Adaptive Logic Programming

M. Keijzer & V. Babovic
DHI — Water & Environment
Hørsholm, Denmark
{mkk|vmb}@dhi.dk

C. Ryan & M. O'Neill
University of Limerick
Limerick, Ireland
{conor.ryan|michael.oneill}@ul.ie

M. Cattolico
Tiger Mountain Scientific Inc.
Kirkland, WA USA
mike@TigerScience.com

Abstract

A new hybrid of Evolutionary Automatic Programming which employs logic programs is presented. In contrast with tree-based methods, it employs a simple GA on variable length strings containing integers. The strings represent sequences of choices used in the derivation of non-deterministic logic programs. A family of Adaptive Logic Programming systems (ALPs) are proposed and from these, two promising members are examined. A proof of principle of this approach is given by running the system on three problems of increasing grammatical difficulty. Although the initialization routine might need improvement, the system as presented here provides a feasible approach to the induction of solutions in grammatically and logically constrained languages.

1 Introduction

Logic Programming [3] makes a rigorous distinction between the declarative aspect of a computer program and the procedural part. The declarative part defines everything that is "true" in the specific domain, while the procedural part derives instances of these "truths".

The programming language Prolog [16] fills in the procedural aspect by employing a strict depth-first search strategy through the rules (clauses) defined by a logic program. In this paper an alternative search strategy is examined. This employs a variable length genetic algorithm that specifies the choice to make at each choice-point in the derivation of a query. The search strategy operates on logic programs that define simple to more constrained languages. This hybrid of a variable length genetic algorithm operating on logic programs is given the name Adaptive Logic Programming.

The paper is organized by first giving a short introduction of logic programming and Prolog, followed by a description of the non-deterministic modifications we propose. A section with related work of applying genetic programming to logic programs follows in section 4. The system thus described is tested on three problems with increasingly more involved grammatical constraints. A discussion and conclusion finish the paper.

2 Logic Programming

A logic program consists of clauses consisting of a head and a body. In Prolog notation, identifiers starting with an uppercase character are considered to be logic variables, while lowercase characters are atoms or function symbols. The logic program

\[
\begin{align*}
&sym(X), \\
&sym(Y), \\
&sym(X + Y) :- sym(X), sym(Y), \\
&sym(X * Y) :- sym(X), sym(Y),
\end{align*}
\]

defines a single predicate sym. The derivation symbol \(\vdash\) should be read as an inverse implication sign. In predicate logic the third clause can then be interpreted as

\[
\forall X, Y : sym(X) \land sym(Y) \rightarrow sym(X + Y)
\]

The query

\[
\neg sym(X)
\]

can be interpreted as the inquiry $\exists X : \text{sym}(X)^1$ and produces in Prolog the following sequence of solutions:

$$
\begin{align*}
X &= x; \\
X &= y; \\
X &= x + x; \\
X &= x + y; \\
X &= x + (x + x); \\
X &= x + (x + y); \\
X &= x + (x + (x + x)); \\
\ldots
\end{align*}
$$

Extrapolating this sequence it is easy to see that without bounds on the depth or size of the derivation, the depth-first clause selection with backtracking strategy employed in Prolog will never generate an expression that contains the multiplication character. Therefore, while the depth-first selection of clauses may be sound, it is not complete w.r.t. an arbitrary logic program$^2$.

Logic programming is a convenient paradigm for specifying languages and constraints. A predicate can have several attributes and these attributes can be used to constrain the search space. For example, the logic program and query

\[
\text{sym}(x,1). \\
\text{sym}(y,1). \\
\text{sym}(x+y,S) : - \\
\text{sym}(x,S1), \text{sym}(y,S2), S \leftarrow S1+S2+1. \\
\text{sym}(x+y,S) : - \\
\text{sym}(x,S1), \text{sym}(y,S2), S \leftarrow S1+S2+1.
\]

?\- \text{sym}(X, S), S<10.

specifies all expressions of size smaller than 10. With such terse yet powerful descriptiveness, it is therefore no surprise that attribute logic and constraint logic programming are more often than not implemented in Prolog. It is this convenient representation of data or program structures together with constraints that we are trying to exploit in this paper.

Formally, a Logic Programming system is defined by Selected Literal Definite clause resolution (or SLD-resolution for short), and an oracle function that selects the next clause or the next literal$^3$. This oracle function is in Prolog implemented as:

- Select first clause

$^1$Formally the negation of this formula is disproven, thus proving this formula.

$^2$A depth-first strategy is however far more efficient than the breadth-first alternative.

$^3$A literal is a single predicate call in the body of a clause or query. In the query above, \text{sym}(X,S) and $S < 10$ are literals.

- Select first literal
- Backtrack on failure

3 Grammatical Evolution and Logic Programming

Grammatical Evolution$^4$ aims at inducing arbitrary computer programs based on a context-free specification of the language. It employs a variable length integer representation that specifies a sequence of choices made in the context-free grammar to generate an expression. Due to the specific representation of a sequence of choices, no type information needs to be maintained in the evolving strings and no custom mutation and crossover operators need to be designed. The variable length one-point crossover employed in GE was shown to have an elegant interpretation in closed grammars in$^5$.

In this paper we similarly use a sequence of choices as the base representation, but rather than choosing between the production rules of a context-free grammar, they are used to make a choice between clauses in a logic program. The sequence of choices thus represents one part of the selection function operating together with SLD-resolution on the logic program. Furthermore, backtracking is implemented in the system together with an alternative strategy on failure: restarting the original query.

As an example of the mapping process, consider the grammar defined above in Section 2, and an evolutionary induced sequence of choices$^6$.$^7$.$^8$. The derivation of an instance then proceeds as follows:

?\- \text{sym(X).} \\
?\- \text{sym(X1), sym(X2). [X1 + X2]/X} 2 \\
?\- \text{sym(y), sym(X2). [y/X1]} 1 \\
?\- \text{sym(X3), sym(X4). [X3 * X4]/X2} 3 \\
?\- \text{sym(x), sym(X4). [x/X3]} 0 \\
?\- \text{sym(y).} \\
\text{[y/X4]} 1

Applying all bindings made, this produces the symbolic expression $y + x \ast y$. The values from the sequence of choices are in this example conveniently chosen to lie between 0 and 3 inclusively in practice a number encountered in the genotype can be higher than the number of choices present. The choice will then be taken modulo the number of available choices.

In this example, the depth-first clause selection of Prolog is replaced by a guided selection where choices are drawn from the genotype. The first unresolved literal is still chosen to be the first to derive. It is possible to replace this with guided selection as well, be it in the
A logic program is thus used as a formal specification of the language, the sequence of choices is used to steer the resolution process and a small external program is used to evaluate the expressions generated. See Figure 1 for the typical flow of information. The scope of the system are then logic programs where there is an abundance of solutions that satisfy the constraints, which are subsequently evaluated for performance on a problem domain.

3.1 Backtracking

In ALP-1, at every step in the derivation process, a list is maintained of clauses that are not tried yet. When a query fails at a certain point, the selection function will be asked to pick a new choice out of the remaining clauses. This choice is removed and when all are exhausted, the branch reports failure to the previous level where this procedure starts again.

ALP-4 does not use backtracking; on failure it will restart the original, top-level, query, while the reading continues from where it left off.

If the sequence runs out of choices, i.e., the end of the genotype is reached, the derivation is cut off and the individual gets the worst performance value available. This will be labelled a failure.

3.2 Initialization

Initialization is performed by doing a random walk through the grammar, maintaining the choices made; backtracking on failure (ALP-1) or restarting (ALP-4). After a successful derivation is found, the shortest, non-backtracking path to the complete derivation is calculated. An occurrence check is performed and if the path is not present in the current population, a new individual is initialized with this shortest non-backtracking path. Individuals in the initial population will thus consist solely of non-backtracking derivations to sentences.

Typically a depth limit is employed.

3.3 Performance Evaluation

Performance is typically evaluated in a special module, written in a compiled language such as C. This program walks through the tree structure and evaluates each node. This is however not necessary if the fitness can be readily evaluated in the logic program itself. The query investigated typically has the form: find that derivation for sentence(X) such that fitness.eval(X,F) returns the maximal or minimal F.
3.4 Variational Operators

Crossover is implemented as a simple variable length string crossover. Two independent random points are chosen in the strings and strings starting at these points are swapped. The two points are chosen within the *expressed* code of a string — code that is used in the derivation.

The effects of the crossover in this case is quite different from that of subtree crossover. This is because the derivation tree is created in a pre-order fashion, i.e., the left-most literal of a goal is always mapped to completion before the rest of the goal is processed.

Crossover operates on the linear structure, and single point crossover thus divides an individual into a partially mapped tree, and a stack of choices. In general, all subtrees to the right of the crossover site are removed, as in Figure 2, leaving multiple vacant sites on the derivation tree. These sites are said to be *ripple* up from the crossover site.

An integer in the genome is said to be *intrinsically polymorphic*, meaning that it can be interpreted (or re-interpreted) by any node in a derivation tree in whatever context. By adding codons from the other parent to the incomplete derivation tree in Figure 2, the sites vacated by the crossover event are again filled with new subnodes of the appropriate type.

In contrast with subtree crossover, the percentage of genetic material exchanged is on average 50% and it has been shown that this crossover is quite effective in exploring the search space of possible programs as it is less susceptible to premature convergence [7].

Although many mutations can be defined on a string of integers, the one used here simply replaces a randomly selected integer from the string with a randomly drawn integer lower than $2^{31}$.

3.5 Special Predicates

All Prolog built-in clauses such as assignment (is/2) are evaluated in Prolog directly. This is done as often such clauses are deterministic and depend on the Prolog depth-first search strategy. Also calls to libraries etc., are evaluated directly.

A special predicate ext_int/2 is employed that, when encountered in the derivation, binds the first argument with an integer drawn from the genotype modulo the second argument (which therefore needs to be grounded). Using this technique, floating point constants can be specified as part of the logic program. The floating point grammar used in this paper is:

\[
\begin{align*}
fp_{\text{unsigned}}(X) & : - \\
& \text{ext}_{\text{int}}(\text{Num}, 256), \\
& \text{ext}_{\text{int}}(\text{Denom}, 256), \\
& X \text{ is } \text{Num} / (\text{Denom} + 1) . \\
fp_{\text{unsigned}}(X) & : - \\
& fp_{\text{unsigned}}(\text{First}), \\
& fp_{\text{unsigned}}(\text{Second}), \\
& X \text{ is } \text{First} \ast \text{Second}. \\
fp(X) & : - \\
& \text{ext}_{\text{int}}(\text{S}, 2), \\
& \text{Sign is } (S-0.5) \ast 2 , \\
& fp_{\text{unsigned}}(Y), \\
& X \text{ is } \text{Sign} \ast Y .
\end{align*}
\]

There is nothing particularly innovative or clever about this program. Although it specifies up to machine precision floating points, it can only model rational numbers for which the numerator and denominator are factors of primes smaller than 256. It does show however, how intricate calculations can be made a part of the language. A call to \(fp/1\) will bind the argument to a floating point value instead of an expression. Future versions of ALPs will undoubtedly support floating point numbers that evolve together with the list of choices, so that specialized mutation operators can be used.

4 Related Work

Wong and Leung [17] hybridized inductive logic programming and genetic programming in their system LOGENPRO. The representation that is being manipulated by the genetic operators consist of derivation trees. LOGENPRO first applies a preprocessing step that transforms a logic grammar (a Definite Clause Grammar) into a logic program. Apart from expressions in the specified language, this logic program also
produces a symbolic representation of the derivation tree. This derivation tree is subsequently manipu-
lated by the genetic operators. Some fairly intricate crossover and mutation operators are used which, to-
gether with semantic validation, ensure that the re-
sulting derivation tree specifies a valid instantiation of the logic grammar. Because the logic program is able
to parse derivation trees, semantic verification reduces
to checking whether Prolog accepts the derivation tree.

Ross [15] describes a similar system that uses
Definite Clause Translation Grammars. This representation is
also translated into a logic program that is able to
parse and generate derivation trees in the language
defined by the grammar. The crossover described in
[15] seems to only use type information contained in
the predicate names and arity at the heads of the
clauses and swaps derivation subtrees that contain
the same head. A semantic verification (running the Pro-
log program on the derivation tree), is subsequently
performed.

Even for typed crossovers, semantic validation is neces-
sary as the body of a clause can introduce additional
constraints, not related to the type but to the actual
values found in the derivation. An additional problem
for strongly typed crossover occurs when the number of
distinct types grows. As the operator will only swap
subtrees that have the same type, every type needs
to be present multiple times with different derivations
in the population to make the operator swap some-
thing other than identical trees. If a specific type dis-
appears from a population, or only has a single dis-

tinct instance, the system has to rely on mutation to
re-introduce instances. Every additional type or con-
straint thus partitions the search space further and
thereby restricts the crossover.

Yet another problem with subtree crossover is that it
will process an increasingly smaller percentage of ge-
netic material as the size of the individuals grows [1],
while the crossover employed here will always swap on
average half of the genetic material [7].

In contrast with the systems described above, the
ALPs do not use an explicit representation of the
derivation tree, thus being time and memory ef-
cient. In the systems described above, every step in
the derivation process is recorded in a node to-
gether with the bindings that are made, effectively
doubling the size of an expression tree. In ALPs, no
pre-processing step is necessary; it works on logic pro-
grams directly. Also no bookkeeping is necessary when
trying crossovers and mutations. The downside of this
is that the ALPs can generate invalid individuals, i.e.,
strings of choices that have no valid derivation. How-

ever, such a failed derivation is equivalent with a failed
semantic validation in the systems described above.
The rate at which this happens is ultimately bound to
the language and constraints used.

5 Proof of Principle

The system outlined above was implemented using
SWI-Prolog4, mainly because of the two-way C API
that it implements. A steady-state genetic algorithm
using a tournament size of 5 was implemented using
the evolutionary objects library5. Crossover and mu-
tation were applied with rates 0.9 and 0.1 respectively.
What follows are three experiments with grammars of
increasing degrees of complexity. The purpose of these
experiments is to present a proof of the principle that a
variable length GA can indeed be used to successfully
induce sentences in both easy and difficult languages.

The experiments were run for 100 generations using
both ALP-1 and ALP-4. For the symbolic regression
and Santa Fe trail problem, 100 runs were performed,
the results on the sediment transport experiment
are reported on the basis of 500 runs. As a baseline
test, for each problem, 10 million random individuals
were generated using the initialization procedure from
ALP-1 (denoted by ALP-1R). Also 10 million individu-
als were generated by Prolog (ALP-0). As Prolog was
not able to produce a single correct individual for any
of the problems, these results are further omitted. For
all methods, the same depth limit was set.

5.1 Symbolic Regression: \(0.3\sin(2\pi x)\)

From this function 100 equally spaced points in the
interval \([-1,1]\) were generated. This problem has been
studied in [6] with data in the range \([0,1]\). For the
experiments a population size of 1000 was used. A
success was determined to be a root mean squared er-
ror less than 0.01.

5.2 An Artificial Ant on the Santa Fe Tail

The artificial ant problem has been studied intensively
in [10] for a closed grammar. Here a context free gram-
mar is employed like in [7].

A population of size 500 was used. The best perfor-
ance achievable was 89 food pellets eaten.

---

4http://www.swi.psy.uva.nl/projects/SWI-Prolog
5http://www.sourceforge.net/projects/codev
5.3 Units of Measurement: Sediment Transport

The units of measurement problem used here has been studied previously in [2]. In contrast with [2] the system is constrained to generate only dimensionally correct equations. Another approach for this class of problems is studied in [14] where a context free grammar is generated that models a subset of the language of units of measurement.

The desired output for this problem is a dimensionless quantity, a concentration. Two experiments were performed, one where the desired output is given and one experiment where no desired output is given. These are denoted in Table 2 as Sed1 and Sed2 respectively. The second experiment thus seeks for a dimensionally consistent formulation stated in any units. It is quite common for empirical equations to multiply the resulting equation with a constant stated in some units to obtain an equation stated in the desired units of measurement6; this is usually a residual coefficient that tries to describe some unmodelled phenomena.

The parameters were set at the same values as the symbolic regression problem above. A successful run was determined by comparing the error produced to that of a benchmark model, which was an equation induced by a scientist [5]. Because success rates were low, 500 runs were performed for this problem.

6 Results

For all problems, solutions were found, Table 2 summarizes the results. Although the differences between ALP-1 and ALP-4 are not significant (α = 0.05) on the symbolic regression problem7 and the Santa-Fe problem, the failure of ALP-4 to find any solutions on any of the sediment transport problems clearly shows the need for backtracking. The sediment transport problem involves non-trivial constraints, and inspection of the expressions produced by ALP-4 showed that it got very quickly trapped into derivations of shallow depth, often converging on a single constant. It is hypothesized that the use of backtracking allows the genotype to specify a particular start of the derivation process, relying on backtracking as a local search operator to find feasible solutions.

Confidence intervals were calculated around the 99% computational effort statistic proposed by Koza ([8] p. 194). The first fifty percent of the runs were used to find the generation that maximized the effort statistic, the results reported were subsequently calculated on the latter (independent) half of the runs. As the confidence interval calculated for the sediment transportation problem included a 0% success rate, the upper bound of the confidence interval is infinite. This is to be expected, as the success predicate demanded that the system should improve upon an equation proposed by an expert in the field of sediment transport. Interestingly enough, for the second sediment transportation problem (that allows dimensionally consistent equations that do not produce the desired dimensionless output), the success rate is significantly higher. This illustrates the dangers of providing too much bias to a weak search algorithm such as ALP.

The confidence intervals were calculated in response to a question posed by Miller [11] on the value of this statistic on experiments with a low success rate. Table 2 shows that, indeed, for a low success rate such as 1.6%, the statistic can only give a lower (highly optimistic) bound on the number of individuals to process. It also shows that the statistic is highly volatile even for moderate success rates. For the Santa-Fe problem that has an overall success rate of 37%, the width of the confidence interval (i.e., the uncertainty around the statistic) is nearly as large as the value of the computational effort itself. The confidence intervals clearly show that a straightforward comparison of computational effort, even differing in an order of magnitude, is not possible.

Figure 3 shows the average fail ratio for ALP-1. As

<table>
<thead>
<tr>
<th></th>
<th>ALP-1</th>
<th>ALP-4</th>
<th>ALP-IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>S. R.</td>
<td>4253(9%)</td>
<td>550(6%)</td>
<td>inf(0%)</td>
</tr>
<tr>
<td></td>
<td>[2351, 11642]</td>
<td>[2924, 16868]</td>
<td>inf(0%)</td>
</tr>
<tr>
<td>S. F.</td>
<td>185(37%)</td>
<td>284(28%)</td>
<td>1229(3.6% - 4%)</td>
</tr>
<tr>
<td></td>
<td>[124, 305]</td>
<td>[172, 584]</td>
<td>[852, 2302]</td>
</tr>
<tr>
<td>Sed1</td>
<td>100(7.6%)</td>
<td>inf(0%)</td>
<td>inf(0%)</td>
</tr>
<tr>
<td></td>
<td>[36, 3629]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sed2</td>
<td>1610(26%)</td>
<td>inf(0%)</td>
<td>inf(0%)</td>
</tr>
<tr>
<td></td>
<td>[130, 2054]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Computational Effort divided by 1000 for solving the three problems. Overall success rate in round brackets. Numbers in square brackets denote 95% confidence intervals around the effort statistic calculated above. Confidence intervals are calculated with resampling statistics, using a bootstrap sample of 10000. The success rates are calculated on the final (100th) generation.
the initial generation includes only valid individuals, the ratio is zero. It is clear from the figure that this initial population is not well adapted to produce valid individuals. For the less constraint problems, the percentage of failed derivations quickly drops to low values. For the problems involving units of measurement, the level of failed derivations does not drop that quickly; even after 100 generations, more than one in five crossover and/or mutation events results in a failed derivation.

Although it might seem that the crossover and mutation employed here are very destructive, and might even lead to the hasty conclusion that a strongly typed crossover is necessary, this is in our opinion not warranted. The high fail rates are a symptom of the highly constrained nature of this search space. A strongly typed crossover would have this same problem; it would either obscure it by only swapping identical subtrees, or by a high failure rate in the semantic validation. Figure 4 shows that despite this high failure rate, the system is still able to perform significant optimization. It would however be instructive to see how well a strongly typed system would fair on this problem.

7 Discussion

The system presented here is the first prototype for evolving sentences in languages with constraints. It has proven to be able to optimize all the problems described here, including a difficult language such as the units of measurement grammar.

The initialization procedure as is described here does not provide an optimal starting point for the ALP systems. The initialization procedure consists of non-backtracking points to derivations, with no unexpressed code. It is an avenue of future research to find a better initialization procedure. However, the highly exploitative nature of the crossover used here, enables the system to overcome this and even with a non-optimal starting point, it is able to find competitive solutions to the problems presented to it.

The main benefit of the ALPs system in contrast with strongly typed genetic programming systems is that the variational operators do not depend as heavily on the grammar that is used. A strongly typed crossover is constrained to search in the space of available types in the population, thus having a strong macro-mutation flavor [1]. The ALP systems, borrowing the mapping process from Grammatical Evolution, is in principle not thus constrained. New instances of types can be created during the run.

Although this paper has focussed on expression induction, due to the general nature of logic programs, we also expect to be able to perform optimization on transformational problems [12], as well as on constructive (embryonic) problems [4, 9].

8 Conclusion

An implementation and proof of principle is given for an adaptive logic programming system called ALPs.
GENETIC PROGRAMMING

It modifies the standard Prolog clause selection to a selection strategy that is guided by a variable length genotype. The system was tested on three different problems of increasing difficulty and was able to produce solutions to these problems.

Although backtracking did not seem necessary for the simpler grammars, it made a significant difference in the difficult grammar of units of measurement.

Acknowledgements

The first two authors would like to acknowledge the Danish Technical Research Council (STVF) for partly funding Talent Project 9800463 entitled "Data to Knowledge - D2K" http://www.d2k.dk

References


