# An example of the use of context-sensitive constraints in the ALP system

Maarten Keijzer
Free University Amsterdam
mkeijzer@cs.vu.nl

Mike Cattolico
Tiger Mountain Scientific
mike@tigerscience.com

March 7, 2002

## Abstract

This paper examines the use of context-sensitive constraints in the ALP system. Experiments using a 'dumb' and 'sensible' ant are performed using a base-line definition of ant behaviour which is contrasted with an extended program that disallows certain a-priori 'bad' actions. Comparing both subtree crossover and ripple crossover shows that this particular form of introducing bias significantly helps the system to find solutions.

## 1 Introduction

The ALP system (Keijzer et al., 2001a) is a generalization of grammatical evolution (O'Neill and Ryan, 2001). The main representation vehicle is not a context-free grammar but a fully fledged logic programming system. It is comparable with other logic programming approaches (Wong and Leung, 1997; Ross, 1999); the main difference lies in the use of logic programs directly rather than logic grammars as well as the straightforward representation of genetic material as variable length strings of choices in a logic program.

By virtue of the use of logic programming as the underlying representation, it is possible to implement context-sensistive constraints. The paper presented here does just that: a number of constraints are implemented that use the context — previous choices made in the rest of the program — to exclude certain syntactical constructs. An empirical comparison between a typed subtree crossover and the untyped ripple crossover (Keijzer et al., 2001b) is performed which highlights the differences between these two approaches.

## 2 A Sensible Ant on the Santa Fe Trail

The artificial ant problem has been studied intensively in (Langdon and Poli, 1998) which showed that it is a difficult problem with multiple ridges and local optima. The goal is to find a computer program that steers an ant over a trail of food pieces, eating as much food as possible. The trail that is used is the well-known Santa Fe trail that contains 89 pieces of food. The success criterion for an artificial ant program is then to steer the ant to eat these 89 pieces of food within 600 steps.

The logic program that defines the space of allowable ant programs can be stated as follows.

```
ant(move)
ant(left)
ant(right)
ant(iffoodahead(X,Y)) :-  ant(X), ant(Y).
ant(seq(X,Y)) :-          ant(X), ant(Y).
```

Program 2.1: Program for inducing an artificial ant.

The `move` atom indicates a move forward by the ant, left and right turn the ant 90 degrees on the grid, the iffoodahead instruction branches on the information whether there is a food pellet present in the cell the ant is facing, while the seq operator simply applies its arguments in sequence. A program consisting of these instructions is iteratively applied until time runs out.

In the usual genetic programming notation of terminal and function sets, this logic program above can be described by: T = { move/0, left/0, right/0} and F = { iffoodahead/2, seq/2 }[1]. The Program 2.1 will be extended to implement a few context-sensitive constraints.

The space of possible ant programs contains many ineffective pieces of code that can be identified even before trying to find a program for a specific trail. It is for example ineffective to let the ant move left and subsequently move right without any commands in between as its overall state would not have changed. Furthermore, directly nesting iffoodahead/2 calls is also ineffective as the outcome of the check is already known. This embodies knowledge about the semantics of the function set. We might also assume that if there is food ahead, moving toward the food seems to be a good idea.

These constraints are readily implemented in the ALP system. For this an ant/3 predicate will be used. The first argument will be used to induce the computer program that is evaluated. The second argument is used to specify the input constraints, constraints that are imposed by the caller, while the third argument is used to specify some output constraints: constraints subsequent clauses need to abide. The second and third argument are thus used to store some *context* which can be used to steer the derivation process.

The start clause is of the form ant/1 that calls ant/3, not demanding any constraints on the program (signified by the empty list []), and ignores the output constraints (the underscore symbol). Thus:

```
ant(X) :- ant(X,[],_).
```

[1]The number behind the function names denotes the arity of the function.

The sequence operator is used to propagate the constraints.

```
ant(seq(X,Y),In,Out) :- ant(X, In, C),
ant(Y, C, Out).
```

The two additional arguments In and Out get imposed on the arguments of the sequence operator. The intermediate variable C is used to propagate the constraints from the first subexpression to the other.

The clause that induces the move atom is defined as follows

```
ant(move, C, []) :- \+ member(no_move, C).
```

where \+ is the Prolog symbol for negation by failure: it will only succeed when the execution of its argument fails. The member/2 function is a built-in clause, and is executed in Prolog directly. This clause constrains the move atom to be only applicable when the set of constraints does not contain the no_move atom. A move furthermore removes all constraints, it thus returns the empty list []. Turning to left:

```
ant(left, C, [no_right]) :- \+
member(no_left, C).
```

Is applicable only when there is no no_left constraint. It imposes a no_right constraint to the next action. So when the ant turns left, it will not immediately turn right, because:

```
ant(right, C, [no_left]) :- \+
member(no_right, C).
```

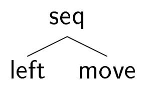
For the iffoodahead function the following constraints are imposed:

```
ant(iffoodahead(X, Y), In, Out) :-
      \+ member(no_if, In),
      ant(X,[no_left, no_right, no_if], OutLeft),
      ant(Y, [no_move, no_if], OutRight),
      intersection(OutLeft, OutRight, Out).
```

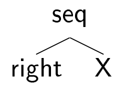
The first literal in the body of the clause imposes the constraint that the clause can only be used when there is no `no_if` constraint pending. Furthermore it specifies that when food is spotted, turning is not allowed, which will necessarily lead to a move as the next action (note that due to the propagating of constraints any number of intermediate sequence operators can occur as long as the first action that is applied is a `move`). When there is no food ahead however, moving is not allowed: this will mean that a construction such as '`iffoodahead(seq(move,X), seq(move,Y))`' is not allowed, as this could equivalently (and shorter) be specified as '`seq(move,iffoodahead(X,Y))`'. For both branches, it is not allowed to immediately check for food again. The clause will return the intersection of the constraints imposed by the two branches, thus if both branches end with imposing the same constraint(s) these will be propagated to the next action.

This in effect implements information about some immediate redundancies in the function set for the artificial ant problem. It reduces the search space by disallowing specific combinations of code. No knowledge about the trail is included in the program, the constraints are imposed to remove redundancies and to take one maybe sensible action: when there's food spotted, eat it.

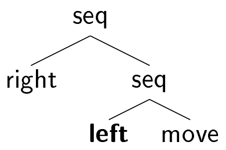
The effects of subtree crossover and ripple crossover in the context of these constraints are expected to be quite different. In the case of subtree crossover, attempting to move a subtree to an illegal context will result in a failure. Consider for example inserting the subtree



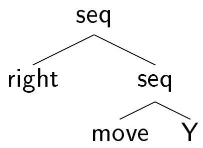

into location X of




thus trying to form the tree




The constraint against turning would make this tree invalid. When using ripple crossover however, the backtracking operator would simply skip the `left` action and the intermediate tree would become




Which is as such invalid. If there is however still genetic material left, possibly in the tail of unexpressed code, the resolution process would use that to fill in the value of Y and then create a valid tree. As the constraints in this grammar only exclude certain pairs of actions or conditions, the backtracking operator is expected to be relatively successful in creating valid offspring.

The effect of using these constraints is dramatic in the ability of ALP to find solutions to the problem. Figure 1 shows the cumulative probability of success over the generations for both ripple crossover and subtree crossover for both logic programs. The success rate for solving the problem goes to 97% for ripple crossover, while subtree crossover's performance goes to 80%. On the standard formulation of the problem, success rates are much lower. We cannot conclude that ripple crossover together with backtracking is better suited to handle the constraints, as the success rates on the standard formulation of the problem already show a significant advantage for ripple crossover. It appears that for this type of problem the more destructive variation applied through ripple crossover has a benefit over the more localized changes of subtree crossover. This is especially pronounced in the later stages of the run, where cumulative success for subtree crossover levels off, while the runs using ripple crossover keep finding solutions.

It is however instructive to examine the failure rate of both methods. These are depicted in Figure 2. Initially the failure rates of both methods peak when the first illegal crossovers occur. Very soon however, the tail

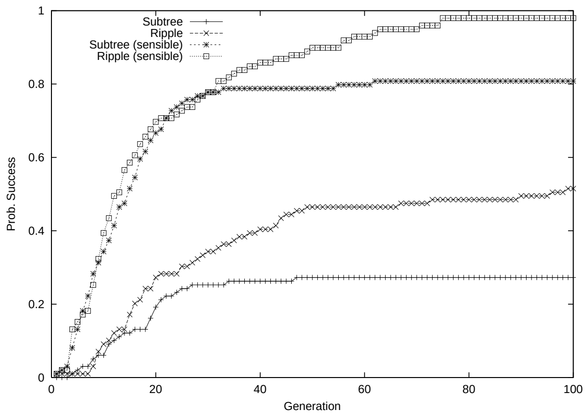



Figure 1: Probability of success in eating 89 pieces of food scattered on the Santa Fe trail for subtree crossover and ripple crossover using the straightforward Program 2.1 and the 'sensible' ant program.

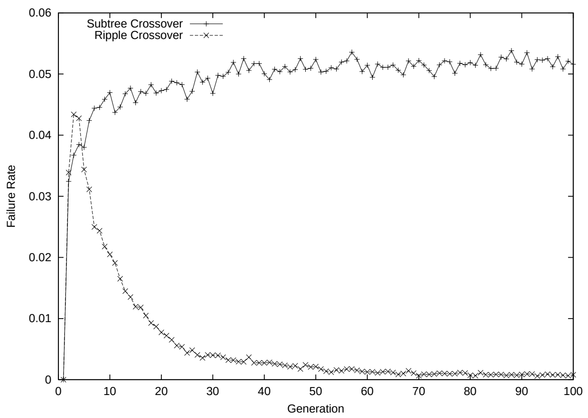



Figure 2: Failure rates of subtree crossover and ripple crossover on the sensible artificial ant problem.

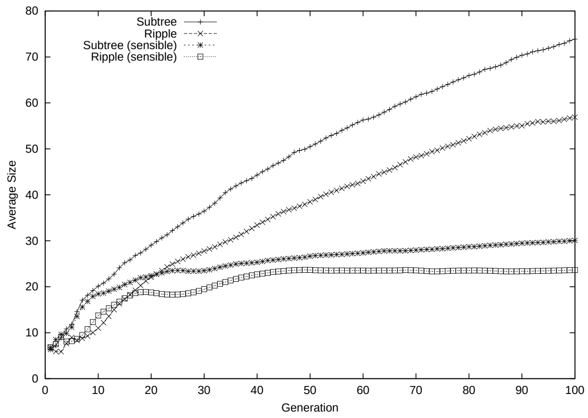



Figure 3: The evolution of the average size of the population on the artificial ant problems.

of unexpressed code begins to form and together with the backtracking mechanism, the failure rate of ripple crossover drops to very low values. The failure rate of subtree crossover on the other hand initially rises and levels off at a rate of 5%. Due to the elitist truncation selection method that is used, invalid individuals only slow down the search. The 5% invalid individuals that are created by subtree crossover would mean that the genetic algorithm effectively processes 475 individuals per generation rather than 500.

It is clear from this experiment that the 'sensible' program helps in solving the problem faster than the straightforward program, regardless of the variation operator that is used. The 'sensible' program uses context-sensitive constraints to introduce some a priori information to exclude redundant subexpressions.

By reducing the search space to exclude certain constructs, an implicit bias towards shorter solutions is introduced. It was estimated in (Langdon and Poli, 1998) that the space of possible ant programs has a high density of solutions of a relative short size. Figure 3 shows the evolution of the size of the population undergoing both crossover operators. The 'sensible' ant constraints induce populations containing programs that are significantly shorter than when using the unconstrained definition of possible ant programs.

# 3 Concluding the Experiment

Although the 'sensible' ant-problem studied here is a toy problem, the way the constraints are introduced point to a more general application: disallowing certain constructs because they are redundant or meaningless. This can often be identified just examining the definition and description of the function set. It proved to be fairly easy to disallow some commonly occurring fragments from the ant programs, the reduced search space allowed to search with great success. Many function sets have known redundancies or allow nonsensical combinations. By using a logic programming approach it is possible to disallow specific constructs. This can be contrasted with a pure syntactical approach (context-free grammars), where it is only possible to express what is allowed.

A possible application of this technique in symbolic regression would be to specifically disallow nesting of some functions. For example, the clause

```
expr(exp(X),C) :- \+ member(no_exp,C),
expr(X, [no_exp|C]).
```

would disallow nested exponentiation functions. In the practice of performing symbolic regression it occurs fairly often that such nested expressions appear, and their presence makes it usually difficult to interpret the equation. When using a subtree crossover using such a grammar, any attempt of inserting a subtree containing an `exp` function underneath an `exp` function that is already present would lead to a failure. Using ripple crossover with backtracking, such an offending operation would simply be skipped and the next instruction would be read from the genotype. Which of the two approaches is the best cannot be answered conclusively at this point.

# References


Deb, K., Agrawal, S., Pratab, A., and Meyarivan, T. (2000). A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In Schoenauer, M., editor, *Proceedings of PPSN-6*, pages 849–858. Springer-Verlag.

Keijzer, M., Babovic, V., Ryan, C., O'Neill, M., and Cattolico, M. (2001a). Adaptive logic programming. In Spector, L., Goodman, E. D., Wu, A., Langdon, W. B., Voigt, H.-M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M. H., and Burke, E., editors, *Proceedings of GECCO 2001*. Morgan Kaufmann.

Keijzer, M., Ryan, C., O'Neill, M., Cattolico, M., and Babovic, V. (2001b). Ripple crossover in genetic programming. In Miller, J. F., Tomassini, M., Lanzi, P. L., Ryan, C., Tettamanzi, A. G. B., and Langdon, W. B., editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 74–86, Lake Como, Italy. Springer-Verlag.

Langdon, W. B. and Poli, R. (1998). Why ants are hard. Technical Report CSRP-98-4, University of Birmingham, School of Computer Science. Presented at GP-98.

O'Neill, M. and Ryan, C. (2001). Grammatical evolution. *IEEE Transaction on Evolutionary Compuation*. Forthcomming.

Ross, B. (1999). Logic based genetic programming with definite clause translation grammars. Technical report, Department of Computer Science, Brock University, Ontario Canada.

Wong, M. L. and Leung, K. S. (1997). Evolutionary program induction directed by logic grammars. *Evolutionary Computation*, 5(2):143–180.