Automatic Programming with Ant Colony Optimization

Jennifer Green
University of Kent
jg9@kent.ac.uk

Jacqueline L. Whalley
University of Kent
J.L.Whalley@kent.ac.uk

Colin G. Johnson
University of Kent
C.G.Johnson@kent.ac.uk

Abstract

Automatic programming is the use of search techniques to find programs that solve a problem. The most commonly explored automatic programming technique is genetic programming, which uses genetic algorithms to carry out the search. In this paper we introduce a new technique called Ant Colony Programming (ACP) which uses an ant colony based search in place of genetic algorithms. This algorithm is described and compared with other approaches in the literature.

1 Introduction

The aim of this paper is to present some preliminary work which applies Ant Colony Optimization techniques (Dorigo *et al.*, 1996; Bonabeau *et al.*, 1999) to the automatic creation of computer programs.

Automatic programming allows the programmer to avoid the tedious task of creating a program to solve a well-defined problem (Boryczka and Wiezorek, 2003). Automatic programming requires the specification of goals that are to be realized by the program and it is on the basis of this specification that the program is constructed automatically.

Previously, work towards automatically generated programs largely employed Genetic Programming (GP) techniques. GP takes specific inputs and produces desired outputs to solve a specified problem using evolutionary inspired techniques such as genetic algorithms (Koza, 1992; Banzhaf *et al.*, 1998; de Jong, 1999).

Typically a problem is defined as a number of inputs and the expected output for each input. A heuristic search technique is then carried out on a space of graphs where the nodes represent functions, variables and constants and the graph overall represents a parse tree for that function.

Functions are usually defined mathematically in terms of arithmetic operators, operands and boolean functions. The set of functions defining a given problem is called a function set and the collection of variables and constants to be used are known as the terminal set. Genetic programming represents a program as a tree structure, the nodes containing a function from the function set and the leaf nodes holding a member from the terminal set.

According to Boryczka and Wiezorek (2003), there are four preparatory steps which much be accomplished before a searching process for a program can commence namely, choice of terminal symbols, choice of functions, definition of the fitness function, and defining the termination criteria.

The program components are made up of terminal symbols and functions. The choice of these components and the definition of the fitness function form the primary means of defining the problem space that will be searched. Variable parameters such as population size, probabilities of crossover and mutation, maximum tree size etc. belong to the set of control parameters.

Many of the principles used in GP can be adapted to develop an ant colony algorithm that may be applied to automatic programming. Ant colony systems arose from research into systems inspired by the behavior of real ants (Wilson and Hölldobler, 1990). Therefore the artificial ants or agents used in ant colony systems have some features taken from real ants, for example the choice of route depends on the amount of pheromone.

Earlier work using ant colony algorithms worthy of note include Dorigo *et al.* (1991, 1996), who solved the travelling salesman problem using an ant colony algorithm, and more recently Roux and Fonlupt (2000) who made the first attempt at utilizing ants for automatic programming. Their approach was used to solve some simple problems in symbolic regression and a multiplexer problem.

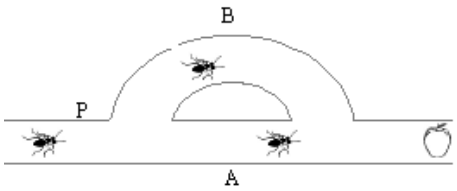

Figure 1: How ants find the shortest path.

2 The Ant Colony Algorithm

"All good work is done the way ants do things: Little by little."

— Lafcadio Hearn

Ants are able to find their way efficiently from their nest to food sources. They do this by laying trails of pheromone as they explore. Once many ants have explored a region around their nest, the routes which lead from the nest to good food sources get the strongest pheromone trail. Essentially this works by a process of positive feedback. If an ant has a choice of trails to follow, the preferred route is the trail with the highest deposit of pheromone (Wilson and Hölldobler, 1990). Initially there will be no pheromone laid down, so ants which take shorter routes will lay down the most pheromone on those routes. This behaviour accounts for why ants are able to find the optimal or shortest route without any need for direct communication or central control.

The algorithm at a very simplified level is as follows (see figure 1. Two ants (A1 and A2) are travelling along route P and come to a junction. A1 takes path A and A2 takes path B. As they are travelling along the route the ants are depositing a pheromone trail both ants continue along their chosen paths, collect the food and return to the nest. A1 will reach the nest first because it has travelled the shortest route. A third ant (A3) now leaves the nest, travels along path P and reaches the junction. At this point A2 has not yet returned through the junction and is still travelling along path B so there is twice the amount of pheromone deposited along path A at the junction as along path B. Therefore A3 will opt for path A thus increasing the pheromone level on path A. The reality is not quite so simple as other factors need to be considered such as evaporation of the pheromone trail.

In our Ant Colony Programming technique, the search space consists of a graph where the nodes are the functions and terminals, and the edges are weighted by pheromone. An example of such a graph is given in figure 2. The ants move through the network of trails within the graph thereby creating hierarchical structured programs. Each node in the graph holds either a function ($f_i \in F$) or a terminal ($t_i \in T$).

In our system this graph is generated by a randomised process. For the symbolic regression problems discussed later, graphs of 80 nodes are used: 10 each of the functions Add, Subtract, Multiply and Divide, and 10 each of the terminals X, 1.0, 2.0 and 5.0. Links are generated at random between each pair of node, with a probability of 0.6 (this needs to be greater than 0.5 so that a *giant component*, where a single component of the graph contains "most" nodes, is created (Watts, 1999; Erdős and Rényi, 1960)).

Each edge is given a "pheromone" weight. This starts out as 1.0 for all edges on initialization.

2.1 Terminal Symbols and Functions

A terminal can be a constant, for example $t_i = 1.0$, or a variable such as $t_i = x$. As in GP, the functions are related to the problem and are, for example, an arithmetic operator ($+, -, \times, \div^1$), a boolean operator (or, and, not) or an arithmetic function (e.g., cos, tan, exp). Every function utilized in this paper has fixed arity (two in all examples) however, it should be possible to extend the function set to include alternative functions (if-then-else) and iterative operators for example while.

The terminal symbols and functions were chosen such that they provided sufficient expressive power to express the solution to a problem (Koza, 1992; Poli, 1996). This means that the problem must be able to be solved by a composition of functions and terminals specified.

2.2 Overall Process

The overall process is as follows. There are a number of *generations*. In each generation a number (100 in the examples below) of ants are started off from starting points (section 2.3), and follow a route through the graph in a way which is biased by the pheromone weights on the edges, as discussed in section 2.4. This route is then interpreted as a program built from the functions and terminals visited as the ant moves across the graph. The fitness of each of these routes

1. protected division to avoid divide-by-zero errors Koza (1992).

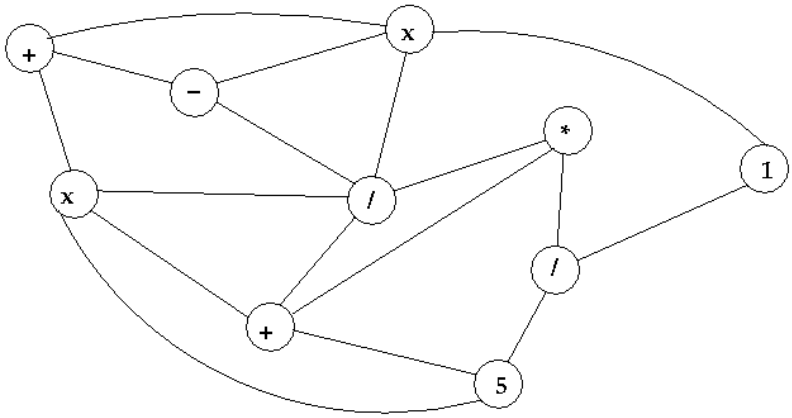

Figure 2: Example of the graph of functions/terminals which is explored by the ants.

is assessed relative to the problem at hand. At the end of each generation the pheromone is updated; firstly the pheromone on the edges which were involved in the most successful programs is increased, then an "evaporation" of pheromone from all edges is carried out.

2.3 Starting Points

At the beginning of each generation 10 ants start from each of 10 starting points. These starting points are fixed throughout a run of the algorithm. Future work will explore the idea of learning the best starting point.

2.4 Ants on the Move

Ants move along the trails or connections between nodes creating a tour of the graph. An ant (a) can move from its current node (c) to any other neighbouring node (n) along an edge (c, n), its decision on which route to take is determined by the strength of the pheromone.

The process is as follows. Initially a random number $q_0 \in [0, 1]$ is generated, this is used to determine whether the ant moves according to the pheromone strength, or whether it travels along a random choice of accessible paths. The random number is compared with a value $q \in [0, 1]$ known as the *learning rate* (Bonabeau *et al.*, 1999). If $q_0 < q$ then the ant will follow a neighbouring edge selected using roulette wheel selection, where the probability of choosing an edge is proportional to the amount of pheromone on that edge compared with all the other edges. Otherwise an edge is chosen uniformly at random from the neighbouring edges. In the examples below this is set to a high value of $q = 0.95$.

When an ant reaches a node it determines if the node is a terminal or a function node. If the ant is on a terminal node, the end of the tour has been reached for that ant. However, if the ant is on a function node it determines how many parameters, p, the function needs. If the function requires more than one parameter, the original ant will reproduce so that the number of ants starting out from the function node is equal to the number of parameters. This reproduction is repeated every time an ant reaches a function node containing a function with an arity greater than one.

One problem with this is that the number of ants touring the graph may become so large that the algorithm becomes highly inefficient. For this reason a limit on the number of child ants being produced was introduced to the algorithm. This limit may be set according to the size of the graph being used. In a typical run of the ACP any number of ants are placed on the graph (the number of ants is ultimately limited by the scope of the graph) simultaneously. Each ant travels along its own independent route following the algorithm described above. This group of ants and their collective tours is called a generation. The procedure is repeated for a set number of generations. After each generation the tours are compared using a fitness evaluation (section 2.6) and the pheromone trail is then updated (section 2.7).

There are some constraints on the depth of the program tree that can be found. The starting points are chosen so that they are always

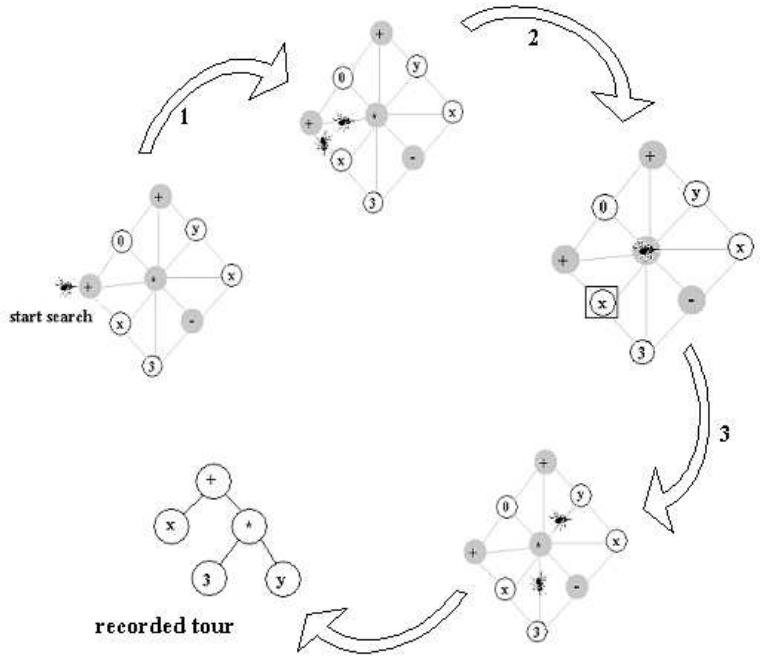

Figure 3: Ants on the move.

function nodes. If an ant (or its ancestors) has explored fewer than some minimum number of edges (in the examples below this is 3), then the ant will always choose a function node if one is accessible from the current node. If is has explored more than some maximum number (in the examples below this is 8), then the ant will always choose a terminal node where one is accessible. This is to ensure (i) that very simple functions which perform well in early generations do not dominate the results, thus producing functions on which the exploratory process can take hold; and (ii) so that excessively large functions are not created.

This part of the process is summarized in figure 3.

2.5 Ants have Memory

Once the graph is initialized and the ants are moving within the graph. An ant will create a tour moving from one node to the next. In order to evaluate the route that the ant has taken a record of the ants tour needs to be created. Each ant has a working memory that stores data about a tour or route. The ants' memory is represented programmatically by a tree structure. In this tree, the nodes contain the functions and the leaves contain the terminals. The depth of the memory tree is limited according the nature of the problem. An example of such a tree is shown in figure 4.

2.6 Calculating Fitness

Each problem put through the ACP has a set of inputs and outputs. The inputs are the values for the variables in the terminal set. Each terminal in a given tour is replaced with the supplied values and then each function in the tour is evaluated to give a result. The result is then compared with the expected output and the difference between the two is analyzed. This process is carried out for each of the specified terminal values, summing the difference to provide a 'raw' overall fitness value (table 1).

The raw fitness for a given tour provides a measure of how fit a solution is, a value of zero indicates an optimal solution. If this is not the case then a standardized fitness can be calculated (Koza, 1992). The fittest tour in a generation therefore is the tour with the lowest raw fitness value.

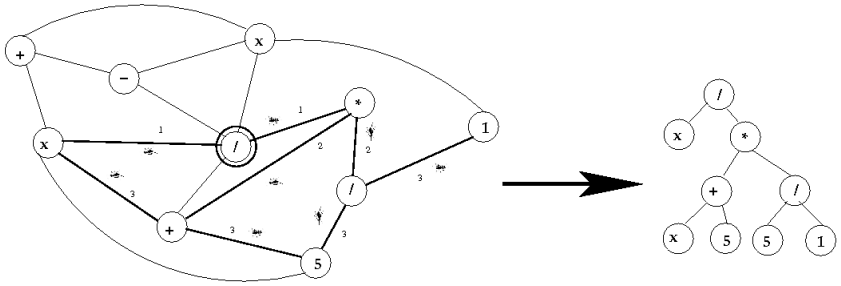

Figure 4: The program tree embedded in the graph.

Value of x	Output of tour	Expected output	Abs. value of Difference
0	12	10	2
1	15	20	5
2	21	30	9
3	28	40	12
4	30	50	20
5	31	60	29
		Fitness	77

Table 1: Calculating the raw fitness.

2.7 Updating Pheromone Levels

Pheromone levels are updated at the end of each generation, globally and locally on individual edges. Global updating has the net result of rewarding the edges of the fittest tours within the generation. The ants with the best tours in the generation deposit pheromone along the edges of its tour. In the examples below the best four ants are chosen, and the pheromone increase by 3.0, 2.0, 1.0 and 1.0 respectively for each edge which is involved in that trail. By making the strength of the pheromone deposit directly related to the quality of the solution, the best solutions are favoured and so the next generation of ants will be more effectively directed by the pheromone trails.

2.8 Pheromone Evaporation

Following this the pheromone level is reduced on each edge; this is driven by the need to model 'real world' pheromone evaporation. Evaporation promotes the exploration of different routes by the ants thereby avoiding congestion caused by rapid convergence on local minima. At the end of each generation the pheromone level on each edge is reduced by between 0-25% (chosen uniformly at random) of its current value.

3 Results and Discussion

As a basic test to see whether the method is effective we have applied this to a number of basic symbolic regression problems. The setup of the algorithm for these problems is given in table 2.

Four such problems were attempted:

1. $f(x) = 6x^2 + 10x + 12$
2. $f(x) = cos(x)$
3. $f(x) = 20x + 10$
4. $f(x) = x^4 + x^3 + x^2 + x$

The results (the mean error per generation averaged over 10 runs) are given in figure 5, 6, 7 and 8. These are compared with a randomized control which is identical except that no pheromone changes are made, so there is no fitness information fed back into the system.

Clearly these are problems which could be solved readily using standard genetic programming techniques Koza (1992); Banzhaf *et al.* (1998); this technique is not competitive with such methods. Nonetheless for a simple first attempt it does demonstrate that fitness improvement does occur.

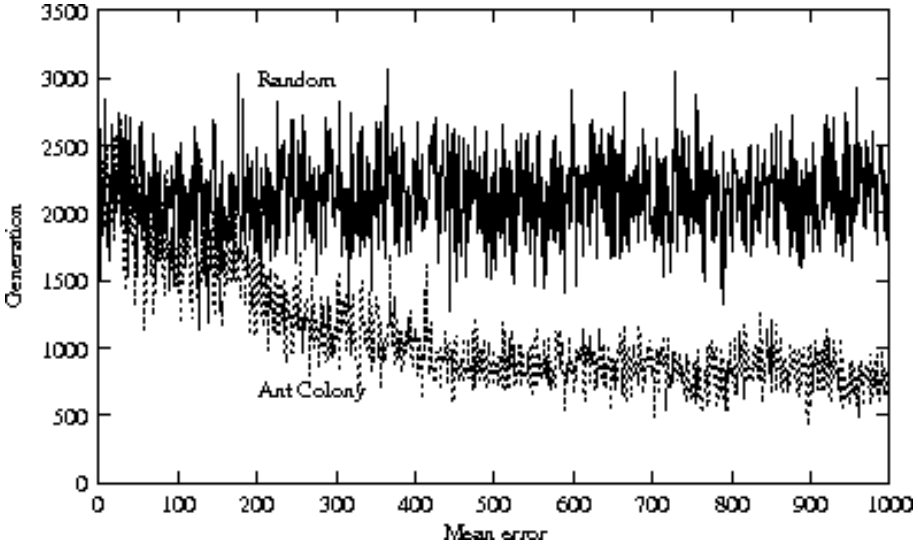

Figure 5: Mean error vs. generations for test function 1.

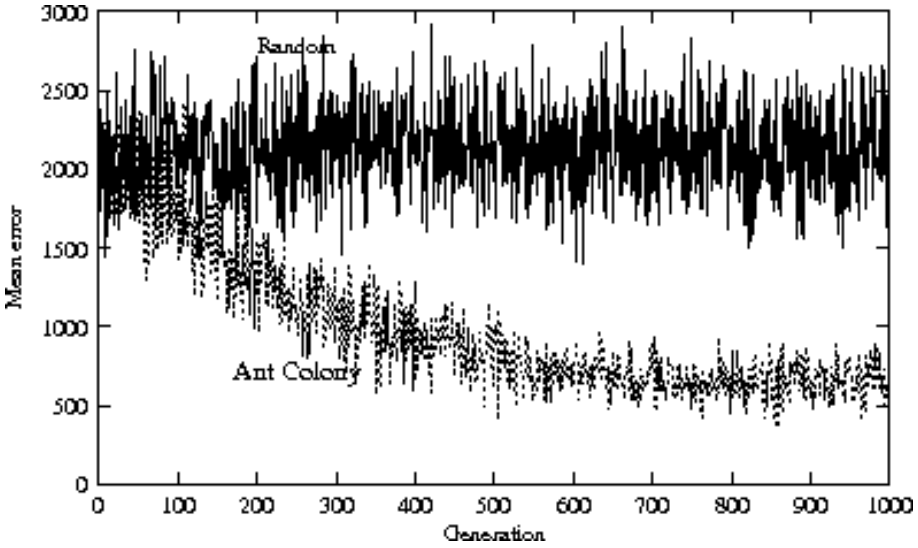

Figure 6: Mean error vs. generations for test function 2.

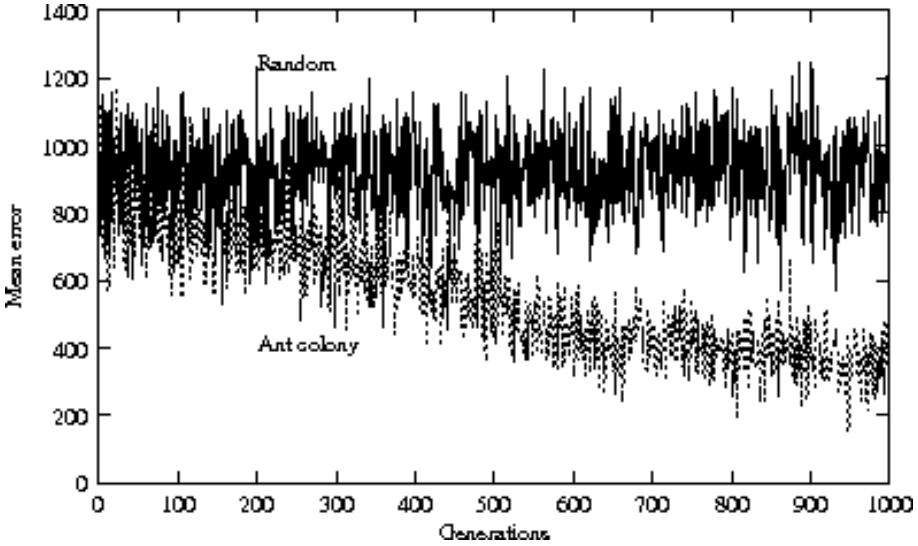

Figure 7: Mean error vs. generations for test function 3.

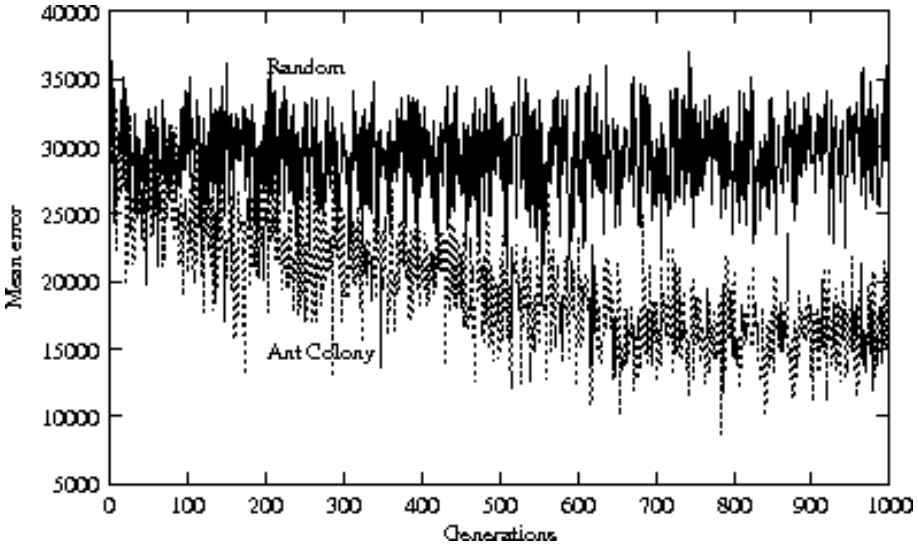

Figure 8: Mean error vs. generations for test function 4.

Parameter	Value
Problem	Minimize the error in a symbolic regression problem
Functions	$+, -, \times, \div$
Terminals	1,2,5,x
Fitness	Absolute value of error for $x = -10, -9, -8, \ldots, 8, 9, 10$
Ants per generation	100
Maximum generations	1000

Table 2: Parameter settings tableau for the symbolic regression problems.

4 Conclusions and Future Work

In this paper we have given a preliminary study on the application of ant colony optimization to automatic programming. It has been demonstrated to have some basic success on simple symbolic regression problems. Whilst the system is not competitive with other systems such as genetic programming, little effort has yet gone into details of how to make the algorithm work well and what parameter choices to make.

Future work will consist of examining these parameter choices more carefully, and examining runs from the program in detail to study what routes ants are taking. This will be followed by a study which attempts to ascertain which, if any, problem-types this search method is particularly suited to.

References

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming: An Introduction*. Morgan Kaufmann.

Bonabeau, E., Dorigo, M., and Theraulaz, G. (1999). *Swarm Intelligence*. Oxford University Press.

Boryczka, M. and Wiezorek, W. (2003). Solving approximation problems using and colony programming. In *Proceedings of AI-METH 2003*, pages 55–60.

de Jong, K. (1999). Genetic algorithms: A 30 year perspective. `http://www.pscs.umich.edu/jhhfest/abstracts.html`.

Dorigo, M., Maniezzo, V., and Colorni, A. (1991). Positive feedback as a search strategy. Technical Report Politechnico di Milano, Italy.

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

Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, **5**, 17–61.

Koza, J. R. (1992). *Genetic Programming : On the Programming of Computers by means of Natural Selection*. Series in Complex Adaptive Systems. MIT Press.

Poli, R. (1996). Introduction to evolutionary computation. The University of Birmingham. `http://www.cs.bham.ac.uk/~rmp/slide_book/slide_book.html`.

Roux, O. and Fonlupt, C. (2000). Ant programming: Or, how to use ants for automatic programming. In M. Dorigo, editor, *Proceedings of ANTS'2000*, pages 121–129.

Watts, D. J. (1999). *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press.

Wilson, E. and Hölldobler, B. (1990). *The Ants*. Springer-Verlag.