ [4].

$$steps = \delta + (\delta * (\gamma - 1) + 1) \qquad (1)$$

3.3 Ant Internal State and Program Synthesis

Each ant has a memory to keep a trace of its tour. This memory is a character string containing the symbol identifiers of the nodes visited in each step. The string can be considered a kind of genotype that can be mapped to a phenotype (program tree) by using a traversing algorithm. A level-order traversal (breadth-first search) was chosen in order to keep the functional nodes into the first δ nodes of the string, in the same way as proposed for GEP [4]. The final synthesized program is expressed as a sentence in prefix notation. Suppose for example that the colony is requested to solve the 6-multiplexor problem (fig. 2). For this problem the functional set is **`F={I}`**, where `I` represents the function `IF(•,•,•)` and terminal set is comprised of the inputs **`T=`**`{a,b,w,x,y,z}` with a maximum arity γ=3 and depth δ=4. The best path followed by any ant, as showed in fig 3(a), is **`IaIIbwxbyzxw`** which is mapped (3b) to the program **`IF(a,IF(b,w,x),IF(b,y,z))`**. Observe that during the mapping, the last two steps of the path are discarded.

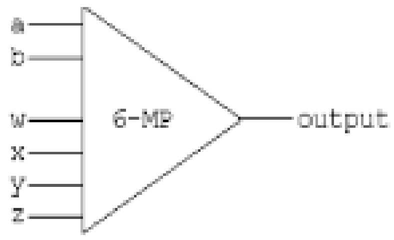
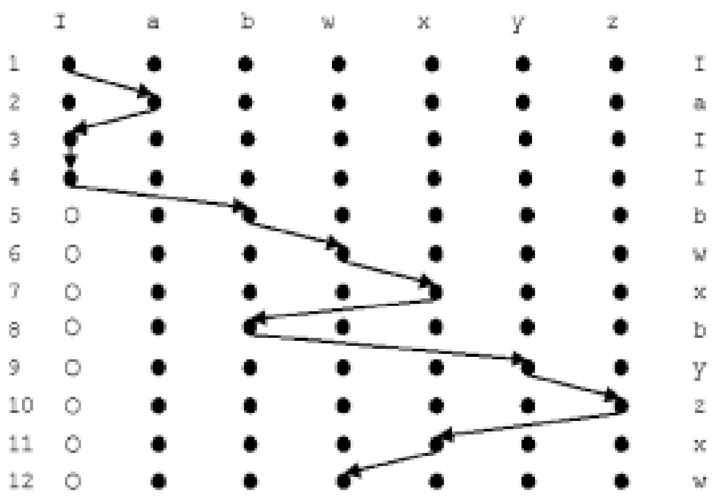
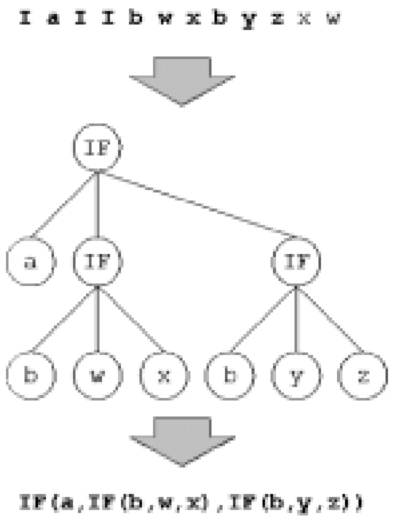


Fig. 2. The 6-multiplexor. Inputs `a` and `b` are used to code the address of the line `w`, `x`, `y`, `z` which will be retrieved at the output.



(a)



(b)

Fig. 3. Example of a path followed by an ant to solve the 6-multiplexor problem. (a) Grid representation for the problem. The nodes chosen by the ant are shown on the right. Step numbers are shown on the left. After the 4th step, nodes corresponding to functions are banned to the ant. (b) Program synthesis. The character string memory of the ant is mapped to a program tree by an order-level traversal. The executable program is shown below. During the mapping, the two last positions of the path were discarded because they are non-coding regions of the "genotype".

3.4 Stigmergy Dynamics

The pheromone updating rules are the same proposed for the ACS [3], as can be seen in equations (2) and (3),

$$\tau(r,s) = (1-\alpha)\tau(r,s) + \alpha\tau_0 \qquad (2)$$

$$\tau(r,s) = (1-\alpha)\tau(r,s) + \alpha\Delta\eta \qquad (3)$$

where $\tau(r,s)$ is the pheromone level between nodes r and s. Equation (2) is executed on-line each time an ant choose to move from r to s. This is intended to simulate the pheromone evaporation caused by all the ants that followed the same path. On the other hand, equation (3) is executed off-line in each iteration by a daemon that takes the edges of the tour completed by the best ant, in order to reinforce their pheromone concentration with a quantity proportional to the winner's fitness. α is a tuning parameter such that $0 < \alpha < 1$.

3.5 State Transition Policy

The transition policy followed by the ants in GACP resembles that of the ACS [3] with a subtle difference, the whole colony is moved concurrently from one row to the following in the grid. For each step a parameter $0 < e0 < 1$ is used to randomly divide the colony in two groups: exploitation and exploration ants. An exploiting ant is biased towards the path with the largest concentration of pheromone $\tau(r,u)$, so it decides where to move using equation (4). The affinity value $\eta(r,u) \in \{0,1\}$ is used to prevent the ant to choose an invalid edge (for example after δ steps, ants are not allowed to follow nodes belonging to functional set, so $\eta(r,u) = 0$ for them). In contrast, an explorer ant decides which neighbor node to jump by means of a roulette wheel rule based on the probability obtained with equation (5). In both equations r represents the current node visited by the ant, and u are all the nodes connected to it in the subsequent row of the grid.

$$s = \arg\max_u [\tau(r,u) * \eta(r,u)] \qquad (4)$$

$$p_s = \frac{[\tau(r,u) * \eta(r,u)]}{\sum_i [\tau(r,i) * \eta(r,i)]} \qquad (5)$$

3.6 Fitness function

The paths explored by the ants in the program search-space of GACP have no length, but yet they can be assessed with a fitness function which is problem-dependant. The fitness function must allocate a better score for those programs that are closer to the given set of specifications. This score will be reinforced in the pheromone trail of the best ant.

So far the GACP system has been tested to build programs to solve Boolean problems. For this kind of problems usually a score is given by the number of correct answers obtained by the program in a set of test cases. However it has been found that the correlation measure of the number of true positives (*tp*), true negatives (*tn*), false positives (*fp*) and false negatives (*fn*) cases could give a more accurate measure of the predictive power of the synthesized program [8]. So the correlation rate given by (6) was chosen as a fitness function. A good solution for the problem will have a fitness near to one, whilst a worse program will have a score close to zero.

$$fitness = 0.5 * \left(1 - \frac{tp * tn - fn * fp}{\sqrt{(tn + fn)(tn + fp)(tp + fn)(tp + fp)}} \right) \tag{6}$$

```
Scores (correlation) -> Solutions
--------------------------------------------------------------

Run 1: 0.89 ->  iif(a,iif(x,x,w),iif(b,y,z))
Run 2: 1.00 ->  iif(a,iif(b,w,x),iif(b,y,z))
Run 3: 0.88 ->  iif(b,y,iif(iif(w,a,a),x,z))
Run 4: 0.88 ->  iif(a,w,iif(b,y,z))
Run 5: 0.89 ->  iif(a,iif(b,w,x),iif(y,b,y))
Run 6: 0.88 ->  iif(a,iif(b,w,x),y)
Run 7: 0.88 ->  iif(a,x,iif(b,y,z))
Run 8: 0.88 ->  iif(a,w,iif(iif(a,w,b),iif(b,y,a),z))
Run 9: 0.88 ->  iif(a,x,iif(b,y,z))
Run 10: 0.88 ->  iif(b,w,iif(a,x,z))
Run 11: 0.88 ->  iif(a,x,iif(b,y,z))
Run 12: 1.00 ->  iif(b,iif(a,w,y),iif(a,x,z))
Run 13: 0.88 ->  iif(a,w,iif(iif(b,b,b),y,z))
Run 14: 0.88 ->  iif(b,w,iif(a,x,z))
Run 15: 0.88 ->  iif(iif(iif(b,b,b),b,a),iif(b,y,x),z)
Run 16: 0.88 ->  iif(b,y,iif(a,x,z))
Run 17: 0.88 ->  iif(a,w,iif(iif(a,z,b),y,z))
Run 18: 1.00 ->  iif(b,iif(a,w,y),iif(a,x,z))
Run 19: 1.00 ->  iif(b,iif(a,w,y),iif(a,x,z))
Run 20: 0.88 ->  iif(a,iif(iif(a,b,z),w,x),z)
Run 21: 0.88 ->  iif(b,w,iif(a,iif(w,x,x),z))
Run 22: 0.88 ->  iif(a,x,iif(b,y,z))
Run 23: 0.81 ->  iif(iif(iif(w,x,y),a,b),w,z)
Run 24: 0.88 ->  iif(iif(b,iif(a,a,w),b),w,iif(a,x,z))
Run 25: 1.00 ->  iif(b,iif(a,w,y),iif(a,x,z))
Run 26: 0.88 ->  iif(b,y,iif(a,x,z))
Run 27: 0.88 ->  iif(a,iif(b,w,x),y)
Run 28: 0.88 ->  iif(b,w,iif(a,iif(x,x,x),z))
Run 29: 0.88 ->  iif(a,w,iif(b,iif(a,z,y),z))
Run 30: 0.88 ->  iif(a,w,iif(b,y,z))

Average best fitness: 0.89
```

Fig. 4. System output for the 6-MP task. Correlation fitness and program explored by the best ant during 30 experiments.

4 Preliminary Experiments

4.1 6-Multiplexor

A number of preliminary experiments using GACP with the parameters shown in table 1 have been performed. Although some work is still needed for tuning the parameters and the implemented algorithms, so far results are encouraging. During 30 experiments, GACP has found the solution to the problem with a fitness score average of 0.89. In five occasions the system synthesized perfect programs. The results are summarized in figure 4. The number of correct fitness cases per tour (averaged over the 30 experiments) are shown in figure 5. The correlation fitness of the best ant in each experiment is plotted in figure 6.

Table 1. Parameters used in GACP for the 6-MP problem.

Parameter	α	*e0*	γ	δ	*colony_size*	*generations*	*fitness_cases*
Value	0.1	0.8	3	6	100	80	64

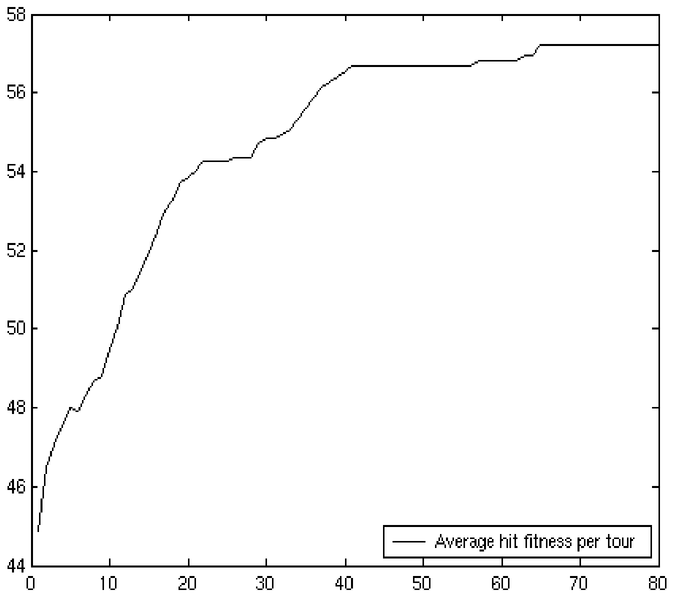


Fig. 5. Number of corrected classified cases per tour averaged over 30 experiments.

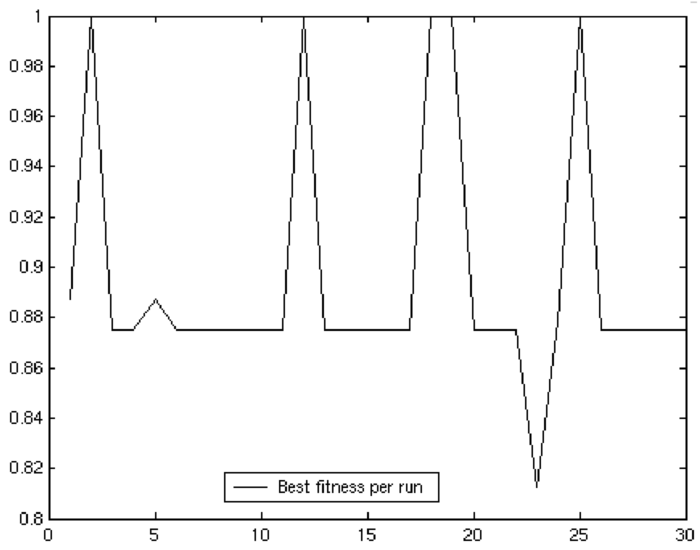


Fig. 6. Correlation fitness of the best ant per experiment.

4.2 11-Multiplexor

Just some few trials were performed for this task. Results are summarized in figure 7. Despite the fact that more exhaustive experiments should be conducted, the programs obtained by GACP had on average a 75% of correct answers to the 2048 test cases, which is better than the 59% reported with AP in [17] for the same task.

```
[0.62] 1<1216> 2<1248> 3<1216> 4<1200> 5<1248> … 8<1248>
[0.62] 9<1280> 10<1248> 11<1248> 12<1216> 13<1280>
[0.69] 14<1408> 15<1280> 16<1344> … 74<1408> 75<1280> 76<1280>
[0.69] 77<1408> 78<1408> 79<1408> 80<1408> 81<1408>
[0.75] 82<1536> 83<1536> 84<1536> … 166<1440> 167<1504> 168<1504>
[0.77] 169<1568> 170<1568> 171<1568> … 198<1568> 199<1568>
200<1568>

Solution proposed: iif(c,iif(b,iif(a,r,w),y),iif(iif(z,w,v),x,z))

[0.62] 1<1280> 2<1280>
[0.63] 3<1280>
[0.63] 4<1280>
[0.63] 5<1280>
[0.63] 6<1280>
[0.65] 7<1312>
[0.69] 8<1408>
[0.69] 9<1408> 10<1408> 11<1408> 12<1408> … 37<1408> 38<1408>
[0.75] 39<1536> 40<1536> 41<1536> … 298<1536> 299<1536> 300<1536>

Solution proposed: iif(b,iif(a,iif(c,r,s),w),t)
```

Fig. 7. Some programs found by GACP for the 11-MP problem.

5 Conclusions and Directions for Future Work

This paper has introduced a novel instantiation of ACO for automatic programming. Grid Ant Colony Programming (GACP) uses a colony of artificial ants to explore a program search space represented as a grid. The ants develop programs by taking advantage of global and local information harvested through a stigmergy dynamics. The grid (as an alternative to a tree) allows the ants to maintain temporal data of the steps they follow. Preliminary experiments showed that is possible to find perfect solutions to Boolean problems of medium size. For bigger problems, GACP performs better than some previous approaches. Some of the advantages of this approach are the closeness and validity of any generated program, and the possibility of parallelized synthesis of programs.

The system reported here is ongoing work and several specific issues such as scalability, inclusion of constants, experiments with different fitness functions, use of other ACO variants, automatic function definitions, inclusion of loops and memory management, may be targets for further research. Moreover there are yet other significant topics that can emerge from this research. The first one is using GACP with the combination of ACO and local heuristics. This is motivated by the fact that most of the time the colony built programs with the correct structure needed to solve the problem, but with a different ordering of terminals. In this sense, a local heuristic that reassembles those terminals in different ways could be very useful to improve the performance of the system.

Secondly, recent advances in the theoretical research of automatic programming by GP have been done with the aim of providing deeper explanations [11, 16]. As it was showed with GACP, programming by means of ACO can viewed into the framework of genotypic/phenotypic mapping with innovative "genetic" operations to explore the search space. It is anticipated that an extension of those theoretical studies to this new field may prove fruitful.

One further avenue for research may be to use other natural computation approaches for automatic programming. GP has shown successful application to some domains. Artificial ants may also be applied. Therefore, it seems likely that particle swarms, molecular computation or immune systems could be used for the same purpose. This could give rise to a whole class of algorithms intended to achieve that long dream of computer programmers, this time by "bio-inspired automatic programming".

References

1. Boryczka, M.; Czech, Z. Solving Approximation Problems by Ant Colony Programming. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02), New York City, NY, USA, July 9-13, (2002)
2. Dorigo, M., Di Caro, G., Gambardella, L. Ant Algorithms for Discrete Optimization. Artificial Life, Vol. 5, No.3. (1999). 137-172.

3. Dorigo M., Maniezzo, V., Colorni, A. The Ant System: Optimization by a Colony of Cooperating Agents. IEEE Transactions on Systems, Man, and Cybernetics-Part B, 26(1). (1996). 29-41.
4. Ferreira, C. Gene Expression Programming: a New Adaptive Algorithm for Solving Problems. Complex Systems, Vol. 13, issue 2: 87-129. (2001).
5. Kantschik, W.; Banzhaf, W. Linear-Tree GP and its comparison with other GP structures. Proceedings of the 4th European Conference on Genetic Programming, (EuroGP 2001), Italy, April 18-20, (2001).
6. Keber, C., Shuster, M. Option valuation with Generalized Ant Programming. Proceedings of the Genetic and Evolutionary Computation Conference, (GECCO'2002), (2002).
7. Koza, J. Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: The MIT Press. (1992).
8. Koza, J., Bennett III, F., David, A. Using programmatic motifs and genetic programming to classify protein sequences as to extracellular and membrane cellular location. Proceedings of Evolutionary Programming VII. 7h International Conference (1998).
9. Koza, J., Keane, M., Matthew J., Mydlowec, W., Yu, J., Lanza, G., and Fletcher, D. Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers. (2003).
10. Levine, J., Ducatelle, F. Ants Can Solve Difficult Bin Packing Problems. Proceedings of the 1st Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2003), Nottingham, UK, August 13-16th. (2003).
11. McPhee, N. F., Poli, R. A schema theory analysis of the evolution of size in genetic programming with linear representations. Proceedings of EuroGP'2001, Lecture Notes in Computer Science 2038, pp. 108-125. (2001).
12. Miller, J. What bloat? Cartesian Genetic Programming on Boolean problems. Late Breaking Papers, Proceedings of the 3rd Genetic and Evolutionary Computation Conference, (GECCO'01), pp. 295 -302 , (2001).
13. Miller, J.; Thomson, P. Cartesian Genetic Programming. Proceedings of the Third European Conference on Genetic Programming, (EuroGP 2003), Edinburgh, April 15-16, (2000).
14. O'Neill M., Ryan C. Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Language. Kluwer Academic Publishers. (2003).
15. Poli, R. Evolution of Graph-like Programs with Parallel Distributed Genetic Programming. Proceedings of the Seventh International Conference on Genetic Algorithms, Michigan State University, July 19-23, (1997).
16. Poli, R., Langdon, W. B. Schema Theory for Genetic Programming with One-point Crossover and Point Mutation. Evolutionary Computation Journal, 6(3): 231-252. (1998).
17. Roux, O.; Fonlupt, C. Ant Programming: Or How to Use Ants for Automatic Programming. Proceedings of the Second International Conference on Ant Algorithms (ANTS2000), Belgium, September, (2000).
18. Salustowicz, R., Schmidhuber, J. Probabilistic Incremental Program Evolution. Evolutionary Computation 5(2):123-141. (1997).