Workshop on Approaches and Applications of Inductive Programming (AAIP)

to be held on August 7th 2005 in conjunction with the 22nd International Conference on Machine Learning (ICML 2005) in Bonn, Germany.

Organized by Emanuel Kitzelmann, Roland Olsson, Ute Schmid

Contents

Foreword **5**

Program Committee **7**

Invited Talks **8**

Stephen Muggleton:
Learning the Time Complexity of Logic Programs 9

Jürgen Schmidhuber:
How to Learn a Program: Optimal Universal Learners & Goedel Machines 11

Fritz Wysotzki:
Development of Inductive Synthesis of Functional Programs 13

Full Papers **15**

Emanuel Kitzelmann, Ute Schmid:
An Explanation Based Generalization Approach to Inductive Synthesis of Functional Programs 15

Oleg Monakhov, Emilia Monakhova:
Synthesis of Scientific Algorithms based on Evolutionary Computation and Templates 29

A. Passerini, P. Frasconi, L. De Raedt:
Kernels on Prolog Proof Trees: Statistical Learning in the ILP Setting . . 37

M. R. K. Krishna Rao:
Learning Recursive Prolog Programs with Local Variables from Examples 51

Work in Progress **59**

Ramiro Aguilar, Luis Alonso, Vivian López, María N. Moreno:
Incremental discovery of sequential patterns for grammatical inference . . 59

Palem GopalaKrishna:
Data-dependencies and Learning in Artificial Systems 69

Author Index **81**

Foreword

The ICML workshop AAIP 2005 – "Approaches and Applications of Inductive Programming" – was held at the 22nd International Conference on Machine Learning (ICML 2005) in Bonn, Germany on 7th of August 2005. The main goal of AAIP was to bring together researchers from different areas of machine learning who are especially interested in the inductive synthesis of programs from input/output examples.

Automatic induction of programs from input/output examples is an active area of research since the sixties and of interest for machine learning research as well as for automated software engineering. In the early days of inductive programming research there were proposed several approaches to the synthesis of Lisp programs from examples or traces. Due to only limited progress, interest decreased in the mid-eighties and research focused on inductive logic programming (ILP) instead. Although ILP proved to be a powerful approach to learning relational concepts, applications to learning of recursive clauses had only moderate success. Learning recursive programs from examples is also investigated in genetic programming and other forms of evolutionary computation. Here again, functional programs are learned from sets of positive examples together with an output evaluation function which specify the desired input/output behavior of the program to be learned. A fourth approach is learning recursive programs in the context of grammar inference.

Currently, there is no single prominent approach to inductive program synthesis. Instead, research is scattered over the different approaches. Nevertheless, inductive programming is a research topic of crucial interest for machine learning and artificial intelligence in general. The ability to generalize a program – containing control structures as recursion or loops – from examples is a challenging problem which calls for approaches going beyond the requirements of algorithms for concept learning. Pushing research forward in this area can give important insights in the nature and complexity of learning as well as enlarging the field of possible applications.

Typical areas of application where learning of programs or recursive rules are called for, are first in the domain of software engineering where structural learning, software assistants and software agents can help to relieve programmers from routine tasks, give programming support for endusers, or support of novice programmers and programming tutor systems. Further areas of application are language learning, learning recursive control rules for AI-planning, learning recursive concepts in web-mining or for data-format transformations.

Today research on program induction is scattered over different communities. The AAIP 2005 workshop brought together researchers from these different communities with the common interest on induction of general programs regarding theory, methodology and applications. The three invited speakers represent inductive program synthesis research in the areas of classical functional synthesis (Fritz Wysotzki, TU Berlin, Germany), inductive logic programming (Stephen Muggleton, Imperial College London, UK), and

optimal universal reinforcement learning (Jürgen Schmidhuber, IDSIA, Manno-Lugano, Switzerland). The four full papers and two work in progress reports are distributed over all areas mentioned above, that is, they address inductive functional synthesis, inductive logic programming, evolutionary computation, and grammatical inference.

In our opinion, getting acquainted with other approaches to this problem, their relative merits and limits, will strengthen research in inductive programming by cross-fertilization. We hope, that for the machine learning community at large the challenging topic of program induction which is currently rather neglected, might come in the focus of interest.

We want to thank the all members of the program committee for their support in promoting the workshop and for reviewing submitted papers and we thank the ICML organizers, especially the workshop chair Hendrik Blockeel for technical support.

We are proud to present you the proceedings of the first workshop ever striving to cover all areas of program induction!

Emanuel Kitzelmann
(Dept. of Information Systems and Applied Computer Science, Otto-Friedrich-University Bamberg, Germany)

Roland Olsson
(Faculty of Computer Science, Østfold College, Norway)

Ute Schmid
(Dept. of Information Systems and Applied Computer Science, Otto-Friedrich-University Bamberg, Germany)

Program Committee

Pierre Flener
: Computing Science Division, Department of Information Technology, Uppsala University, Sweden

Colin de la Higuera
: EURISE, Université de Saint-Etienne, France

Emanuel Kitzelmann (co-organizer)
: Dept. of Information Systems and Applied Computer Science, Bamberg University, Germany

Steffen Lange
: Dept. of Computer Science, University of Applied Sciences Darmstadt, Germany
++49-6151-16-8441, s.lange@fbi.fh-darmstadt.de
http://www.fbi.fh-darmstadt.de/s̃lange/

Stephen Muggleton
: Computational Bioinformatics Laboratory, Department of Computing, Imperial College,London, UK

Roland Olsson (co-organizer)
: Faculty of Computer Science, Østfold college, Halden, Norway

M.Jose Ramírez
: Departamento de Sistemas Informáticos y Computación (DSIC), Universidad Politécnica de Valencia, Spain

Ute Schmid (co-organizer)
: Dept. of Information Systems and Applied Computer Science, Bamberg University, Germany

Fritz Wysotzki
: Dept. of Electrical Engineering and Computer Science, Technical University Berlin, Germany

Learning the Time Complexity of Logic Programs

Stephen Muggleton SHM@DOC.IC.AC.UK
Computational Bioinformatics Laboratory
Department of Computing
Imperial College, London, UK

One of the key difficulties in machine learning recursive logic programs is associated with the testing of examples. Inefficient hypotheses, though usually of least interest to the learner, take more time to test. Almost by definition, efficient learners require a bias towards low-complexity hypotheses. However, it is unclear how such a bias can be implemented. To these ends, in this presentation we address the problem of recognising inefficient logic programs. To the author's knowledge, this problem has not been considered previously in the literature. A successful approach to recognition of inefficient logic programs should bring the prospects of effective machine learning of recursive logic programs closer. In general the problem of recognising the time complexity of an arbitrary program is incomputable since halting is undecidable. However, partial solutions cannot be ruled out. In this presentation we provide an initial investigation of the problem based on developing a framework for machine learning higher-order patterns associated with various complexity orders.

How to Learn a Program: Optimal Universal Learners & Goedel Machines

Jürgen Schmidhuber JUERGEN@IDSIA.CH
IDSIA
Manno-Lugano, Switzerland

Rational embedded agents should try to maximize future expected reward. In general, this requires learning an algorithm that uses internal states to remember relevant past sensory inputs. Most machine learning algorithms, however, just learn reactive behaviors, where output depends only on the current input. Is there an optimal way of learning non-reactive behaviors in general, unknown environments? Our new insights affirm that the answer is yes. I will discuss both theoretical results on optimal universal reinforcement learning & Goedel Machines, and mention applications of the recent Optimal Ordered Problem Solver.

Links related to this talk:

Cogbotlab:
`http://www.idsia.ch/~juergen/cogbotlab.html`

Universal learning machines / Goedel Machines:
`http://www.idsia.ch/~juergen/unilearn.html`
`http://www.idsia.ch/~juergen/goedelmachine.html`

Optimal Ordered Problem Solver:
`http://www.idsia.ch/~juergen/oops.html`

Development of Inductive Synthesis of Functional Programs

Fritz Wysotzki WYSOTZKI@CS.TU-BERLIN.DE
Technische Universität Berlin

Inductive program synthesis is yet in the state of fundamental research. The task consists of inductive construction of a program from pairs of given input-output examples or instantiated initial parts of the intended program using specific methods of generalisation. Well known is the field of Inductive Logic Programming (ILP) which is concerned with the induction of logical programs. The pioneering work in synthesis of functional programs was done by Summers 1977 using the programming language LISP. Summers approach was restricted to structural list problems i.e. his algorithms depend on the structure of a list but not of its content. After Summers other work on synthesis of functional programs was done using LISP, too. The main topic of this lecture will be the synthesis of functional programs in an abstract formalism, i.e. without using a special programming language. The basis are mainly theoretical investigations and programmed realisations of the algorithms which have been performed at the TU Berlin over many years. The lecture starts with some historical remarks, after that the theoretical basis is introduced. It is a term algebra including a special non-strict term (corresponding to the IF-THEN-ELSE) for realizing tests. A functional program is represented by a system of equations the left hand side of which are function variables (Ònames of subprogramsÓ) with parameters (Òformal parametersÓ), the right hand sides are terms, which may contain the function variables and there is a special term representing the Ómain programÓ. This system is called Recursive Program Scheme (RPS, Courcelle and Nivat 1978). It can be ÒsolvedÓ syntactically by an ordered sequence of finite terms (KLEENE-sequence), approximating the fixpoint solution which is an infinite term. This corresponds to unfolding the RPS.The induction principle works as follows: If one has a finite example term (i.e. with instantiated variables) one can try to Òexplain itÓ as being an element of a KLEENE-sequence of an RPS. Extrapolation with introduction of variables as generalisation principle and folding gives a hypothetical RPS which explains the given example term. The reliability of the hypothesis depends on the position of the given term in the KLEENE-sequence. One of the main problems which will be discussed is the detection and induction of subprograms in the example term by finding an appropriate segmentation. Another problem is how to get the example term. Two domain dependent approaches will be discussed: the integration of (mutually excluding) production rules and planning. In the latter case from a problem solving graph a shortest path tree (Universal Plan) is constructed. In the second step by a generalisation procedure similar to subsumption used in ILP a goal hierarchy is computed. This goal tree is then transformed into an initial program (program tree) which can be used as a basis for inductive construction of a RPS mentioned above. The main principles of our approach to the inductive construction of functional programs will be demonstrated by examples.

An Explanation Based Generalization Approach to Inductive Synthesis of Functional Programs

Emanuel Kitzelmann EMANUEL.KITZELMANN@WIAI.UNI-BAMBERG.DE
Ute Schmid UTE.SCHMID@WIAI.UNI-BAMBERG.DE
Department of Information Systems and Applied Computer Science, Otto-Friedrich-University, Bamberg

Abstract

We describe an approach to the inductive synthesis of recursive equations from input/output-examples which is based on the classical two-step approach to induction of functional Lisp programs of Summers (1977). In a first step, I/O-examples are rewritten to traces which explain the outputs given the respective inputs based on a datatype theory. This traces can be integrated into one conditional expression which represents a non-recursive program. In a second step, this initial program term is generalized into recursive equations by searching for syntactical regularities in the term. Our approach extends the classical work in several aspects. The most important extensions are that we are able to induce a *set* of recursive equations in one synthesizing step, the equations may contain more than one recursive call, and additionally needed parameters are automatically introduced.

1. Introduction

Automatic induction of recursive programs from input/output-examples (I/O-examples) is an active area of research since the sixties and of interest for AI research as well as for software engineering (Lowry & McCarthy, 1991; Flener & Partridge, 2001). In the seventies and eighties, there were several approaches to the synthesis of Lisp programs from examples or traces (see Biermann et al., 1984 for an overview). The most influential approach was developed by Summers (1977), who put inductive synthesis on a firm theoretical foundation.

Summers' early approach is an explanation based generalization (EBG) approach, thus it widely relies on algorithmic processes and only partially on search: In a first step, traces—steps of computations executed from a program to yield an output from a particular input—and predicates for distinguishing the inputs are calculated for each I/O-pair. Construction of traces, which are *terms* in the classical functional approaches, relies on knowledge of the inductive datatype of the inputs and outputs. That is, traces *explain* the outputs based on a theory of the used datatype given the respective inputs. The classical approaches for synthesizing Lisp-programs used the general Lisp datatype *S-expression*. By integrating traces and predicates into a conditional expression a non-recursive program explaining all I/O-examples is constructed as result of the first synthesis step. In a second step, regularities are searched for between the traces and predicates respectively. Found regularities are then inductively generalized and expressed in form of the resulting recursive program.

The programs synthesized by Summers' system contain exactly one recursive function, possibly along with one constant term calling the recursive function. Furthermore, all synthesizable functions make use of a small fixed set of Lisp-primitives, particularly of exactly one predicate function, *atom*, which tests whether its argument is an atom, e.g., the empty list. The latter implies two things: First, that Summers' system is restricted to induce programs for *structural* problems on S-expressions. That means, that execution of induced programs depends only on the structure of the input S-expression, but never on the semantics of the atoms contained in it. E.g., *reversing* a list is a structural problem, yet not *sorting* a list. The second implication is, that calculation of the traces is a deterministic and algorithmic process, i.e. does not rely on search and heuristics.

Due to only limited progress regarding the class of programs which could be inferred by functional synthesis, interest decreased in the mid-eighties. There was a renewed interest of inductive program synthesis in the field of inductive logic programming (ILP) (Flener & Yilmaz, 1999; Muggleton & De Raedt, 1994), in genetic programming and other forms of evolutionary computation (Olsson, 1995) which rely heavily on search.

We here present an EBG approach which is based on the methodologies proposed by Summers (1977). We regard the functional two-step approach as worthwhile for the following reasons: First, algebraic datatypes provide guid-

ance in expressing the outputs in terms of the inputs as first synthesis step. Second, it enables a seperate and thereby specialized handling of a knowledge dependent part and a purely syntactic driven part of program synthesis. Third, both using algebraic datatypes and seperating a knowledge-dependent from a syntactic driven part enables a more accurate utilization of search than in ILP or evolutionary programming. Fourth, the two-step approach using algebraic datatypes provides a systematic way to introduce auxiliary recursive equations if necessary.

Our approach extends Summers in several important aspects, such that we overcome fundamental restrictions of the classical approaches to induction of Lisp programs: First, we are able to induce a *set* of recursive equations in one synthesizing step, second, the equations may contain more than one recursive call, and third, additionally needed parameters are automatically introduced. Furthermore, our generalization step is domain-independent, in particular independent from a certain programming language. It takes as input a first-order term over an arbitrary signature and generalizes it to a recursive program scheme, that is, a set of recursive equations over that signature. Hence it can be used as learning component in all domains which can represent their objects as recursive program schemes and provide a system for solving the first synthesis step. E.g., we use the generalization algorithm for learning recursive control rules for AI planning problems (cp. Schmid & Wysotzki, 2000; Wysotzki & Schmid, 2001).

2. Central Concepts and an Example

The three central objects dealt with by our system are (1) sets of *I/O-examples* specifying the algorithm to be induced, (2) *initial (program) terms* explaining the I/O-examples, and (3) *recursive program schemes (RPSs)* representing the induced algorithms. Their functional role in our two-step synthesis approach is shown in Fig. 1.

An example for I/O-examples is given in Tab. 1. The examples specify the *lasts* function which takes a list of lists as input and yields a list of the last elements of the lists as output. In the first synthesis step, an initial term is constructed from these examples. An initial term is a term respecting an arbitrary first-order signature extended by the special constant symbol Ω, meaning the *undefined* value and directing generalization in the second synthesis step. Suitably interpreted, an initial term evaluates to the specified output when its variable is instantiated with a particular input of the example set and to *undefined* for all other inputs.

Tab. 2 gives an example of an initial term. It shows the result of applying the first synthesis step to the I/O-examples for the *lasts* function as shown in Tab. 1. *if* means the 3ary non-strict function which returns the value of its second parameter if its first parameter evaluates to *true* and otherwise returns the value of its third parameter; *empty* is a predicate which tests, whether its argument is the empty list; *hd* and *tl* yield the first element and the rest of a list respectively; *cons* constructs a list from one element and a list; and [] denotes the empty list.

Table 1. I/O-examples for *lasts*

$$
\begin{array}{rcl}
[] & \mapsto & [], \\
[[a]] & \mapsto & [a], \\
[[a,b]] & \mapsto & [b], \\
[[a,b,c]] & \mapsto & [c], \\
[[a,b,c,d]] & \mapsto & [d], \\
[[a],[b]] & \mapsto & [a,b], \\
[[a],[b,c]] & \mapsto & [a,c], \\
[[a,b],[c],[d]] & \mapsto & [b,c,d], \\
[[a,b],[c,d],[e,f]] & \mapsto & [b,d,f], \\
[[a],[b],[c],[d]] & \mapsto & [a,b,c,d]
\end{array}
$$

Table 2. Initial term for *lasts*

```
if(empty(x), [],
 cons(
  hd(
   if(empty(tl(hd(x))), hd(x),
    if(empty(tl(tl(hd(x)))), tl(hd(x)),
     if(empty(tl(tl(tl(hd(x))))), tl(tl(hd(x))),
      Ω)))),
  if(empty(tl(x)), [],
   cons(
    hd(
     if(empty(tl(hd(tl(x)))), hd(tl(x)),
      Ω)),
    if(empty(tl(tl(x))), [],
     cons(
      hd(
       if(empty(tl(hd(tl(tl(x))))), hd(tl(tl(x))),
        Ω)),
      if(empty(tl(tl(tl(x)))), [],
       Ω)))))))
```

Calculation of initial terms relies on knowledge of the datatypes of the example inputs and outputs. For our exemplary *lasts* program inputs and outputs are lists. Lists are uniquely constructed by means of the empty list [] and the constructor *cons*. Furthermore they are uniquely decomposed by the functions *hd* and *tl*. That allows to calculate a unique term which expresses an example output in terms of the input. For example, consider the third I/O-example from Tab. 1: If x denotes the input $[[a,b]]$, then the term $cons(hd(tl(hd(x))),[])$ expresses the specified output $[b]$ in terms of the input. Such traces are constructed for each I/O-pair. The overall concept for integrating the resulting traces into one initial term is to go through all traces in parallel position by position. If the same function symbol is contained at the current position in all traces, then it is in-

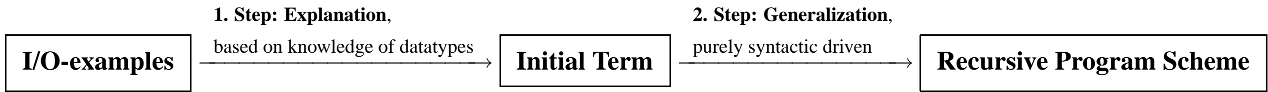

Figure 1. Two synthesis steps

troduced to the initial term at this position. If at least two traces differ at the current position, then it is introduced an *if*-expression. Therefore a predicate function is calculated to discriminate the inputs according to the different traces. Construction of the initial term proceeds from the discriminated inputs and traces for the second and third branch of the *if*-tree respectively. We describe the calculation of initial terms from I/O-examples, i.e. the first synthesis step, in Sec. 4.

In the second synthesis step, initial ground terms are generalized to a recursive program scheme. Initial terms are considered as *(incomplete) unfoldings* of an RPS which is to be induced by generalization. An RPS is a set of recursive equations whose left-hand-sides consist of the names of the equations followed by their parameter lists and whose right-hand-sides consist of terms over the signature from the initial terms, the set of the equation names, and the parameters of the equations. One equation is distinguished to be the main one. An example is given in Tab. 3. This RPS, suitably interpreted, computes the *lasts* function as described above and specified by the examples in Tab. 1. It

Table 3. Recursive Program Scheme for *lasts*

$$\begin{aligned}\mathrm{lasts}(x) &= \mathrm{if}(\mathrm{empty}(x), [],\\ &\qquad \mathrm{cons}(\mathrm{hd}(\mathrm{last}(\mathrm{hd}(x))), \mathrm{lasts}(\mathrm{tl}(x))))\\ \mathrm{last}(x) &= \mathrm{if}(\mathrm{empty}(\mathrm{tl}(x)), x, \mathrm{last}(\mathrm{tl}(x)))\end{aligned}$$

results from applying the second synthesis step to the initial term shown in Tab. 2. Note that it is a *generalization* from the initial term in that it not merely computes the *lasts* function for the example inputs but for input-lists of arbitrary length containing lists of arbitrary length.

The second synthesis step does not depend on domain knowledge. The meaning of the function symbols is irrelevant, because the generalization is completely driven by detecting syntactical regularities in the initial terms. To understand the link between initial terms and RPSs induced from them, we consider the process of incrementally unfolding an RPS. *Unfolding* of an RPS is a (non-deterministic and possibly infinite) rewriting process which starts with the instantiated head of the main equation of an RPS and which repeatedly rewrites a term by substituting any instantiated head of an equation in the term with either the equally instantiated body or with the special symbol Ω. Unfolding stops, when all heads of recursive equations in the term are rewritten to Ω, i.e., the term contains no rewritable head any more. Consider the *last* equation from the RPS shown in Tab. 3 and the initial instantiation $\{x \mapsto [a,b,c]\}$. We start with the instantiated head $last([a,b,c])$ and rewrite it to the term:

$$\mathrm{if}(\mathrm{empty}(\mathrm{tl}([a,b,c])), [a,b,c], \mathrm{last}(\mathrm{tl}([a,b,c])))$$

This term contains the head of the *last* equation instantiated with $\{x \mapsto tl([a,b,c])\}$. When we rewrite this head again with the equally instantiated body we obtain:

$$\begin{aligned}&\mathrm{if}(\mathrm{empty}(\mathrm{tl}([a,b,c])), [a,b,c],\\ &\quad \mathrm{if}(\mathrm{empty}(\mathrm{tl}(\mathrm{tl}([a,b,c]))), \mathrm{tl}([a,b,c]),\\ &\qquad \mathrm{last}(\mathrm{tl}(\mathrm{tl}([a,b,c])))))\end{aligned}$$

This term now contains the head of the equation instantiated with $\{x \mapsto tl(tl([a,b,c]))\}$. We rewrite it once again with the instantiated body and then replace the head in the resulting term with Ω and obtain:

$$\begin{aligned}&\mathrm{if}(\mathrm{empty}(\mathrm{tl}([a,b,c])), [a,b,c],\\ &\quad \mathrm{if}(\mathrm{empty}(\mathrm{tl}(\mathrm{tl}([a,b,c]))), \mathrm{tl}([a,b,c]),\\ &\qquad \mathrm{if}(\mathrm{empty}(\mathrm{tl}(\mathrm{tl}(\mathrm{tl}([a,b,c])))), \mathrm{tl}(\mathrm{tl}([a,b,c])), \Omega)))\end{aligned}$$

The resulting *finite* term of a *finite* unfolding process is also called *unfolding*. Unfoldings of RPSs contain regularities if the heads of the recursive equations are more than once rewritten with its bodies before they are rewritten with Ωs. The second synthesis step is based on detecting such regularities in the initial terms.

We describe the generalization of initial terms to RPSs in the following section. The reason why we first describe the second synthesis step and only afterwards the first synthesis step is, that the latter is governed by the goal of constructing a term which can be generalized in the second step. Therefore, for understanding the first step, it is necessary to know the connection between initial terms and RPSs as established in the second step.

3. Generalizing an Initial Term to an RPS

Since our generalization algorithm exploits the relation between an RPS and its unfoldings, in the following we will first introduce the basic terminology for terms, substitutions, and term rewriting as for example presented in Dershowitz and Jouanaud (1990). Then we will present definitions for RPSs and the relation between RPSs and their

unfoldings. The set of all possible RPSs constitutes the hypothesis language for our induction algorithm. Some restrictions on this general hypothesis language are introduced and finally, the componentes of the generalization algorithm are described.

3.1. Preliminaries

We denote the set of natural numbers starting with 0 by $\mathbb{N}$ and the natural numbers greater 0 by $\mathbb{N}_+$. A signature Σ is a set of (function) symbols with $\alpha : \Sigma \to \mathbb{N}$ giving the arity of a symbol. We write T_Σ for the set of ground terms, i.e. terms without variables, over Σ and $T_\Sigma(X)$ for the set of terms over Σ and a set of variables X. We write $T_{\Sigma,\Omega}$ for the set of ground terms—called partial ground terms—constructed over $\Sigma \cup \{\Omega\}$, where Ω is a special constant symbol denoting the *undefined* value. Furthermore, we write $T_{\Sigma,\Omega}(X)$ for the set of partial terms constructed over $\Sigma \cup \{\Omega\}$ and variables X. With $T^\infty_{\Sigma,\Omega}(X)$ we denote the set of inifinite partial terms over Σ and variables X. Over the sets $T_{\Sigma,\Omega}$, $T_{\Sigma,\Omega}(X)$ and $T^\infty_{\Sigma,\Omega}(X)$ a complete partial order (CPO) $\leq$ is defined by: a) $\Omega \leq t$ for all $t \in T_{\Sigma,\Omega}, T_{\Sigma,\Omega}(X), T^\infty_{\Sigma,\Omega}(X)$ and b) $f(t_1,\ldots,t_n) \leq f(t'_1,\ldots,t'_n)$ iff $t_i \leq t'_i$ for all $i \in [1;n]$.

Terms can uniquely be expressed as labeled trees: If a term is a constant symbol or a variable, then the corresponding tree consists of only one node labeled by the constant symbol or variable. If a term has the form $f(t_1,\ldots,t_n)$, then the root node of the corresponding tree is labeled with f and contains from left to right the subtrees corresponding to $t_1,\ldots,t_n$. We use the terms *tree* and *term* as synonyms. A *position* of a term/tree is a sequence of positive natural numbers, i.e. an element from $\mathbb{N}^*_+$. The set of positions of a term t, denoted $\mathbf{pos}(t)$, contains the empty sequence ε and the position iu, if the term has the form $t = f(t_1,\ldots,t_n)$ and u is a position from $\mathbf{pos}(t_i), i \in [1;n]$. Each position of a term uniquely denotes one subterm. We write $t|_u$ for denoting that subterm which is determined as follows: (a) $t|_\varepsilon = t$, (b) if $t = f(t_1,\ldots,t_n)$ and u is a position in t_i, then $t|_{iu} = t_i|_u$, $i \in [1;n]$. We say that position u is smaller than position u', $u \leq u'$, if u is a prefix of u'. If u is a position of term t and $u' \leq u$, then u' is a position of t. For a term t and a position u, $\mathbf{node}(t,u)$ denotes the fixed symbol $f \in \Sigma$, if $t|_u = f(t_1,\ldots,t_n)$ or $t|_u = f$ respectively. The set of all positions at which a fixed symbol f appears in a term is denoted by $\mathbf{pos}(t,f)$. The replacement of a subterm $t|_u$ by a term s in a term t at position u is written as $t[u \leftarrow s]$. Let U denote a set of positions in a term t. Then $t[U \leftarrow s]$ denotes the replacement of all subterms $t|_u$ with $u \in U$ by s in t.

A *substitution* σ is a mapping from variables to terms. Substitutions are naturally continued to mappings from terms to terms by $\sigma(f(t_1,\ldots,t_n)) = f(\sigma(t_1),\ldots,\sigma(t_n))$. Substitutions are written in postfix notation, i.e. we write $t\sigma$ instead of $\sigma(t)$. Substitutions $\beta : X \to T_\Sigma$ from variables to *ground* terms are called (variable) *instantiations*. A term p is called *pattern* of a term t, iff $t = p\sigma$ for a substitution σ. A pattern p of a term t is called *trivial*, iff p is a variable and *non-trivial* otherwise. We write $t \leq_s p$ iff p is a pattern of t and $t <_s p$ iff additionally holds, that p and t can not be unified by variable renaming only.

A *term rewriting system (TRS)* over Σ and X is a set of pairs of terms $\mathscr{R} \subseteq T_\Sigma(X) \times T_\Sigma(X)$. The elements (l,r) of $\mathscr{R}$ are called rewrite rules and are written $l \to r$. A term t' can be derived in one rewrite step from a term t using $\mathscr{R}$ ($t \to_{\mathscr{R}} t'$), if there exists a position u in t, a rule $l \to r \in \mathscr{R}$, and a substitution $\sigma : X \to T_\Sigma(X)$, such that (a) $t|_u = l\sigma$ and (b) $t' = t[u \leftarrow r\sigma]$. $\mathscr{R}$ implies a rewrite relation $\to_{\mathscr{R}} \subseteq T_\Sigma(X) \times T_\Sigma(X)$ with $(t,t') \in \to_{\mathscr{R}}$ if $t \to_{\mathscr{R}} t'$.

3.2. Recursive Program Schemes

Definition 1 (Recursive Program Scheme). Given a signature Σ, a set of *function variables* $\Phi = \{G_1,\ldots,G_n\}$ for a natural number $n > 0$ with $\Sigma \cap \Phi = \emptyset$ and arity $\alpha(G_i) > 0$ for all $i \in [1;n]$, a natural number $m \in [1;n]$, and a set of equations

$$\mathscr{G} = \left\{ \begin{array}{l} G_1(x_1,\ldots,x_{\alpha(G_1)}) = t_1, \\ \vdots \\ G_n(x_1,\ldots,x_{\alpha(G_n)}) = t_n \end{array} \right\}$$

where the t_i are terms with respect to the signature $\Sigma \cup \Phi$ and the variables $x_1,\ldots,x_{\alpha(G_i)}$, $\mathscr{S} = (\mathscr{G},m)$ is an RPS. $G_m(x_1,\ldots,x_{\alpha(G_m)}) = t_m$ is called the main equation of $\mathscr{S}$.

The function variables in Φ are called *names* of the equations, the left-hand-sides are called *heads*, the right-hand-sides *bodies* of the equations. For the *lasts* RPS shown in Tab. 3 holds: $\Sigma = \{if, empty, cons, hd, tl, []\}$, $\Phi = \{G_1, G_2\}$ with $G_1 = lasts$ and $G_2 = last$, and $m = 1$. $\mathscr{G}$ is the set of the two equations.

We can identify a TRS with an RPS $\mathscr{S} = (\mathscr{G},m)$:

Definition 2 (TRS implied by an RPS). Let be $\mathscr{S} = (\mathscr{G},m)$ an RPS over Σ, Φ and X, and Ω the bottom symbol in $T_{\Sigma,\Omega}(X)$. The equations in $\mathscr{G}$ constitute rules $\mathscr{R}_{\mathscr{S}} = \{G_i(x_1,\ldots,x_{\alpha(G_i)}) \to t_i \mid i \in [1;n]\}$ of a term rewriting system. The system additionally contains rules $\mathscr{R}_\Omega = \{G_i(x_1,\ldots,x_{\alpha(G_i)}) \to \Omega \mid i \in [1;n],\ G_i \text{ is } \textit{recursive}\}$.

The standard interpretation of an RPS, called *free interpretation*, is defined as the supremum in $T^\infty_{\Sigma,\Omega}(X)$ of the set of all terms in $T_{\Sigma,\Omega}(X)$ which can be derived by the implied TRS from the head of the main equation. Two RPSs are called *equivalent*, iff they have the same free interpretation, i.e. if they compute the same function for every interpretation of the symbols in Σ. Terms in $T_{\Sigma,\Omega}$ which can be derived by the *instantiated* head of the main equation

regarding some instantiation $\beta : X \to T_\Sigma$ are called *unfoldings* of an RPS relative to β. Note, that terms derived from RPSs are partial and do not contain function variables, i.e. all heads of the equations are eventually rewritten by Ωs.

The goal of the generalization step is to find an RPS which *explains* a set of initial terms, i.e. to find an RPS such that the initial terms are unfoldings of that RPS. We denote initial terms by $\bar{t}$ and a set of initial terms by $\mathscr{I}$. We liberalize $\mathscr{I}$ such that it may include *incomplete* unfoldings. Incomplete unfoldings are unfoldings, where some subtrees containing Ωs are replaced by Ωs.

We need to define four further concepts, namely *recursion positions* which are positions in the equation bodies where recursive calls appear, *substitution terms* which are the argument terms in recursive calls, *unfolding positions* which are positions in unfoldings at which the heads of the equations are rewritten with their bodies, and finally *parameter instantiations in unfoldings* which are subterms of unfoldings resulting from the initial parameter instantiation and the substitution terms:

Definition 3 (Recursion Positions and Substitution Terms). Let $G(x_1,\ldots,x_{\alpha(G)}) = t$ with parameters $X = \{x_1,\ldots,x_{\alpha(G)}\}$ be a recursive equation. The set of *recursion positions* of G is given by $R = \mathbf{pos}(t,G)$. Each recursive call of G at position $r \in R$ in t implies substitutions $\sigma_r : X \to T_\Sigma(X) : x_j \mapsto t|_{rj}$ for all $j \in [1;\alpha(G)]$ for the parameters in X. We call the terms $t|_{rj}$ *substitution terms* of G.

For equation *lasts* of the *lasts* RPS (Tab. 3) holds $R = \{32\}$ and $x\sigma_{32} = tl(x)$. For equation *last* holds $R = \{3\}$ and $x\sigma_3 = tl(x)$.

Now consider an unfolding process of a recursive equation and the positions at which rewrite steps are applied in the intermediate terms. The first rewriting is applied at root-position ε, since we start with the instantiated head of the equation which is completely rewritten with the instantiated body. In the instantiated body, rewrites occur at recursion positions R. Assume that on recursion position $r \in R$ the instance of the head is rewritten with an instance of the body. Then, relative to the resulting *subtree* at position r, rewrites occur again at recursion positions, e.g. at position $r' \in R$. Relative to the entire term these latter rewrites occur therefore at compositions of position r and recursion positions, e.g. at position rr' and so on. We call the infinite set of positions at which rewrites can occur in the intermediate terms within an unfolding of a recursive equation *unfolding positions*. They are determined by the recursion positions as follows:

Definition 4 (Unfolding Positions). Let be R the recursion positions of a recursive equation G. The set of *unfolding positions* U of G is defined as the smallest set of positions which contains the position ε and, if $u \in U$ and $r \in R$, the position ur.

The unfolding positions of equation *lasts* of the *lasts* RPS are $\{32, 3232, 323232, \ldots\}$.

Now we look at the variable instantiations occuring during unfolding a recursive equation. Recall the unfolding process of the *last* equation (see Tab. 3) described at the end of Sec. 2. The initial instantiation was $\beta_\varepsilon = \beta = \{x \mapsto [a,b,c]\}$, thus in the body of the equation (replaced for the instantiated head as result of the first rewrite step), its variable is instantiated with this initial instantiation. Due to the substitution term $tl(x)$, the variable of the head in this body is instantiated with $\beta_3 = \sigma_3\,\beta_\varepsilon = \{x \mapsto tl([a,b,c])\}$, i.e. the variable in the body replaced for this instantiated head is instantiated with $\sigma_3\,\beta_\varepsilon$. A further rewriting step implies the instantiation $\beta_{33} = \sigma_3\,\sigma_3\,\beta_\varepsilon = \sigma_3\,\beta_3 = \{x \mapsto tl(tl([a,b,c]))\}$ and so on. We index the instantiations occuring during unfolding with the unfolding positions at which the particular instantiated heads were placed. They are determined by the substitutions implied by recursive calls and an initial instantiation as follows:

Definition 5 (Instantiations in Unfoldings). Let be $G(x_1,\ldots,x_{\alpha(G)}) = t$ a recursive equation with parameters $X = \{x_1,\ldots,x_{\alpha(G)}\}$, R and U the recursion positions and unfolding positions of G resp., σ_r the substitutions implied by the recursive call of G at position $r \in R$, and $\beta : X \to T_\Sigma$ an initial instantiation. Then a family of instantiations indexed over U is defined as $\beta_\varepsilon = \beta$ and $\beta_{ur} = \sigma_r\,\beta_u$ for $u \in U, r \in R$.

3.3. Restrictions and the Generalization Problem

An RPS which can be induced from initial terms is restricted in the following way: First, it contains no mutual recursive equations, second, there are no calls of recursive equations within calls of recursive equations (no nested recursive calls). The first restriction is not a semantical restriction, since each mutual recursive program can be transformed to an equivalent (regarding a particular algebra) non-mutual recursive program. Yet it is a *syntactical* restriction, since unfoldings of mutual RPSs can not be generalized using our approach. A restriction similar to the second one was stated by Rao (2004). He names TRSs complying with such a restriction *flat* TRSs.

Inferred RPSs conform to the following syntactical characteristics: First, all equations, potentially except of the main equation, are recursive. The main equation may be recursive as well, but, as only equation, it is not required to be recursive. Second, inferred RPSs are minimal, in that (i) each equation is directly or indirectly (by means of other equations) called from the main equation, and (ii) no parameter of any equation can be omitted without changing the free

interpretation. RPSs complying with the stated restrictions and characteristics are called *minimal, non-mutual, flat recursive program schemes*.

There might be several RPSs which explain an initial term $\bar{t}$, but have different free interpretations. For example, Ω is an unfolding of *every* RPS with a recursive main equation. Therefore, an important question is which RPS will be induced. Summers (1977) required that recurrence relations hold at least over three succeeding traces and predicates to justify a generalization. A similar requirement would be that induced RPSs explain the initial terms *recurrently*, meaning that $\mathcal{I}$ contains at least one term $\bar{t}$ which can be derived from an unfolding process, in which each recursive equation had to be rewritten at least three times with its body. We use a slightly different requirement: One characteristic of minimal RPSs is, that if at least one substitution term is replaced by another, then the resulting RPS has a different free interpretation. We call this characteristic *substitution uniqueness*. Thus, it is sensible to require that induced RPSs are substitution unique regarding the initial terms, i.e. that *if* some substitution term is changed, then the resulting RPS *no longer* explains the initial terms. It holds, that a minimal RPS explains a set of initial trees recurrently, if it explains it substitution uniquely.

Thus the problem of generalizing a set of initial terms $\mathcal{I}$ to an RPS is to find an RPS which explains $\mathcal{I}$ and which is substitution unique regarding $\mathcal{I}$.

3.4. Solving the Generalization Problem

We will not state the generalization algorithm in detail in this section but we will describe the underlying concepts and the algorithm in a more informal manner. For this section and its subsections we use the term *body* of an equation for terms which are strictly speaking *incomplete* bodies: They contain only the name of the equation instead of complete recursive calls including substitution terms at recursion positions. For example, we refer to the term $\mathit{if}(\mathit{empty}(x), [], \mathit{cons}(\mathit{hd}(\mathit{last}(\mathit{hd}(x))), \mathit{lasts}))$ as the body for equation *lasts* of the *lasts* RPS (see Tab. 3). The reason is, that we infer the complete body in two steps: First the term which we name body in this context, second the substitution terms for the recursive calls.

Generalization of a set of initial terms to an RPS is done in three successive steps, namely *segmentation of the terms*, *construction of equation bodies* and *calculation of substitution terms*. These three generalization steps are organized in a divide-and-conquer algorithm, where backtracking can occur to the divide-phase. *Segmentation* constitutes the divide-phase which proceeds top-down through the initial terms. Within this phase recursion positions (see Def. 3) and positions indicating further recursive equations are searched for each induced equation. The latter set of positions is called *subscheme positions* (see Def. 6 below). Found recursion positions imply unfolding positions (see Def. 4). As result of the divide-phase the initial terms are divided into several parts by the subscheme positions, such that—roughly speaking—each particular part is assumed to be an unfolding of *one* recursive equation. Furthermore, the particular parts are segmented by the unfolding positions, such that—roughly speaking—each segment is assumed to be the result of *one* unfolding step of the respective recursive equation.

Consider the initial tree in Fig. 2, it represents the initial term for *lasts*, shown in Tab. 2. The curved lines on the path to the rightmost Ω divide the tree into three segments which correspond to unfolding steps of the main equation, i.e. equation *lasts*. The short broad lines denote three subtrees which are—except of their root *hd*—unfoldings of the *last* equation. The curved lines within these subtrees divide each subtree into segments, such that each segment correspond to one unfolding step of the *last* equation.

When the initial trees are segmented, calculation of equation bodies and of substitution terms follows within the conquer-phase. These two steps proceed bottom-up through the divided initial trees and reduce the trees during this process. The effect is, that bodies and substitution terms for each equation are calculated from trees which are unfoldings of *only* the currently induced equation and hence, each segment in these trees is an instantiation of the body of the currently induced equation. E.g., for the *lasts* tree shown in Fig. 2, a body and substitution terms are first calculated from the three subtrees, i.e. for the *last* equation. Since there are no further recursive equations called by the *last* equation—i.e. the segments of the three subtrees contain themselves no subtrees which are unfoldings of further equations—each segment is an instantiation of the body of the *last* equation. When this equation is completely inferred, the three subtrees are replaced by suitable instantiations of the head of the inferred *last* equation. The resulting reduced tree is an unfolding of merely *one* recursive equation, the *lasts* equation. The three segments in this reduced tree—indicated by the curved lines on the path to the rightmost Ω—are instantiations of the body of the searched for *lasts* equation. From this reduced tree, body and substitution terms for the *lasts* equation are induced and the RPS is completely induced.

Segmentations are searched for, whereas calculation of bodies and substitution terms are algorithmic. Construction of bodies always succeeds, whereas calculation of substitution terms—such that the inferred RPS explains the initial terms—may fail. Thus, an inferred RPS can be seen as the result of a search through a hypothesis space where the hypotheses are segmentations (divide-phase), and a *constructive* goal test, including construction of bodies and calcu-

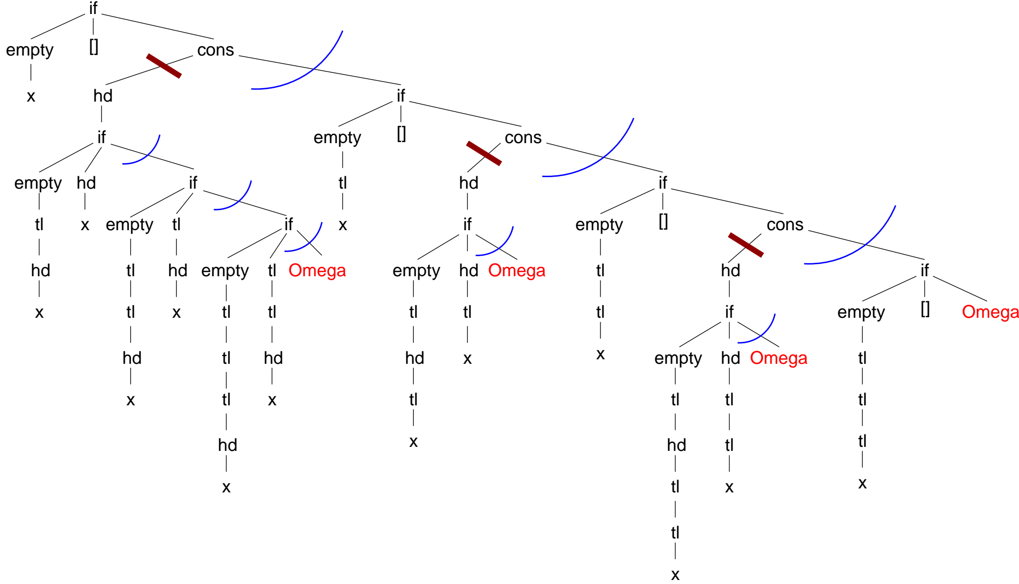

Figure 2. Initial Tree for *lasts*

lation of substitution terms (conquer-phase), which tests, whether the completely inferred RPS explains the initial terms (and is substitution unique regarding them). In the following we describe each step in more detail:

3.4.1. Segmentation

When induction of an RPS from a set of initial trees $\mathscr{I}$ starts, the hypothesis is, that there exists an RPS with a recursive main equation which explains $\mathscr{I}$. First, recursion and subscheme positions for the hypothetical main equation G_m are searched for.

Definition 6 (Subscheme Positions). Subscheme positions are all smallest positions in the body of a recursive equation G which denote subterms, in which calls of *further* recursive equations from the RPS appear, but *no recursive* call of equation G.

E.g., the only subscheme position of equation *lasts* of the *lasts* RPS (Tab. 3) is $u = 31$. A priori, only particular positions from the initial trees come into question as recursion and subscheme positions, namely those which belong to a path leading from the root to an Ω. The reason is, that eventually *each* head of a recursive equation at any unfolding position in an intermediate term while unfolding this equation is rewritten with an Ω:

Lemma 1 (Recursion and Subscheme Positions imply Ωs). *Let $\bar{t} \in T_{\Sigma,\Omega}$ be an (incomplete) unfolding of an RPS $\mathscr{S} = (\mathscr{G}, m)$ with a recursive main equation G_m. Let R, U and S be the sets of recursion, unfolding and subscheme positions of G_m respectively. Then for all $u \in U \cap \mathbf{pos}(\bar{t})$ holds:*

1. $\mathbf{pos}(\bar{t}|_u, \Omega) \neq \emptyset$
2. $\forall s \in S :$ *if* $us \in \mathbf{pos}(\bar{t})$ *then* $\mathbf{pos}(\bar{t}|_{us}, \Omega) \neq \emptyset$

It is not very difficult to see that this lemma holds. For a lack of space we do not give the proof here. It can be found in (Kitzelmann, 2003) where Lem. 1 and Lem. 2 are proven as one lemma. Knowing Lem. 1, before search starts, the initial trees can be reduced to their *skeletons* which are terms resulting from replacing subtrees without Ωs with variables.

Definition 7 (Skeleton). The skeleton of a term $t \in T_{\Sigma,\Omega}(X)$, written $\mathbf{skeleton}(t)$ is the minimal pattern of t for which holds $\mathbf{pos}(t, \Omega) = \mathbf{pos}(\mathbf{skeleton}(t), \Omega)$.

For example, consider the subtree indicated by the leftmost short broad line of the tree in Fig. 2. Omitting the root *hd*, it is an unfolding of the *last* equation of the *lasts* RPS shown in Tab. 3. Its skeleton is the *substantially* reduced term $if(x_1, x_2, if(x_3, x_4, if(x_5, x_6, \Omega)))$. Search for recursion and subscheme positions is done on the skeletons of the original initial trees. Thereby the hypothesis space is substantially narrowed without restricting the hypothesis language, since only those hypotheses are ruled out which are a priori known to fail the goal test.

Ωs are not only implied by recursion and subscheme positions, but also imply Ωs recursion and subscheme positions since Ωs in unfoldings result *only* from rewriting an instantiated head of a recursive equation in a term with an Ω:

Lemma 2 (Ωs imply recursion and subscheme positions). *Let $\bar{t} \in T_{\Sigma,\Omega}$ be an (incomplete) unfolding of an RPS $\mathscr{S} = (\mathscr{G}, m)$ with a recursive main equation G_m. Let R, U*

and S be the sets of recursion, unfolding and subscheme positions of G_m respectively. Then for all $v \in \mathbf{pos}(\bar{t},\Omega)$ hold

- *It exists an $u \in U \cap \mathbf{pos}(\bar{t}), r \in R$ with $u \leq v < ur$ or*
- *it exists an $u \in U \cap \mathbf{pos}(\bar{t}), s \in S$ with $us \leq v$.*

Proof: in (Kitzelmann, 2003).

From the definition of subscheme positions and the previous lemma follows, that subscheme positions are determined, if a set of recursion positions has been fixed. Lem. 1 restricts the set of positions which come into question as recursion and subscheme positions. Lem. 2 together with characteristics from subscheme positions suggests to organize the search as a search for recursion positions with a depending parallel calculation of subscheme positions. When hypothetical recursion, unfolding, and subscheme positions are determined they are checked regarding the labels in the initial trees on pathes leading to Ωs. The nodes between one unfolding position and its successors in unfoldings result from the same body (with different instantiations). Since variable instantiations only occur in subtrees at positions *not* belonging to pathes leading to Ωs, for each unfolding position the nodes between it and its successors *are necessarily equal*:

Lemma 3 (Valid Segmentation). *Let $\bar{t} \in T_{\Sigma,\Omega}$ be an unfolding of an RPS $\mathscr{S} = (\mathscr{G}, m)$ with a recursive main equation G_m. Then it exists a term $\check{t}_G \in T_{\Sigma,\Omega}(X)$ with $\mathbf{pos}(\check{t}_G,\Omega) = R \cup S$ such that for all $u \in U \cap \mathbf{pos}(\bar{t})$ hold: $\check{t}_G \leq_\Omega \bar{t}|_u$ where $\leq_\Omega$ is defined as (a) $\Omega \leq_\Omega t$ if $\mathbf{pos}(t,\Omega) \neq \emptyset$, (b) $x \leq_\Omega t$ if $x \in X$ and $\mathbf{pos}(t,\Omega) = \emptyset$, and (c) $f(t_1,\ldots,t_n) \leq_\Omega f(t'_1,\ldots,t'_n)$ if $t_i \leq_\Omega t'_i$ for all $i \in [1;n]$.*

Proof: in (Kitzelmann, 2003).

This lemma has to be slightly extended, if one allows for initial trees which are incomplete unfoldings. Lem. 3 states the requirements to assumed recursion and subscheme positions which can be assured at segmentation time. They are necessary for an RPS which explains the initial terms, yet not sufficient to assure, that an RPS complying with them exists which explains the initial trees. That is, later a backtrack can occur to search for other sets of recursion and subscheme positions. If found recursion and subscheme positions *R* and *S* comply with the stated requirements, we call the pair (*R*, *S*) a *valid segmentation*.

In our implemented system the search for recursion positions is organized as a greedy search through the space of sets of positions in the skeletons of the initial trees. When a valid segmentation has been found, compositions of unfolding and subscheme positions denote subtrees in the initial trees assumed to be unfoldings of further recursive equations. Segmentation proceeds recursively on each set of (sub)trees denoted by compositions of unfolding positions and one subscheme position $s \in S$. We denote such a set of initial (sub)trees $\mathscr{I}_s$.

3.4.2. Construction of Equation Bodies

Construction of each equation body starts with a set of initial trees $\mathscr{I}$ for which at segmentation time a valid segmentation (*R*, *S*) has been found, and an already inferred RPS for each subscheme position $s \in S$ which explains the subtrees $\mathscr{I}_s$. These subtrees of the trees in $\mathscr{I}$ are replaced by the suitably instantiated heads or respectively bodies of the main equations of the already inferred RPSs. For example, consider the initial tree for *lasts* shown in Fig. 2. When calculation of a body for the main equation *lasts* starts from this tree, an RPS containing only the *last* equation which explains all three subtrees indicated by the short broad lines has already been inferred. The initial tree is reduced by replacing these three subtrees by suitable instantiations of the head of the *last* equation. We denote the set of reduced initial trees also with $\mathscr{I}$ and its elements also with $\bar{t}$. By reducing the initial trees based on already inferred recursive equations, the problem of inducing a set of recursive equations is reduced to the problem of inducing merely *one* recursive equation (where the recursion positions are already known from segmentation).

An equation body is induced from the segments of an initial tree which is assumed to be an unfolding of one recursive equation.

Definition 8 (Segments). Let be $\bar{t}$ an initial tree, *R* a set of (hypothetical) recursion positions and *U* the corresponding set of unfolding positions. The set of *complete segments* of $\bar{t}$ is defined as: $\{\bar{t}|_u[R \leftarrow G] \mid u \in U \cap \mathbf{pos}(\bar{t}), R \subset \bar{t}|_u\}$

For example, consider the subtree indicated by the leftmost short broad line of the initial tree in Fig. 2 without its root *hd*. It is an unfolding of the *last* equation as stated in Tab. 3. When the only recursion position 3 has been found it can be splitted into three segments, indicated by the curved lines:

1. $\mathrm{if}(\mathrm{empty}(\mathrm{tl}(\mathrm{hd}(x))), \mathrm{hd}(x), G)$
2. $\mathrm{if}(\mathrm{empty}(\mathrm{tl}(\mathrm{tl}(\mathrm{hd}(x)))), \mathrm{tl}(\mathrm{hd}(x)), G)$
3. $\mathrm{if}(\mathrm{empty}(\mathrm{tl}(\mathrm{tl}(\mathrm{tl}(\mathrm{hd}(x))))), \mathrm{tl}(\mathrm{tl}(\mathrm{hd}(x))), G)$

Expressed according to segments, the fact of a repetitive pattern between unfolding positions (see Lem. 3) becomes the fact, that the sequences of nodes between the root and each *G* are equal for each segment. Each segment is an instantiation of the body of the currently induced equation. In general, the body of an equation contains other nodes among those between its root and the recursive calls. These further nodes are also equal in each segment. Differences in segments of unfoldings of a recursive equation can *only*

result from different instantiations of the variables of the body. Thus, for inducing the body of an equation from segments, we assume each position in the segments which is equally labeled in *all* segments as belonging to the body of the assumed equation, but each position which is variably labeled in at least two segments as belonging to the instantiation of a variable. This assumption can be seen as an inductive bias since it might occur, that also positions which are equal over all segments belong to a variable instantiation. Nevertheless it holds, that if an RPS exists which explains a set of initial trees, then it also exists an RPS which explains the initial trees and is constructed based on the stated assumption. Based on the stated assumption, the body of the equation to be induced is determined by the segments and defined as follows:

Definition 9 (Valid Body). Given a set of reduced initial trees, the most specific maximal pattern of all segments of all the trees is called *valid body* and denoted $\hat{t}_G$.

The maximal pattern of a set of terms can be calculated by first order anti-unification (Plotkin, 1969).

Calculating a valid body regarding the three segments enumerated above results in the term $if(empty(tl(x)), x, G)$. The *different* subterms of the segments are assumed to be instantiations of the parameters in the calculated valid body. Since each segment corresponds to one unique unfolding position, instantiations of parameters in unfoldings as defined in Def. 5 are now given. E.g., from the three segments enumerated above we obtain:

1. $\beta_\varepsilon(x) = \mathrm{hd}(x)$
2. $\beta_3(x) = \mathrm{tl}(\mathrm{hd}(x))$
3. $\beta_{33}(x) = \mathrm{tl}(\mathrm{tl}(\mathrm{hd}(x)))$

3.4.3. Inducing Substitution Terms

Induction of substitution terms for a recursive equation starts on a set of reduced initial trees which are assumed to be unfoldings of one recursive equation, an already inferred (incomplete) equation body which contains only a G at recursion positions, and variable instantiations in unfoldings according to Def. 5. The goal is to complete each occurence of G to a recursive call including substitution terms for the parameters of the recursive equation.

The following lemma follows from Def. 5 and states characteristics of parameter instantiations in unfoldings more detailed. It characterizes the instantiations in unfoldings against the substitution terms of a recursive equation considering each single position in them.

Lemma 4 (Instantiations in Unfoldings). *Let be $G(x_1, \ldots, x_{\alpha(G)}) = t$ a recursive equation with parameters $X = \{x_1, \ldots, x_{\alpha(G)}\}$, recursion positions R and unfolding positions U, $\beta : X \to T_\Sigma$ an instantiation, σ_r substitution terms for each $r \in R$ and β_u instantiations as defined in Def. 5 for each $u \in U$. Then for all $i, j \in [1; \alpha(G)]$ and positions v hold:*

1. *If $(x_i \sigma_r)|_v = x_j$ then for all $u \in U$ hold $(x_i \beta_{ur})|_v = x_j \beta_u$.*
2. *If $(x_i \sigma_r)|_v = f((x_i \sigma_r)|_{v1}, \ldots, (x_i \sigma_r)|_{vn}), f \in \Sigma, \alpha(f) = n$ then for all $u \in U$ hold $\mathbf{node}(x_i \beta_{ur}, v) = f$.*

We can read the implications stated in the lemma in the inverted direction and thus we get almost immediately an algorithm to calculate the substitution terms of the searched for equation from the known instantiations in unfoldings.

One interesting case is the following: Suppose a recursive equation, in which at least one of its parameters *only* occurs within a recursive call in its body, for example the equation $G(x, y, z) = if(zerop(x), y, +(x, G(prev(x), z, succ(y))))$ in which this is the case for parameter z[1]. For such a variable no instantiations in unfoldings are given when induction of substitution terms starts. Also such variables are not contained in the (incomplete) valid equation body. Our generalizer introduces them each time, when none of the both implications of Lem. 4 hold. Then it is assumed, that the currently induced substitution term contains such a "hidden" variable at the current position. Based on this assumption the instantiations in unfoldings of the hidden variable can be calculated and the inference of subtitution terms for it proceeds as described for the other parameters.

When substitution terms have been found, it has to be checked, whether they are substitution unique with regard to the reduced initial terms. This can be done for each substitution term that was found separately.

3.4.4. Inducing an RPS

We have to consider two further points: The first point is that segmentation presupposes the initial trees to be explainable by an RPS with a *recursive* main equation. Yet in Sec. 3.3 we characterized the inferable RPSs as liberal in this point, i.e. that also RPSs with a non-recursive main equation are inferable. In such a case, the initial trees contain a constant (not repetitive) part at the root such that no recursion positions can be found for these trees (as for example the three subtrees indicated by the short broad lines in Fig. 2 which contain the constant root *hd*). In this case, the root node of the trees is assumed to belong to the body of a non-recursive main equation and induction of RPSs recursively proceeds at each subtree of the root nodes.

The second point is that RPSs explaining the subtrees which are assumed to be unfoldings of further recursive equations at segmentation time are already inferred. Based

[1] A practical example is the Tower of Hanoi problem

on these already inferred RPSs, the initial trees are reduced and then a body and substitution terms are induced. Calculation of a body always succeeds, whereas calculation of substitution terms may fail. One important question is, whether success of calculating substitution terms depends on the already inferred RPSs. Is the (problematic) case possible, that two different sets of RPSs both explain the subtrees which will be replaced, such that calculation of substitution terms from the reduced initial trees succeeds presupposed one particular set of RPSs, but fails for the other set? Fortunately we could prove, that this is *not* possible. *If* a set of RPSs explaining the subtrees exists such that substitution terms can be calculated, then substitution terms can be calculated pressuposed *any* set of RPSs explaining the subtrees. Thus, inducing RPSs for the subtrees can be dealt with as an independent problem as it is done in our divide-and-conquer approach.

4. Generating an Initial Term

Our theory and prototypical implementation for the first synthesis step uses the datatype *List*, defined as follows: The empty list [] is an (α-)list and if a is in element of type α and l is an α-list, then $cons(a,l)$ is an α-list. Lists may contain lists, i.e. α may be of type *List* α'.

4.1. Characterization of the Approach

The constructed initial terms are composed from the list constructor functions $[], cons$, the functions for decomposing lists hd, tl, the predicate *empty* testing for the empty list, one variable x, the 3ary (non-strict) conditional function *if* as control structure, and the bottom constant Ω meaning *undefined*. Similar to Summers (1977), the set of functions used in our term construction approach implies the restriction of induced programs to solve *structural* list programs. An extension to Summers is that we allow the example inputs to be *partially* ordered instead of only totally ordered. This is related to the extension of inducing *sets* of recursive equations as described in Sec. 3 instead of only one recursive equation.

We say that an initial term *explains* I/O-examples, if it evaluates to the specified output when applied to the respective input or to *undefined*. The goal of the first synthesis step is to construct an initial term which explains a set of I/O-examples and which can be explained by an RPS.

4.2. Basic Concepts

Definition 10 (Subexpressions). The set of subexpressions of a list l is defined to be the smallest set which includes l itself and, if l has the form $cons(a,l')$, all subexpressions of a and of l'. If a is an atom, then a itself is its only subexpression.

Since hd and tl—which are defined by $hd(cons(a,l)) = a$ and $tl(cons(a,l)) = l$—decompose lists uniquely, each subexpression can be associated with the unique term which computes the subexpression from the original list. E.g., consider the following I/O-pair which is the third one from Tab. 1: $[[a,b]] \mapsto [b]$. The set of all subexpressions of the input list $[[a,b]]$ together with their associated terms is: $\{x = [[a,b]],\ hd(x) = [a,b],\ tl(x) = [],\ hd(hd(x)) = a,\ tl(hd(x)) = [b],\ hd(tl(hd(x))) = b,\ tl(tl(hd(x))) = []\}$.

Since lists are uniquely constructed by the constructor functions [] and *cons*, traces which compute the specified output can uniquely be constructed from the terms for the subexpressions of the respective input:

Definition 11 (Construction of Traces). Let $i \mapsto o$ be an I/O-pair (i is a list). If o is a subexpression of i, then the trace is defined to be the term associated with o. Otherwise o has the form $cons(a,l)$. Let t and t' be the traces for the I/O-pairs $i \mapsto a$ and $i \mapsto l$ respectively. The the trace for $i \mapsto o$ is defined to be the term $cons(t,t')$.

E.g., the trace for computing the output $[b]$ from its input $[[a,b]]$ is the term $cons(hd(tl(hd(x))), [])$.

Similar to Summers, we discriminate the inputs with respect to their structure, more precisely wrt a partial order over them implied by their structural complexity. As stated above, we allow for arbitrarily nested lists as inputs. A partial order over such lists is given by: $[] \leq l$ for all lists l and $cons(a,l) \leq cons(a',l')$, iff $l \leq l'$ and, if a and a' are again lists, $a \leq a'$.

Consider any unfolding of an RPS. Generally it holds, that greater positions on a path leading to an Ω result from more rewritings of a head of a recursive equation with its body compared to some smaller position. In other words, the computation represented by a node at a greater position is one on a deeper recursion level than a computation represented by a smaller position. Since we use only the complexity of an input list as criterion whether the recursion stops or whether another call appears with the input decomposed in some way, deeper recursions result from more complex inputs in the induced programs.

4.3. Solving the Term Construction Problem

The overall concept of constructing the initial tree is to introduce the nodes from the traces position by position to the initial tree *as long as the traces are equal* and to introduce an *if*-expression as soon as at least two (sub)traces differ. The predicate in the *if*-expression divides the inputs into two sets. The "then"-subtree is recursively constructed from the input/trace-pairs whose inputs evaluate to *true* with the predicate and the "else"-subtree is recursively constructed from the other input/trace-pairs. Eventually only one single input/trace-pair remains when an *if*-

expression is introduced. In this case an Ω indicating a recursive call on this path is introduced as leaf at the current position in the initial term and (this subtree of) the initial tree is finished. The reason for introducing an Ω in this case is, that we assume, that *if* the input/trace-set would contain a pair with a more complex input, than the respective trace would at some position differ from the remaining trace and thus it would imply an *if*-expression, i.e. a recursive call at some deeper position. Since we do not know the position at which this difference would occur, we can not use this single trace, but have to indicate a recursive call on this path by an Ω. Thus, for principal reasons, the constructed initial terms are undefined for the most complex inputs of the example set.

We now consider the both cases that all roots of the traces are equal and that they differ respectively more detailed.

4.3.1. Equal Roots

Suppose all generated traces have the same root symbol. In this case, this symbol constitutes the root of the initial tree. Subsequently the sub(initial)trees are calculated through a recursive call to the algorithm. Suppose the initial tree has to explain the I/O-examples $\{[a] \mapsto a,\ [a,b] \mapsto b,\ [a,b,c] \mapsto c\}$. Calculating the traces and replacing them for the outputs yields the input/trace-set $\{[a] \mapsto hd(x),\ [a,b] \mapsto hd(tl(x)),\ [a,b,c] \mapsto hd(tl(tl(x)))\}$. All three traces have the same root *hd*, thus we construct the root of the initial tree with this symbol. The algorithm for constructing the initial subterm of the constructed root *hd* now starts recusively on the set of input/trace-pairs where the traces are the subterms of the roots *hd* from the three original traces, i.e. on the set $\{[a] \mapsto x,\ [a,b] \mapsto tl(x),\ [a,b,c] \mapsto tl(tl(x))\}$.

The traces from these new input/trace-set have different roots, that is, an *if*-expression is introduced as subtree of the constructed initial tree.

4.3.2. Introducing Control Structure

Suppose the traces (at least two of them) have different roots, as for example the traces of the second input/trace-set in the previous subsection. That means that the initial term has to apply different computations to the inputs corresponding to the different traces. This is done by introducing the conditional function *if*, i.e. the root of the initial term becomes the function symbol *if* and contains from left to right three subtrees: First, a predicate term with the predicate *empty* as root to distinguish between the inputs which have to be computed differently wrt their complexity; second, a tree explaining all I/O-pairs whose inputs are evaluated to *true* from the predicate term; third, a tree explaining the remaining I/O-examples. It is presupposed, that all traces corresponding to inputs evaluating to *true* with the predicate are equal. These equal subtraces become the second subtree of the *if*-expression, i.e. they are evaluated, if an input evaluates to *true* with the predicate. That means that never an Ω occurs in a “then”-subtree of a constructed initial tree, i.e. that recursive calls in the induced RPSs may only occur in the “else”-subtrees. For the “else”-subtree the algorithm is recursively processed on all remaining input/trace-pairs.

5. Experimental Results

We have implemented prototypes (without any thoughts about efficiency) for both described steps, construction of the initial tree and generalization to an RPS. The implementations are in Common-Lisp. In Tab. 4 we have listed experimental results for a few sample problems. The first column lists the names for the induced functions, the second column lists the number of given I/O-pairs, the third column lists the number of induced equations, and the fourth column lists the times consumed by the first step, the second step, and the total time respectively. The experiments were performed on a Pentium 4 with Linux and the program runs are interpreted with the *clisp* interpreter.

Table 4. Some inferred functions

function	#expl	#eqs	times in sec
last	4	2	.003 / .001 / .004
init	4	1	.004 / .002 / .006
evenpos	7	2	.01 / .004 / .014
switch	6	1	.012 / .004 / .016
unpack	4	1	.003 / .002 / .005
lasts	10	2	.032 / .032 / .064
mult-lasts	11	3	.04 / .49 / .53
reverse	6	4	.031 / .036 / .067

All induced programs compute the intended function. The number of given examples is in each case the minimal one. When given one example less, the system does *not* produce an *unintended* program, but produces *no* program. Indeed, an initial term is produced in such a case which is correct on the example set, but no RPS is generalized, because it exists no RPS which explains the initial term *and is substitution unique wrt it* (see Sec. 3.3).

last computes the last element of a list. The main equation is not recursive and only applies a *hd* to the result of the induced recursive equation which computes a one element list containing the last element of the input list. *init* returns the input list without the last element. *evenpos* computes a list containing each second element of the input list. The main equation is not recursive and only deals with the empty input list as special case. *switch* returns a list, in which each two successive elements of the input list are on switched positions, e.g., $switch([a,b,c,d,e]) = [b,a,d,c,e]$. *unpack* produces an output list, in which each element

from the input list is encapsulated in a one element list, e.g., $unpack([a,b,c]) = [[a],[b],[c]]$. *unpack* is the classical example in (Summers, 1977). *lasts* is the program described in Sec. 2. The given I/O-examples are those from Tab. 1. *mult-lasts* takes a list of lists as input just like *lasts*. It returns a list of the same structure as the input list where each inner list contains repeatedly the last element of the corresponding inner list from the input. For example, $mult\text{-}lasts([[a,b],[c,d,e],[f]]) = [[b,b],[e,e,e],[f]]$. All three induced equations are recursive. The third equation computes a one element list containing the last element of an input list. The second equation utilizes the third equation and returns a list of the same structure as a given input list where the elements of the input list are replaced by the last element. The first equation utilizes the second equation to compute the inner lists. Finally *reverse* reverses a list. The induced program has an unusual form, nevertheless it is correct.

6. Conclusion and Further Research

We presented an EBG approach to inducing sets of recursive equations representing functional programs from I/O-examples. The underlying methodologies are inspired by classical approaches to induction of functional Lisp-programs, particularly by the approach of Summers (1977). The presented approach goes in three main aspects beyond Summers' approach: *Sets* of recursive equations can be induced at once instead of only one recursive equation, each equation may contain more than one recursive call, and additionally needed parameters are introduced systematically. We have implemented prototypes for both steps. The generalizer works domain-independent and all problems which comply to our general program scheme (Def. 1) with the restrictions described in Sec. 3.3 can be solved, whereas construction of initial terms as described in Sec. 4 relies on knowledge of datatypes.

We are investigating several extensions for the first synthesis step: First, we try to integrate knowledge about further datatypes such as trees and natural numbers. For example, we believe, that if we introduce *zero* and *succ*, denoting the natural number 0 and the successor function resp. as constructors for natural numbers, *prev* for "decomposing" natural numbers and the predicate *zerop* as bottom test on natural numbers, then it should be possible to induce a program returning the length of a list for example. Another extension will be to allow for more than one input parameter in the I/O-examples, such that *append* becomes inducable for example. A third extension should be the ability to utilize user-defined or in a previous step induced functions within an induction step.

Until now our approach suffers from the restriction to structural problems due to the principal approach to calculate traces deterministically without search in the first synthesis step. We work on overcoming this restriction, i.e. on extending the first synthesis step to the ability of dealing with problems which are not (only) structural, list sorting for example. A strong extension to the second step would be the ability to deal with nested recursive calls, yet this would imply a much more complex structural analysis on the initial terms.

References

Biermann, A. W., Guiho, G., & Kodratoff, Y. (Eds.). (1984). *Automatic program construction techniques*. Collier Macmillan.

Dershowitz, N., & Jouanaud, J.-P. (1990). Rewrite systems. In J. Leeuwen (Ed.), *Handbook of theoretical computer science*, vol. B. Elsevier.

Flener, P., & Partridge, D. (2001). Inductive programming. *Autom. Softw. Eng.*, *8*, 131–137.

Flener, P., & Yilmaz, S. (1999). Inductive synthesis of recursive logic programs: Achievements and prospects. *Journal of Logic Programming*, *41*, 141–195.

Kitzelmann, E. (2003). Inductive functional program synthesis – a term-construction and folding approach. Master's thesis, Dept. of Computer Science, TU Berlin. http://www.cogsys.wiai.uni-bamberg.de/kitzelmann/documents/thesis.ps.

Lowry, M. L., & McCarthy, R. D. (1991). *Autmatic software design*. Cambridge, Mass.: MIT Press.

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming, Special Issue on 10 Years of Logic Programming*, *19-20*, 629–679.

Olsson, R. (1995). Inductive functional programming using incremental program transformation. *Artifi cial Intelligence*, *74*, 55–8.

Plotkin, G. D. (1969). A note on inductive generalization. In *Machine intelligence*, vol. 5, 153–163. Edinburgh University Press.

Rao, M. R. K. K. (2004). Inductive inference of term rewriting systems from positive data. *ALT* (pp. 69–82).

Schmid, U., & Wysotzki, F. (2000). Applying inductive programm synthesis to macro learning. *Proc. 5th International Conference on Artifi cial Intelligence Planning and Scheduling (AIPS 2000)* (pp. 371–378). AAAI Press.

Summers, P. D. (1977). A methodology for LISP program construction from examples. *Journal ACM*, *24*, 162–175.

Wysotzki, F., & Schmid, U. (2001). *Synthesis of recursive programs from fi nite examples by detection of macro-functions* (Technical Report 01-2). Dept. of Computer Science, TU Berlin, Germany.

Synthesis of Scientific Algorithms based on Evolutionary Computation and Templates

Oleg Monakhov and Emilia Monakhova {MONAKHOV,EMILIA}@RAV.SSCC.RU
Institute of Computational Mathematics and Mathematical Geophysics SB RAS, Pr. Lavrentieva, 6, Novosibirsk, 630090, Russia

Abstract

This work describes a new approach for the synthesis of algorithms based on given templates and a set of input-output pairs using evolutionary computation. The presented algorithm of evolutionary synthesis integrates the advantages of the genetic algorithms and genetic programming and was applied for automatically rediscovery and discovery of several computational, combinatorial and graph algorithms.

1. Introduction

In this work the problem of synthesis of an algorithm A is considered as a problem of searching for the parameters and functions of the given template T of an algorithm with the aim of optimization of a given objective function F characterized as a quality of the algorithm A. The template (skeleton (Cole, 1989; Mirenkov & Mirenkova, 1996), design pattern (Gamma et al., 1994)) is a parameterized control structure of the algorithm. The template describes a scanning order of data structures of the algorithm and defines the computational dynamics of the algorithm in space-time coordinates. The template T of the algorithm A contains parameters $P = \{p_k\}$, $k \geq 0$, which describe the values of input and local variables, parameters of the data structures, constants and some primitive operations of the algorithm. The template T also contains a set of functions (formulas) $FM = \{f_n\}$, $n \geq 0$, of the algorithm A. When giving the values of the parameters P and defining the functions FM in the template T, the algorithm $A(T, P, FM)$ is obtained. The objective function F estimates the discrepancy between the observed output data of algorithm $Y'_i = A(T, P, FM, X_i)$ and the given expected values Y_i for the given input values X_i, $1 \leq i \leq N$. The function F also should estimate the complexity of the algorithm $A(T, P, FM)$.

Thus, for the given template T and the given input-output values $\{X_i, Y_i\}$, $1 \leq i \leq N$, a problem of algorithm discovery is to find the parameters P^* and to determine the functions FM^* in the template T defining of the algorithm $A^*(T, P^*, FM^*)$ such that

$$F(A^*(T, P^*, FM^*, X_i)) \leq F(A(T, P, FM, X_i))$$

for all $1 \leq i \leq N$, $P \in Dom(P)$, $FM \in Dom(FM)$.

As a solution to the problem, a new template-based evolutionary approach is proposed for computer discovery (synthesis) of algorithms optimizing a given objective function. This approach integrates the templates, genetic algorithms (GA) (Goldberg, 1989), genetic programming (GP) (Koza, 1992) and obtains some new properties: more complex loop structure and recursion of the created algorithms than in genetic programming, and synthesis of new functions and predicates that the genetic algorithms can not create. These properties of the approach are based on the background knowledge and generalization of the expected algorithm and application field included in the template. The traditional genetic programming (GP) suffers from weak-restricted blind search in huge spaces for real-world problems and has the high time efforts. This work investigates the use of templates of algorithms to restrict the search space to admissible models and structures of solutions.This approach represents the flexible and direct way to incorporate expert knowledge about parameters, variables, functions and structures of the problems into being developed computation models. In the extreme cases, on the one hand, if we do not know a template of the algorithm, this approach gives us the traditional genetic programming. On the other hand, if we know a full structure of the algorithm and do not know only some parameters, this approach gives us the traditional genetic algorithm for searching of the optimal parameters.

A practical approach to a hybrid GA/GP search without templates and multiple trees was used in (Andre, 1994; Nguyen & Huang, 1994; Lee, Hallam & Lund, 1996). The GA performs the search to find the right

values for the constants, while the GP searches the space of parse trees. The chromosome representation contains both the parse tree and the constant values, and there are different genetic operators for manipulating the parse trees and the constants. Another interesting approach in (Olsson, 1998) to the synthesis of scientific algorithms and programs is based on evolutionary computation but without genetic operations and templates.

2. Template-based Evolutionary Algorithm for Automatic Discovery

The automatic discovery algorithm is based on evolutionary computation and the simulation of the survival of the fittest in a population of individuals, each being presented by a point in the space of solutions of the optimization problem. The individuals are presented by data structures Gen - chromosomes. Each chromosome contains underdeterminated parameters p_k and functions (formulas) f_n of the template: $Gen = \{P, FM\} = \{p_1, p_2, ..., p_k; f_1, f_2, ..., f_n\}$, $k, n \geq 0$. These parameters and functions determine the required algorithm $A(T, Gen)$ based on the given template T and the chromosome Gen.

Each population is a set of chromosomes Gen and determines a set of algorithms $A(T, Gen)$ generated based on the template T.

The main idea of the discovery algorithm consists in the evolutionary transformations over sets of the chromosomes (parameters and formulas of the template) based on a natural selection: "the strongest" survive. In our case these individuals are algorithms giving the best possible value of the objective (fitness) function. In the algorithm the starting point is the generation of the initial population. All individuals of the population are created at random, the best individuals are selected and saved. To create the next generation, new solutions are formed through genetic operations named selection, mutation, crossover and adding new elements.

The function F named as fitness function evaluates the sum of quadratic deviations of output data of algorithm $Y_i' = A(T, Gen, X_i)$ from the given expected values Y_i for the given input values X_i, $1 \leq i \leq N$:

$$F = \sum_{i=1}^{N} (A(T, Gen, X_i) - Y_i)^2 + C(A(T, Gen)),$$

where $C(A(T, Gen))$ is an estimation of the complexity of the algorithm A (a time of execution, a number of iterations, a complexity of formulas). In practice, we use a number of iterations for the estimation of the complexity in the case of synthesis of iterative algorithms, otherwise we use the sum of number of nodes (length) of formulas. The purpose of the discovery algorithm is to search for a minimum of F.

3. Data representation

Basic data structures in our program realizing the evolutionary algorithm are the chromosomes Gen.

In this work a new approach for representation of the chromosomes Gen is proposed. The chromosome Gen is based on an integration of a linear structure of the chromosome for representation of parameters p_k (as in genetic algorithms (Goldberg, 1989)) and a multi-tree structure of the chromosome for representation of functions (formulas) f_n (as in genetic programming (Koza, 1992)). The linear structure of a chromosome is used for representation of the following parameters p_k of the given template T: values of integer and real variables and constants; values of indices, increments and decrements; signs of variables, logic operations and relations, types of rounding.

The multi-tree structure of a chromosome is used for representation of the functions (formulas) f_n of the given template T. The tree corresponds to the parse tree of the function. The variables and constants of the formula f_n are represented by terminal nodes TS of the tree. The operations and primitive functions used in the formula f_n are represented by nonterminal nodes NS of the tree. Each operation (primitive function) of the formula and its operands (the arguments of the primitive function) are represented by a node and its descendant nodes in the tree. For example, in Fig. 1 the tree representation for the formula $f_n = (x + 2)/\sqrt{a * x - 5}$ has the following nodes: $TS = \{x, a, 2, 5\}$, $NS = \{+, -, *, /, \sqrt{\ }\}$.

Using the template T and generating the chromosomes Gen, we create an analytical expression for each function f_n and determine a value for each parameter p_k and, after that, we can already produce all evaluations and modifications of the algorithm $A(T, Gen)$. Thus, for the known values of the functions f_n and parameters p_k we can calculate the output values Y_i' of the algorithm $A(T, Gen, X_i)$ generated by evolution of the chromosomes Gen based on the given template T for the given input values X_i, $1 \leq i \leq N$. After evaluation of the algorithms A we obtain the values of the fitness function F and select the best algorithms in the population.

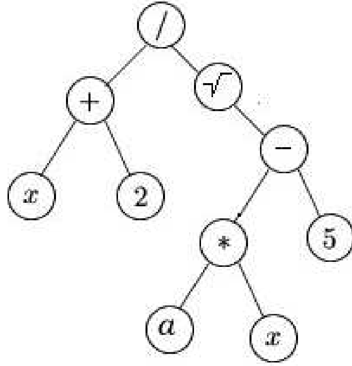

Figure 1. Tree representation of formula.

4. Operators of the Algorithm

The *mutation* operator is applied to the individuals (chromosomes) chosen randomly from the current population with a probability $p_m \in [0, 1]$. Mutation represents a modification of an individual whose number is randomly selected. The modification of the linear structure of the chromosome is understood as a replacement of a randomly chosen parameter p_i by another value selected at random from a set of admissible values. The modification of the tree structure of the chromosome is performed by a replacement of the value of a randomly chosen node in the tree representation of function f_i by another value selected at random from the set of admissible values.

The *crossover* operator is applied to the two individuals (parents) chosen randomly from the current population with a probability $p_c \in [0, 1]$. The crossover consists of the generation of two new individuals by exchanging the parts of the chromosomes of the parents. For the linear (or tree) structures of the chromosomes the crossover is performed by replacing a randomly chosen linear substructure (subtree) of one parent by a linear substructure (subtree) from the other parent.

The creation of a *new element* is the generation of random parameters p_k and functions f_n for the chromosomes. It allows for the adding of a diversification to the elements of a population.

The *selection operator* realizes the principle of the survival of the fittest individuals. It selects the best individuals with the minimum fitness function in the current population.

Note, only simple genetic operators were used in the algorithm, but this approach can use more complex operators developed for GP and GA.

5. Iteration Process

In the search for the optimum of the fitness function F the iteration process in the computer discovery algorithm is organized in the following way.

First iteration: a generation of the initial population. It is realized as follows. All individuals of the population are created by means of the operator *new element* (with a test and rejection of all "impractical" individuals). After filling the whole population, the best individuals are selected and saved in an array *best*.

One iteration: a step from the current population towards the next population. The basic step of the algorithm consists of creating a new generation on the basis of selection, mutation, crossover and also adding some new elements.

After evaluation of fitness function for each individual of the generation, a comparison of the value of this function to values of fitness function of those individuals which are saved in the array *best* is executed. In this case, if an element from the new generation is better than an element $best[i]$, for some i, we locate the new element on place i and shift all remaining ones per a unit of downwards. Thus, the best element is located at the top of the array *best*.

Last iteration (the termination criterion): the iterations are finished either after a given number of steps $T = t$ or after finding optimal algorithm $A(T, Gen)$ (with the given value of fitness function).

By producing a given amount of the basic steps of the template-based evolutionary algorithm, we obtain a set of algorithms $A(T, Gen)$ containing an algorithm $A^*(T, Gen)$ with the minimum fitness function F in the element $best[0]$.

6. Experimental Results

The template-based evolutionary approach was applied for rediscovery of the following algorithms: computation of the power and factorial of a natural number, finding the least (largest) element of an array, computation of the sum of the squares of elements of an array, computation of the dot product of two vectors, finding the formulas for the Fibonacci (Tribonacci) sequence, computation of the sum of matrices, bubble, merge and Shell sort, finding the roots of an equation, load balancing in parallel system, finding the single source shortest paths and minimal spanning tree in a graph. The approach was also applied for discovery of analytical descriptions of new dense families of optimal regular networks (Monakhov & Monakhova, 2003) and a distance function of circulant graphs with

degree four.

The realization of the template-based evolutionary algorithm has been implemented in the C programming language and templates have been presented in this language. The number of iterations and population size were chosen by an experimental way based on parameters from (Goldberg, 1989),(Koza, 1992).

6.1. Finding the roots of the second-order equations

For the first example, the process of rediscovery of algorithms for finding the roots of the second-order equations: $f(x) = ax^2 + bx + c = 0$ is presented. Let 20 of input-output pairs $\{X_i, Y_i\}$ be given, where $X_i = (a_i, b_i, c_i)$ are the coefficients of the equations, $Y_i = (r1_i, r2_i)$ are roots of the equations (for simplicity, we will consider only real roots), $1 \leq i \leq N$, $N = 20$. The following two templates are used: the first template T_1 is given as the following formula:

$$r_{1,2} = f_1(a,b,c) \pm f_2(a,b,c) .$$

Note that the unknown functions are shaded.

The second template T_2 of an approximation algorithm is given as the following iterative loop:

1 $\{k = 0; x_k = x_{beg};$
2 **do** $\{x_{k+1} = f_1(x_k, f(x_k), f'(x_k))$; $k = k + 1; \}$
3 **while** $((f'(x_k) = \epsilon > 10^{-7})\&(k < 500));$
4 **return** $x_k\}$,

with a limited number of iterations $it < 500$, with a given precision of approximation $\epsilon < 10^{-7}$ and an initial point x_{beg}, with a procedure for calculation of derivative $f'(x)$.

The terminal nodes for T_1 are $TS_1 = \{a, b, c, Cr\}$, and for T_2 are $TS_2 = \{x_k, f(x_k), f'(x_k), Cr\}$, where Cr is a set of random natural constants. The set of operations used for synthesis of the formulas is $NS = \{+, -, *, /, \sqrt{\ }, x^2\}$ for the both templates.

In the case of the first template the template-based evolutionary algorithm rediscovered the following known formula: $r_{1,2} = -b/2a \pm \sqrt{b^2 - 4ac}/2a$. In the case of the second template the discovery algorithm found the following expression: $x_{k+1} = x_k - f(x_k)/f'(x_k)$. This result corresponds to the known formula of Newton's method (method of tangents). The first result has been found after 10216 iterations (for time 70 sec.) with a population of 200. The second result has been found after 360 iterations (for time 10 sec.) with a population of 200.

6.2. Finding the recursive functions for the Fibonacci and Tribonacci numbers

The second example shows how the recursive function for the Fibonacci numbers can be generated from the given template and the first eight numbers with index $1 \leq i \leq 8$. We use the following template:

$$F(i) = f_3\,(F(\,f_1(i)\,), F(\,f_2(i)\,)).$$

With the set of operations $NS = \{+, -, *, /\}$ and set of terminals $TS = \{i, Cr\}$ the template-based evolutionary algorithm defined $f_1(i) = i - 1$, $f_2(i) = i - 2$ and $f_3(x_1, x_2) = x_1 + x_2$ after 411 iterations (for time 12 sec.) with a population of 3000.

Tribonacci numbers can be generated from the following template:

$$F(i) = f_4\,(F(\,f_1(i)\,), F(\,f_2(i)\,), F(\,f_3(i)\,)).$$

The template-based evolutionary algorithm defined $f_1(i) = i - 1$, $f_2(i) = i - 2$, $f_3(i) = i - 3$ and $f_4(x_1, x_2, x_3) = x_1 + x_2 + x_3$ after 461 iterations (for time 209 sec.) with a population of 30000, with the set of operations $NS = \{+, -\}$, set of terminals $TS = \{i, Cr\}$ and with the given first ten numbers with index $1 \leq i \leq 10$.

6.3. Finding the distance function of circulant graphs with degree four

For this example, the distance function of circulant graphs with degree 4 is created. The class of circulant networks (Bermond, Comellas & Hsu, 1995; Monakhov & Monakhova, 2000; Hwang, 2003) plays an important role in the design and implementation of interconnection networks. A circulant graph, having the parametric description, is defined as follows. *A circulant* is an undirected graph $G(N; s_1, s_2, \ldots, s_n)$ with a set of nodes $V = 0, 1, 2, \ldots, N - 1$, having $i \pm s_1, i \pm s_2, \ldots, i \pm s_n \pmod{N}$ nodes, adjacent to each node i.

The numbers $S = (s_i)$ $(0 < s_1 < \ldots < s_n < N/2)$ are generators of the finite Abelian group of automorphisms connected to the graph. Circulant graphs $G(N; 1, s_2, \ldots, s_n)$, with the identity generator, are known as loop networks (Bermond, Comellas & Hsu, 1995). The degree of a node in circulant graph G is $2n$, where n is the dimension. We will consider loop networks with degree 4, i.e. circulant networks of the form $G(N; 1, s)$. For example, a circulant graph $C(14; 1, 6)$ with degree 4, $N = 14, s_1 = 1, s_2 = 6$ is shown in Fig. 2.

The diameter of G is defined as $d(N; S) = \max_{u,v \in V} D(u, v)$, where $D(u, v)$ (the *distance func-*

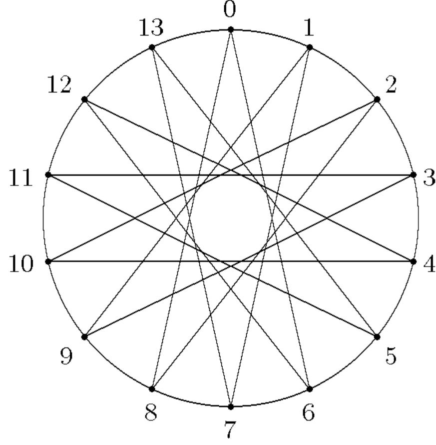

Figure 2. Circulant $C(14; 1, 6)$.

tion) is the length of a shortest path between nodes u and v in G. Because of the symmetry in circulants it is enough to consider the problem of finding a shortest path from 0 to an arbitrary node v. The distance function for nodes 0 and v in circulant $C(200; 1, s)$, $N = 200$, for $0 < s < N/2$ and $0 < v < N/2$ is shown in Fig. 3.

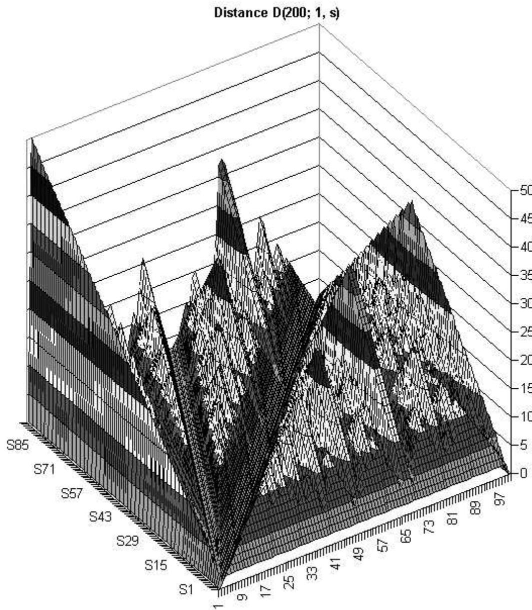

Figure 3. Distance function of circulant $C(200; 1, s)$.

For finding the distance function $D(0, v)$ we will consider the following consideration. For any node w we define $+s$ and $-s$ links from node w depending on whether they are used to go to node $(w + s) mod\ N$ (in clockwise direction) or $(w - s) mod\ N$ (in counterclockwise direction). Similarly, we define $+1$ and -1 links. Note that in circulant graphs a shortest path from 0 to v would be using at most either $(+s, +1)$ or $(+s, -1)$ or $(-s, +1)$ or $(-s, -1)$ links. Therefore further we will consider such combinations of links only. Let $(+s, +1)$-path be a path from 0 to v using $+s$ and $+1$ links only. For other combinations of links we define the analogous notations. In what follows $x/s = \lfloor x/s \rfloor$, $x\%s = x\ mod\ s$.

In order to go to node v from 0 by means of four possible ways we have to use

1) $(+s, +1)$-path: using v/s number of $+s$ links and $v\%s$ number of $+1$ links;

2) $(+s, -1)$-path: using $v/s + 1$ number of $+s$ links and $s - v\%s$ number of -1 links;

3) $(-s, -1)$-path: using $(N - v)/s$ number of $-s$ links and $(N - v)\%s$ number of -1 links;

4) $(-s, +1)$-path: using $(N - v)/s + 1$ number of $-s$ links and $s - (N - v)\%s$ number of $+1$ links.

This corresponds to one loop travelled in clockwise direction and one loop travelled in counterclockwise direction ($t = 0$). Generalizing this process for $t \geq 0$, we obtain for node v :

1) all $(+s, +1)$-paths: using $(v + tN)/s$ number of $+s$ links and $(v + tN)\%s$ number of $+1$ links;

2) all $(+s, -1)$-paths: using $(v+tN)/s+1$ number of $+s$ links and $s - (v + tN)\%s$ number of -1 links;

3) all $(-s, -1)$-paths: using $((t+1)N - v)/s$ number of $-s$ links and $((t + 1)N - v)\%s$ number of -1 links;

4) all $(-s, +1)$-paths: using $((t+1)N-v)/s+1$ number of $-s$ links and $s - ((t + 1)N - v)\%s$ number of $+1$ links.

Note that because of the symmetry of circulants the $(-s, +1)$ and $(-s, -1)$-paths from 0 to $v + tN$ can be changed to the $(+s, -1)$ and $(+s, +1)$-paths, respectively, from 0 to node $(t + 1)N - v$, $t \geq 0$.

It is necessary to find the shortest paths of all the four types and the shortest of the four will give us a global shortest path between 0 and v. The number of loops $t < s$ because $v\%s < s$ for any $0 \leq v < N$.

Based on background knowledge of circulant properties the following template T for the distance function

$dist = D(0, v)$ of circulant $C(N; 1, s)$ is used:

```
1  int t, k, k2, r, r2, d, d1, d2, dist = N;
2  for(t = 0; t < s; t = t + 1)
3     {k = (v + t * N)/s; r = (v + t * N)%s;
4     k2 = ((t+1) * N - v)/s; r2 = ((t+1) * N - v)%s;
5     d1 = f1(k, r, s) ; d2 = f1(k2, r2, s) ;
6     d = min(d1, d2); if (dist > d) dist = d; }
7  return dist
```

In the line 1 of the template the needed local variables are defined. In the line 2 we have the operator **for** with the limited number of loops $t < s$. In the lines 3 and 4 the variables k and r define the numbers of $+s$ and $+1$ links, respectively, for path from 0 to v in clockwise direction, and, similarly, the variables $k2$ and $r2$ define the numbers of $-s$ and $+1$ links for path from 0 to v in counterclockwise direction. In the line 5, for the current loop t, the undefined function $f_1(k, r, s)$ has to calculate the length $d1$ of the shortest path from 0 to v in clockwise direction, and, similarly, $f_1(k2, r2, s)$ calculates the length $d2$ of the shortest path in counterclockwise direction. In the line 6 the length d ($dist$) of the shortest path from 0 to v for the current loop t (for all loops $t' < t$) is defined. In the line 7 we have the result: $dist = D(0, v)$.

The terminal nodes for the undefined function $f_1(x_1, x_2, x_3)$ are local variables $\{k, k2, r, r2\}$, a global $\{s\}$ and $\{Cr\}$ (a set of random natural constants). The set of operations used for synthesis of the formula f_1 is $NS = \{+, -, \min, \lfloor x \rfloor\}$.

For this template the template-based evolutionary algorithm found the following expression: $f_1(x_1, x_2, x_3) = \min((x_1 + x_2), (x_1 + 1 + x_3 - x_2))$. The function $f_1(k, r, s)$ calculates the length of the shortest path from 0 to v in clockwise direction as the minimum of the lengths $(k + r)$ and $((k + 1) + (s - r))$ of the $(+s, +1)$- and $(+s, -1)$-paths, respectively, and, similarly, the function $f_1(k2, r2, s)$ calculates the length of the shortest path from 0 to v in counterclockwise direction (both for the current loop t). The result has been found for the given 99 input-output pairs $\{v, D(0, v)\}$, $1 \le v \le 99$, for graph $C(200; 1, s)$ after 127 iterations (for time 270 sec.) with a population of 500. The correctness for computation of the distance function $D(u, v)$ of circulant based on template T and formula f_1 was proved experimentally and theoretically.

The upper estimate of t (equal to s in the line 2 of the above template) can be decreased.

Lemma 1 *The number of loops in the algorithm defining the distance function (the line 2 of the above template) may not exceed the following value:* $\lfloor (s/2 + 1)/\lfloor N/s \rfloor \rfloor$.

As a result we have

Lemma 2 *The computation of the distance function $D(0, v)$, $0 \le v < N$, for loop network $C(N; 1, s)$ based on template T, formula f_1 and with the number of loops defined by Lemma 1 is correct.*

This algorithm can be used to solve other problems in loop networks of degree 4, such as the routing problem of finding a shortest path between two nodes, or finding the diameter of a graph. For solving the first problem it is sufficient to store numbers of steps and signs of two generators giving a shortest path if the operation $dist = d$ was realized in the line 6 of the above template T.

In (Mukhopadhyaya & Sinha, 1995; Narayanan & Opatrny, 1997; Robic & Zerovnik, 2000), the algorithms of finding a shortest path between any pair of nodes in loop networks of degree 4 are given. The above algorithm generated by the template-based evolutionary algorithm differs from all known algorithms and its estimate is not worse.

7. Conclusions

The represented template-based evolutionary approach has been used successfully to automatically invent computational algorithms and for discovery of mathematical formulas for the given data sets and for the given algorithm's templates (e.g. iterations, recursions, loops and cycles), which describe the scanning of the complex data structures (matrixes, arrays, graphs, trees) and which contain the formula templates in the body. This approach can be used for synthesis of new algorithms, functions, models and solutions, which afterwards can be theoretically investigated and justified.

References

Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation.* The MIT Press.

Mirenkov, N., & Mirenkova, T. (1996). Multimedia Skeletons and Filmification of Methods. *Proc. of The First International Conference on Visual Information Systems* (pp. 58–67). Victoria University, Melbourne, Australia.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable*

Object-Oriented Software. Addison-Wesley, Reading, MA.

Goldberg, D. E. (1989). *Genetic Algorithms, in Search, Optimization and Machine Learning.* Addison-Wesley, Reading, MA.

Koza, J. (1992). *Genetic Programming.* Cambridge, The MIT Press.

Andre, D. (1994). Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and algorithm for using them. In K. Kinnear (Ed.), *Advances in Genetic Programming* (pp. 477-494). MIT Press/Bradford Books.

Nguyen, T. & Huang, T. (1994). Evolvable 3D modeling for model-based object recognition systems. In K. Kinnear (Ed.), *Advances in Genetic Programming* (pp. 459-475). MIT Press/Bradford Books.

Lee, W.P., Hallam, J. & Lund, H. H. (1996). A Hybrid GP/GA Approach for Coevolving Controllers and Robot Bodies to Achieve Fitness - Specified Tasks. *Proc. of IEEE International Conf. on Evolutionary Computation.* IEEE Press.

Olsson, J. R. (1998). Population management for automatic design of algorithms through evolution, *Proc. of IEEE International Conference on Evolutionary Computation*, IEEE Press.

Monakhov, O. & Monakhova, E. (2003). An Algorithm for Discovery of New Families of Optimal Regular Networks. *Proc. of 6th Inter. Conf. on Discovery Science (DS 2003),* Oct. 17-20, 2003, Sapporo, Japan, Lecture Notes in Artificial Intelligence, vol. 2843, (pp. 244-254). Springer- Verlag, Berlin Heidelberg.

Bermond, J.-C., Comellas, F. & Hsu, D.F. (1995). Distributed loop computer networks: a survey, *J. Parallel Distributed Comput.*, *24*, 2–10.

Monakhov, O. & Monakhova, E. (2000). *Parallel Systems with Distributed Memory: Structures and Organization of Interactions*, Novosibirsk, SB RAS Publ. (in Russian).

Hwang, F.K. (2003). A survey on multi-loop networks. *Theoretical Computer Science*, 2003, *299*, 107–121.

Mukhopadhyaya, K. & Sinha, B.P. (1995). Fault-tolerant routing in distributed loop networks. *IEEE Trans. Comput.*, *44(12)*, 1452–1456.

Narayanan, L. & Opatrny, J. (1997). Compact routing on chordal rings of degree four. In D. Krizanc and P. Widmayer, (Ed.), *Sirocco 97*, Carleton Scientific, 125–137.

Robic, B. & Zerovnik, J. (2000). Minimum 2-terminal routing in 2-jump circulant graphs. *Computers and Artificial Intelligence*, *19(1)*, 37–46.

Kernels on Prolog Proof Trees: Statistical Learning in the ILP Setting

A. Passerini PASSERINI@DSI.UNIFI.IT
P. Frasconi P-F@DSI.UNIFI.IT
Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze

L. De Raedt DERAEDT@INFORMATIK.UNI-FREIBURG.DE
Institute for Computer Science, Albert-Ludwigs Universität, Freiburg

Abstract

We develop kernels for measuring the similarity between relational instances using background knowledge expressed in first-order logic. The method allows us to bridge the gap between traditional inductive logic programming representations and statistical approaches to supervised learning. Logic programs will be used to generate proofs of given visitor programs which exploit the available background knowledge, while kernel machines will be employed to learn from such proofs. We report positive empirical results on Bongard-like and M-of-N problems that are difficult or impossible to solve with traditional ILP techniques, as well as on a real data set.

1. Introduction

Within the field of automated program synthesis, inductive logic programming and machine learning, several approaches exist that learn from example-traces. An example-trace is a sequence of steps taken by a program on a particular example input. For instance, Alan Bierman (Biermann & Krishnaswamy, 1976) has sketched how to induce Turing machines from example-traces; Mitchell et al. have developed the LEX system (Mitchell et al., 1983) that learned how to solve symbolic integration problems by analyzing traces (or search trees) for particular example problems; Ehud Shapiro's Model Inference System (Shapiro, 1983) inductively infers logic programs by reconstructing the proof-trees and traces corresponding to particular facts; and Zelle and Mooney (Zelle & Mooney, 1993) show how to speed-up the execution of logic programs by analyzing example-traces of the underlying logic program. The diversity of these applications as well as the difficulty of the learning tasks considered clearly illustrate the power of learning from example-traces for a wide range of applications.

In the present paper, we generalize the idea of learning from example-traces. Rather than explicitly learning a target program from positive and negative example traces, we assume that a particular – so-called *visitor* program – is given and that our task consists of learning from the associated traces. The advantage is that in principle any programming language can be used to model the visitor program and that any machine learning system able use traces as an intermediate representation can be employed. In particular, this allows us to combine two frequently employed frameworks within the field of machine learning: inductive logic programming and kernel methods. Logic programs will be used to generate traces corresponding to specific examples and kernels will be employed for quantifying the similarity between traces. The combination yields an appealing and expressive framework for tackling complex learning tasks involving structured data in a natural manner. We call *trace kernels* the resulting broad family of kernel functions obtainable as a result of this combination. The visitor program is a set of clauses that can be seen as the *interface* between the available background knowledge and the kernel itself. Intuitively, visitors are employed to specify a set of useful features and in this sense play a role similar to `rmode`s in ILP.

Starting from the seminal work of Haussler (Haussler, 1999), several researchers have already proposed kernels on discrete data structures such as sequences (Lodhi et al., 2000; Jaakkola & Haussler, 1998; Leslie et al., 2002; Cortes et al., 2004), trees (Collins & Duffy, 2002; Vishwanathan & Smola, 2002), annotated graphs (Gärtner, 2003; Schölkopf & Warmuth, 2003), and complex individuals defined using higher order logic abstractions (Gärtner et al., 2004). Constructing kernels on structured data types, however, is not the only aim of the proposed framework. In many

symbolic approaches to learning, logic programs allow us to define background knowledge in a very natural way. Similarly, in the case of kernel methods, the notion of similarity between two instances expressed by the kernel function is the main tool for exploiting the available domain knowledge. It seems therefore natural to seek a link between logic programs and kernels, also as a mean for embedding knowledge into statistical learning algorithms in a *principled* and *flexible* way. This aspect is an important contribution of this paper as few alternatives exist to achieve this goal. Propositionalization, for example, transforms a relational problem into one that can be solved by an attribute-value learner by mapping data structures into a finite set of features (Kramer et al., 2000). Although it is known that in many practical applications propositionalization works well, its flexibility is generally limited. A remarkable exception is the method proposed in (Cumby & Roth, 2002) that uses description logic to specify features and that has been subsequently extended to specify kernels (Cumby & Roth, 2003).

The guiding philosophy of trace kernels is very different from the above approaches. Intuitively, rather than defining a kernel function that compares two given instances, we define a kernel function that compares the execution traces of a program (that expresses background knowledge) run over the two given instances. Similar instances should produce similar traces when probed with programs examining characteristics they have in common. Clearly these characteristics can be more general than parts. Hence, trace kernels can be introduced with the aim of achieving a greater generality and flexibility with respect to convolution and decomposition kernels. In particular, *any* program to be executed on data can be exploited within the present framework to form a valid kernel function, provided one can give a suitable definition of the *visitor* program to specify how to obtain relevant traces and proofs to compare examples. In addition, although in this paper we only study trace kernels for logic programs, similar ideas could be used in the context of different programming paradigms and in conjunction with alternative models of computation such as finite state automata or Turing machines.

In this paper, we focus on a specific learning framework for Prolog programs. Prolog execution traces consist of sets of search trees (see e.g. (Sterling & Shapiro, 1994)) associated with goals in the visitor program; these traces can be conveniently represented as Prolog ground terms. Thus, in this case, kernels over traces reduce to Prolog ground terms kernels (PGTKs) (Passerini & Frasconi, 2005). These kernels (which are briefly reviewed in Section 3.3) can be seen as a specialization to Prolog of the kernels between higher order logic individuals earlier introduced in (Gärtner et al., 2004).

The paper is organized as follows. In Section 2 we revise the classic ILP framework and describe the structure of visitor programs. In Section 3 we describe the general form of the kernel on logical objects and, in particular, Prolog proof trees, in Section 4 we give some implementation details, and finally in Section 5 we report an empirical evaluation of the methodology on some classic ILP benchmarks including Bongard problems, M of N problems on sequences, and mutagenesis.

2. Visitors and proof trees in First Order Logic

In traditional inductive logic programming approaches, the learner is given a set of positive and negative examples P and N (in the form of definite clauses that are (resp. are not) entailed by the target theory), and a background theory BK (a set of definite clauses), and has to induce a hypothesis H (a set of definite clauses) such that $BK \cup H$ covers all positive examples and none of the negative ones. More formally, $\forall p \in P : BK \cup H \models p$ and $\forall n \in N : BK \cup H \not\models n$. In practice, rather than working with ground clauses of the form $e \leftarrow f_1, ..., f_n$ as examples, inductive logic programming systems often employ e as the example and add the facts f_i to the background theory BK. As an illustration, consider the famous mutagenicity benchmark by (Srinivasan et al., 1996). There the examples are of the form mutagenic(id) where id is a unique identifier of the molecule and the background knowledge contains information about the atoms, bonds and functional groups in the molecule. A hypothesis in this case could be

```
mutagenic(ID) ← nitro(ID,R), lumo(ID,L), L < -1.5.
```

It entails, i.e., covers, the molecule listed in Fig. 1. For the purposes of this paper, it will be convenient to look at examples as objects and to consider the clausal notation $h(x) \leftarrow f_1, ..., f_n$ where x is a unique identifier of the example. Furthermore, where necessary, we will refer to the head of the example as $h(x)$ and the set of facts in the body as $F(x)$.

We can now introduce the framework of learning from trace kernels. The key difference with the traditional inductive logic programming setting is that the learner is given a set of so-called *visitor* clauses V, which de-

```
mutagenic(225).
molecule(225).
logmutag(225,0.64).
lumo(225,-1.785).
logp(225,1.01).
nitro(225,[f1_4,f1_8,f1_10,f1_9]).
atom(225,f1_1,c,21,0.187).
atom(225,f1_2,c,21,-0.143).
atom(225,f1_3,c,21,-0.143).
atom(225,f1_4,c,21,-0.013).
atom(225,f1_5,o,52,-0.043).
...
```

Figure 1. An example from the mutagenesis domain

fine *visitor* predicates and which replace the hypothesis H. So rather than having to find a set of clauses H, the learner is given a set of clauses V. The idea then is that for each example x, the proofs of the visitor predicates are computed. These proofs then constitute the representation employed by the kernel, which has to learn how to discriminate the set of proofs for a positive example from those of a negative example. The rationale behind the use of the program trace is the idea that not only the success or failure of the goal is of interest in order to characterize a given instance, but also the full trace of steps passed in order to produce such a result. Different visitors can be conceived in order to explore different aspects of the examples and include multiple sources of information.

This idea can be formalized as follows: for each example $(h(x), F(x))$, background theory BK and visitor clauses V defining visitor predicates v_i, we compute the set of proofs $P_i(x) = \{p \mid p \text{ is a proof such that } BK \cup F(x) \cup V \models v_i(x)\}$.

So far, we have not detailed which type of proof or trace is employed. At this point, there are several possibilities. One could employ the SLD-tree, which would not only contain information about succeeding proofs but also about failing ones. The SLD-tree is however a very complex and rather unstructured representation. It is much more convenient to work with *and*-trees for the visitor facts.

An *and-tree* for a query v for an example $(h(x), F(x))$, a background theory BK and visitor clauses V for which $F(x) \cup BK \cup V \models v$ is a tree such that

- v is the root of the tree and
- if v is a fact in $F(x) \cup BK \cup V$ then v is a leaf
- otherwise there must be a clause $w \leftarrow b_1, ..., b_n \in BK \cup V$ and a substitution θ grounding it such that $w\theta = v$ and $BK \cup V \models b_i\theta \forall i$ and there is a subtree of v for each $b_i\theta$ that is an and-tree for $b_i\theta$

The simplest visitor we can imagine just ignores the background knowledge and extracts the ground facts concerning a given example (or a subset of them). Note that visitors actually allow us to expand the example representation as described in (Lloyd, 2003) by naturally including information derived from the background knowledge.

As an example, consider again the mutagenicity benchmark. The following is the atom bond representation of the simple molecule in Figure 2. By looking at the molecule as a graph where atoms are nodes and bonds are edges, we can introduce the common notions of *path* and *cycle*:

```
1 : cycle(E,X):-
      path(E,X,Y,[X]),
      bond(E,Y,X,_).

2 : path(E,X,Y,M):-
      atm(E,X,_,_,_),
      bond(E,X,Y,_),
      atm(E,Y,_,_,_),
      \+ member(Y,M).

3 : path(E,X,Y,M):-
      atm(E,X,_,_,_),
      bond(E,X,Z,_),
      \+ member(Z,M),
      path(E,Z,Y,[Z|M]).
```

A possible visitor in such context would be the one simply looking for a cycle in the molecule, which can be written as:

```
4 : visit(E):
      cycle(E,X).
```

Note that we numbered each clause in $BK \cup V$ (but not in $F(e)$[1]) with a unique identifier. This will allow us to take into account information about the clauses that are used in a proof.

In many situations, the and-tree for a given goal will be unnecessary complex in that it may contain several uninteresting subtrees. To account for this situation, we will often work with *pruned* and-trees, which are trees where subtrees rooted at specific predicates (declared as *leaf* predicates by the user) are turned into leafs. This will allow the kernel to ignore the way atoms involving these predicates are proved. For instance, consider again the molecule in Figure 2, and suppose we have the background knowledge of functional groups as described in (Srinivasan et al., 1996). A potential visitor could look for a benzene ring within the molecule, and eventually find out the details of the

[1] These numbers would change from example to example and hence, would not carry any useful information.

```
atm(d26,d26_1,c,22,-0.093).       bond(d26,d26_1,d26_2,7).
atm(d26,d26_2,c,22,-0.093).       bond(d26,d26_2,d26_3,7).
atm(d26,d26_3,c,22,-0.093).       bond(d26,d26_3,d26_4,7).
atm(d26,d26_4,c,22,-0.093).       bond(d26,d26_4,d26_5,7).
atm(d26,d26_5,c,22,-0.093).       bond(d26,d26_5,d26_6,7).
atm(d26,d26_6,c,22,-0.093).       bond(d26,d26_6,d26_1,7).
atm(d26,d26_7,h,3,0.167).         bond(d26,d26_1,d26_7,1).
atm(d26,d26_8,h,3,0.167).         bond(d26,d26_3,d26_8,1).
atm(d26,d26_9,h,3,0.167).         bond(d26,d26_6,d26_9,1).
atm(d26,d26_10,cl,93,-0.163).     bond(d26,d26_10,d26_5,1).
atm(d26,d26_11,n,38,0.836).       bond(d26,d26_4,d26_11,1).
atm(d26,d26_12,n,38,0.836).       bond(d26,d26_2,d26_12,1).
atm(d26,d26_13,o,40,-0.363).      bond(d26,d26_13,d26_11,2).
atm(d26,d26_14,o,40,-0.363).      bond(d26,d26_11,d26_14,2).
atm(d26,d26_15,o,40,-0.363).      bond(d26,d26_15,d26_12,2).
atm(d26,d26_16,o,40,-0.363).      bond(d26,d26_12,d26_16,2).
```

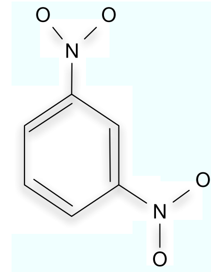

Figure 2. Simple molecule from the mutagenicity benchmark.

atoms involved. In this case it could be convenient to ignore the details of the proof of the ring, provided the atoms involved are extracted. This would be implemented by the predicate `visit_benzene` as follows:

```
1 : atoms(E,[]).            2 : atoms(E,[H|T]):-
                                  atm(E,H,_,_,_),
                                  atoms(E,T).

3 : visit_benzene(E):-
      benzene(E,Atoms),
      atoms(E,Atoms).

leaf(benzene(_,_)).
```

It is important to note that in general a goal can be satisfied in a number of alternative ways. Therefore, a visitor predicate actually generates a (possibly empty) set of proof trees. Furthermore, as we already underlined, different visitors can be conceived in order to analyse different characteristics of the data. An example is thus represented as a tuple of sets of proof trees, obtained by running all the available visitors on it. Given such a representation, we are now able to develop kernels over pairs of examples.

3. Bridging the Gap: Kernels over Logical Objects

Having defined the program traces generated by the visitors, in this section we detail how traces are compared by a kernel over tuples of sets of proof trees.

3.1. Kernels for Discrete Structures

A very general formulation of kernels on discrete structures is that of convolution kernels (Haussler, 1999). Suppose $x \in \mathcal{X}$ is a composite structure made of "parts" $x_1, \ldots, x_D$ such that $x_d \in \mathcal{X}_d$ for all $i \in [1, D]$. This can be formally represented by a relation R on $\mathcal{X}_1 \times \cdots \times \mathcal{X}_D \times \mathcal{X}$ such that $R(x_1, \ldots, x_D, x)$ is true iff $x_1, \ldots, x_D$ are the parts of x. Given a set of kernels $K_d : \mathcal{X}_d \times \mathcal{X}_d \to \mathbb{R}$, one for each of the parts of x, the *R-convolution* kernel is defined as

$$(K_1 \star \cdots \star K_D)(x, z) = \sum_R \prod_{d=1}^{D} K_d(x_d, z_d), \quad (1)$$

where the sum runs over all the possible decompositions of x and z. For finite relations R, this can be shown to be a valid kernel (Haussler, 1999).

A special case of convolution kernel, which will prove useful in defining kernels between proof trees, is the set kernel (Shawe-Taylor & Cristianini, 2004). Provided an object can be represented as a set of simpler objects, we define the part-of relation to be the set-membership, and the kernel reduces to the sum of all pairwise kernels between members:

$$K_{set}(x, z) = \sum_{\xi \in x, \zeta \in z} K_{member}(\xi, \zeta). \quad (2)$$

In order to reduce the dependence on the dimension of the objects, kernels over discrete structures are often normalized. A commmon choice is that of using normalization in feature space, given by:

$$K_{norm}(x, z) = \frac{K(x, z)}{\sqrt{K(x, x)}\sqrt{K(z, z)}}. \quad (3)$$

In the case of set kernels, an alternative is that of dividing by the size of the two sets, thus computing

the mean value between pairwise comparisons:

$$K_{mean}(x, z) = \frac{K_{set}(x, z)}{|x||z|}. \quad (4)$$

This formalism allows us to define a kernel over logical objects as the convolution kernel over the parts in which the objects can be decomposed according to the background knowledge available, provided we are able to define appropriate kernels between individual parts.

3.2. Kernels over Visit Programs

Assume we have a visiting program V made of a number $n \geq 1$ of visitor predicates $v_1, \ldots, v_n$, each producing a (possibly empty) set of proof trees $t_{i,j}(x)$ when tested over an example x. The proof tree representation of x can be written as:

$$P(x) = [P_1(x), \ldots, P_n(x)] \quad (5)$$

where

$$P_i(x) = \{t_{i,1}(x), \ldots, t_{i,h_i(x)}(x)\} \quad (6)$$

and $m_i(x) \geq 0$ is the number of alternative proofs of visitor v_i for example x. Assuming that we do not want to compare proof trees derived from different visitors (but it is straightforward to include such a case), we can define the kernel between examples as:

$$\begin{aligned} K(x, z) &= K_P(P(x), P(z)) \\ &= \sum_{i=1}^{n} K_i(P_i(x), P_i(z)). \end{aligned} \quad (7)$$

By using the definition of set kernel introduced in Section 3.1, we further obtain:

$$K_i(P_i(x), P_i(z)) = \sum_{j=1}^{m_i(x)} \sum_{\ell=1}^{m_i(z)} K(t_{i,j}(x), t_{i,\ell}(z)) \quad (8)$$

The problem boils down to defining the kernel between individual proof trees. Note that we can define different kernels for proof trees originating from different visitors, thus allowing for the greatest flexibility.

At the highest level of kernel between visit programs, we will employ a feature space normalization (eq. (3)). However, it is still possible to normalize lower level kernels, in order to rebalance contributions of individual parts. We will employ a mean normalization (eq. (4)) for the kernel between visitors, and possibly further normalize kernels between individual proof trees, thus reducing the influence of the dimension of proofs.

3.3. Kernels over Proof Trees

Proof trees are discrete data structures and, in principle, existing kernels on trees could be applied (e.g. (Collins & Duffy, 2002; Vishwanathan & Smola, 2002)). However, we can gain more expressiveness by representing individual proof trees as typed Prolog ground terms. In so doing we can exploit type information on constants and functors so that different sub-kernels can be applied to different object types. In addition, while traditional tree kernels would typically compare *all* pairs of subtrees between two proofs, the kernel on ground terms presented below results in a more selective approach that compares certain parts of two proofs only when reached by following similar inference steps, (a distinction that would be difficult to implement with traditional tree kernels).

We will use the following procedure to represent a proof tree as a ground term:

- Nodes corresponding to facts are already ground terms.
- Consider a node corresponding to a clause, with n arguments in the head, and the conjunction of m terms in the body, which correspond to the m children of the node.
  - Let the ground term be a compund term with $n + 1$ arguments, and functor equal to the head functor of the clause.
  - Let the first n arguments be the arguments of the clause head.
  - Let the last argument be a compound term, with functor equal to the clause number[2], and m arguments equal to the ground term representations of the m children of the node.

We are now able to employ kernels on Prolog ground terms as defined in (Passerini & Frasconi, 2005) to compute kernels over individual proof trees. Let us briefly recall the definition of the kernel for typed Prolog ground terms.

We denote by $\mathcal{T}$ the ranked set of type constructors, which contains at least the nullary constructor $\perp$. The type signature of a function of arity n has the form $\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'$ where $n \geq 0$ is the number of arguments, $\tau_1, \ldots, \tau_k \in \mathcal{T}$ their types, and $\tau' \in \mathcal{T}$ the type of the result. Functions of arity 0 have signature $\perp \mapsto \tau'$ and can be therefore interpreted as constants of type τ'. The type of a function is the type of its result. The type signature of a predicate of arity n has the form $\tau_1 \times, \ldots, \times \tau_n \mapsto \Omega$ where $\Omega \in \mathcal{T}$ is the type of booleans, and is thus a special case of type signatures of functions. We write $t : \tau$ to assert that

[2]Actually the number will be prefixed by 'cbody' because Prolog does not allow to use numbers as functors.

t is a term of type τ. We denote by $\mathcal{B}$ the set of all typed ground terms, by $\mathcal{C} \subset \mathcal{B}$ the set of all typed constants, and by $\mathcal{F}$ the set of typed functors. Finally we introduce a (possibly empty) set of *distinguished* type signatures $\mathcal{D} \subset \mathcal{T}$ that can be useful to specify ad-hoc kernel functions on certain compound terms.

Definition 3.1 (Sum Kernels on typed terms) *The kernel between two typed terms t and s is defined inductively as follows:*

- *if $s \in \mathcal{C}$, $t \in \mathcal{C}$, $s : \tau$, $t : \tau$ then $K(s,t) = \kappa_\tau(s,t)$ where $\kappa_\tau : \mathcal{C} \times \mathcal{C} \mapsto I\!R$ is a valid kernel on constants of type τ;*
- *else if s and t are compound terms that have the same type but different arities, functors, or signatures, i.e. $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$, $f : \sigma_1\times, \ldots, \times\sigma_n \mapsto \tau'$, $g : \tau_1\times, \ldots, \times\tau_m \mapsto \tau'$, then*
$$K(s,t) = \iota_{\tau'}(f,g) \tag{9}$$
where $\iota_{\tau'} : \mathcal{F} \times \mathcal{F} \mapsto I\!R$ is a valid kernel on functors that construct terms of type τ'
- *else if s and t are compound terms and have the same type, arity, and functor, i.e. $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, and $f : \tau_1\times, \ldots, \times\tau_n \mapsto \tau'$, then*
$$K(s,t) = \begin{cases} \kappa_{\tau_1\times,\ldots,\times\tau_n\mapsto\tau'}(s,t) \\ \qquad\qquad \text{if } (\tau_1\times, \ldots, \times\tau_n \mapsto \tau') \in \mathcal{D} \\ \iota_{\tau'}(f,f) + \sum_{i=1}^{n} K(s_i,t_i) \quad \text{otherwise} \end{cases} \tag{10}$$
- *in all other cases $K(s,t) = 0$.*

By replacing Equation (10) with
$$K(s,t) = \begin{cases} \kappa_{\tau_1\times,\ldots,\times\tau_n\mapsto\tau'}(s,t) \\ \qquad\qquad \text{if } (\tau_1\times, \ldots, \times\tau_n \mapsto \tau') \in \mathcal{D} \\ \iota_{\tau'}(f,f) \prod_{i=1}^{n} K(s_i,t_i) \quad \text{otherwise} \end{cases} \tag{11}$$
we obtain the *Product Kernel* on typed ground terms. In order to employ such kernels on proof trees, we need a typed syntax for them. We will assume the following default types for constants: `num` (numerical) and `cat` (categorical). Types for compounds terms will be either `fact`, corresponding to leaves in the proof tree, `clause` in the case of internal nodes, and `body` when containing the body of a clause. Note that regardless of the specific implementation of kernels between types, such definitions imply that we actually compare the common subpart of proofs starting from the goal (the visitor clause), and stop whenever the two proofs diverge.

A number of special cases of kernels can be implemented with appropriate choices of the kernel for compound and atomic terms. The *equivalence* kernel outputs one iff two proofs are equivalent, and zero otherwise:
$$K_{equiv}(s,t) = \begin{cases} 1 & \text{if } s \equiv t \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

We say that two proof trees s and t are equivalent iff they have the same number of nodes, and each node is equivalent to its partner in the perfect matching relation between the trees. This can be implemented using the Product Kernel in combination with binary valued kernels, such as the matching one, for kernels on constants and functors , thus implementing the notion of equivalence between individual nodes.

In many cases, we will be interested in ignoring some of the arguments of a pair of ground terms when computing the kernel between them. As an example, consider the atom bond representation in the mutagenicity benchmark, and the background knowledge in the example at the end of Section 2: the argument denoted by `E` indicates the unique identifier of a given molecule, and we would like to ignore its value when comparing two molecules together. This can be implemented using a special *ignore* type for arguments that should be ignored in comparisons, and a corresponding *constant* kernel which always outputs a constant value:
$$K_\eta(s,t) = \eta \tag{13}$$

It is straightforward to see that K_η is a valid kernel provided $\eta \geq 0$. The constant η should be set equal to the neutral value of the operation which is used to combine results for the different arguments of the term under consideration, that is $\eta = 0$ for the sum kernel and $\eta = 1$ for the product one.

The extreme use for this kernel is that of implementing the notion of *functor* equality for nodes, where two nodes are the same iff they share the same functor (and number of arguments), regardless the specific values taken by their arguments. Given two ground terms $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_n)$ the functor equality kernel is given by:
$$K_f(s,t) = \begin{cases} 0 & \text{if } type(s) \neq type(t) \\ \delta(f,g) & \text{if } s,t : \texttt{fact} \\ \delta(f,g) \star K(s_n,t_n) & \text{if } s,t : \texttt{clause} \\ K(s,t) & \text{if } s,t : \texttt{body} \end{cases} \tag{14}$$

where in the internal node case the comparison proceeds on the children, and the operator $\star$ can be either sum or product.

Moreover, it will often be useful to define custom kernels for specific terms, being them clauses or facts, by using distinguished type signatures.

4. Algorithmic Implementation

The algorithm we implemented allows for a high flexibility in customizing the behaviour to match the requirements of the specific task at hand. Four different files should be filled in order to provide the following information:

- The knowledge base describing the data.
- The background knowledge.
- The visit program to be run on the data.
- The specific implementation of kernel over proof trees, as a combination of default behaviours and possibly customized ones.

The first two files are standard in the ILP setting. The visit program is represented as a collection of clauses implementing one or more visitors, together with possible $leaf$ statements aimed at pruning resulting proof trees (see the example at the end of Section 2). Note that it is not necessary to explicitly specify numeric identifiers for clauses, as the program will use the ones automatically provided by Prolog interpreters.

The kernel specification defines the way in which data and knowledge should be treated. The default way of treating compound terms can be declared to be either *sum* or *product*, by writing `compound_kernel(sum)`or `compound_kernel(product)`respectively.

The default atomic kernel is the matching one for symbols, and the product for numbers. Such behaviour can be modified by directly specifying the type signature of a given clause or fact. As an example, the following definition overrides the default kernel between *atm* terms for the mutagenicity problem:

```
type(atm(ignore,ignore,cat,cat,num)).
```

allowing to ignore identifiers for molecule and atom, and change the default behaviour for atom type (which is a number) to categorical.

Default behaviours can also be overriden by defining specific kernels for particular clauses or facts. This corresponds to specifying distinguished types together to appropriate kernels for them. Thus, the kernel between atoms could be equivalently specified by writing[3]:

```
term_kernel(atm(_,_,Xa,Xt,Xc),
            atm(_,_,Ya,Yt,Yc),K):-
  delta_kernel(Xa,Ya,Ka),
  delta_kernel(Xt,Yt,Kt),
  dot_kernel(Xc,Yc,Kc),
  K is Ka + Kt + Kc.
```

A useful kernel which can be selected is the *functor_equality* kernel as defined in Equation (14). For example, by writing

```
term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

at the end of the configuration file it is possible to force the default behaviour for all remaining terms to functor equality, where the combination operator employed for internal nodes will be the one specified with the *compound_kernel* statement.

Finally, hyperparameters must be provided for the particular kernel machine to be run. We employed *gist-svm* [4] as it permits to separate kernel calculation from training by accepting the complete kernel matrix as input. Note that in this phase it is possible to specify kernels other than the linear one (e.g. Gaussian) on top of the visit program kernel, in order to further enlarge the feature space.

In the next section, we will provide a number of experiments showing how to customize the program to the task at hand and providing evidence of the possibilities and limitations of the proposed method.

5. Experiments

5.1. Bongard problems

In order to provide a full basic example of visit program construction, algorithm configuration and exploitation of the proof tree information, we created a very simple Bongard problem (Bongard, 1970). The concept to be learned can be represented with the simple pattern *triangle-*X^n*-triangle* for a given n, meaning that a positive example is a scene containing two triangles nested into one another with exactly n objects (possibly triangles) in between. Figure 3 shows a pair of examples of such scenes with their representa-

[3]Actually, this also allows to possibly override the kernel combination operator specified by the `compound_kernel` statement.

[4]available at `http://microarray.genomecenter.columbia.edu/gist/`

tion as Prolog facts and their classification according to the pattern for $n = 1$.

A possible example of background knowledge introduces the concepts of *nesting* in containment and *polygon* as a generic object, and can be represented as follows:

```
inside(E,X,Y):-
  in(E,X,Y).

inside(E,X,Y):-
  in(E,X,Z),
  inside(E,Z,Y).

polygon(E,X) :-
  triangle(E,X).

polygon(E,X) :-
  rectangle(E,X).

polygon(E,X) :-
  circle(E,X).
```

A visitor exploiting such background knowledge, and having hints on the target concept, could be looking for two polygons contained one into the other. This can be represented as:

```
visit(E):-
  inside(E,X,Y),polygon(E,X),polygon(E,Y).
```

Figure 4 shows the proofs trees obtained running such a visitor on the first Bongard problem in Figure 3.

A very simple kernel can be employed to solve such a task, namely an equivalence kernel with functor equality for nodewise comparison. This can be implemented with the following kernel configuration file:

```
compound_kernel(product).

term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

For any value of n, such a kernel maps the examples into a feature space where there is a single feature discriminating between positive and negative examples, while the simple use of ground facts without background knowledge would not provide sufficient information for the task.

The data set was generated by creating m scenes each containing a series of n randomly chosen objects nested one into the other, and repeating the procedure for n varying from 1 to 19. Moreover, we generated two different data sets by choosing $m = 10$ and $m = 50$ respectively. Finally, for each data set we obtained 15 experimental settings denoted by $n \in [1, 15]$. In each setting, positive examples where scenes containing the pattern *triangle-*X^n*-triangle*. We run an SVM with the above mentioned proof trees kernel and a fixed value $C = 10$ for the regulatization parameter, being the data set noise free. We evaluated its performance with a leave-one-out procedure, and compared it to *Tilde* (Blockeel & Raedt, 1997) trained from the same data and background knowledge (including the visitor).

Results are plotted in Figure 5(a) and 5(b) for $m = 10$ and $m = 50$ respectively. Both methods obtained better performance for bigger data sets, but SVM performance was very stable when increasing the nesting level corresponding to positive examples, whereas Tilde was not able to learn the concept for $n > 5$ when $m = 10$, and $n > 9$ when $m = 50$.

5.2. Strings

The possibility to plug background knowledge into the kernel allows to address problems which are notoriously hard for ILP approaches. An example of such concepts is the *M of N* one, which expects the model to be able to count and make the decision according to the result of such count.

We represented this kind of tasks with a toy problem. Examples are strings of integers $i \in [0, 9]$, and a string is positive iff more than a half of its pairs of consecutive elements is ordered, where we employ the partial ordering relation $\leq$ between numbers. In this task, M and N are example dependent, while their ratio is fixed.

As background knowledge, we introduced the concepts of length two substring and ordering between pairs of elements:

```
substr([],_):-fail.
substr(_,[]):-fail.
substr([A,B],[A,B|_T]).
substr([A,B],[_H|T]):-
  substr([A,B],T).

comp(A,B):-
  A @> B.
comp(A,B):-
  A @=< B.
```

while the visitor actually looks for a substring of length two in the example, and compares its elements:

```
visit(E):-
  string(E,S),substr([A,B],S),comp(A,B).

leaf(substr(_,_)).
```

Note that we state *substr* is a leaf, because we are not interested in where the substring is located within the example.

The kernel we employed for this task is a sum kernel with functor equality for nodewise comparison. This can be implemented with the following kernel configuration file:

```
compound_kernel(sum).
```

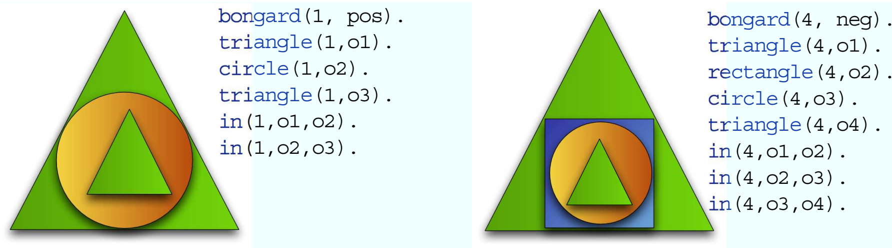

Figure 3. Graphical and Prolog facts representation of two Bongard scenes. The left and right examples are positive and negative, respectively, according to the pattern *triangle-X-triangle*.

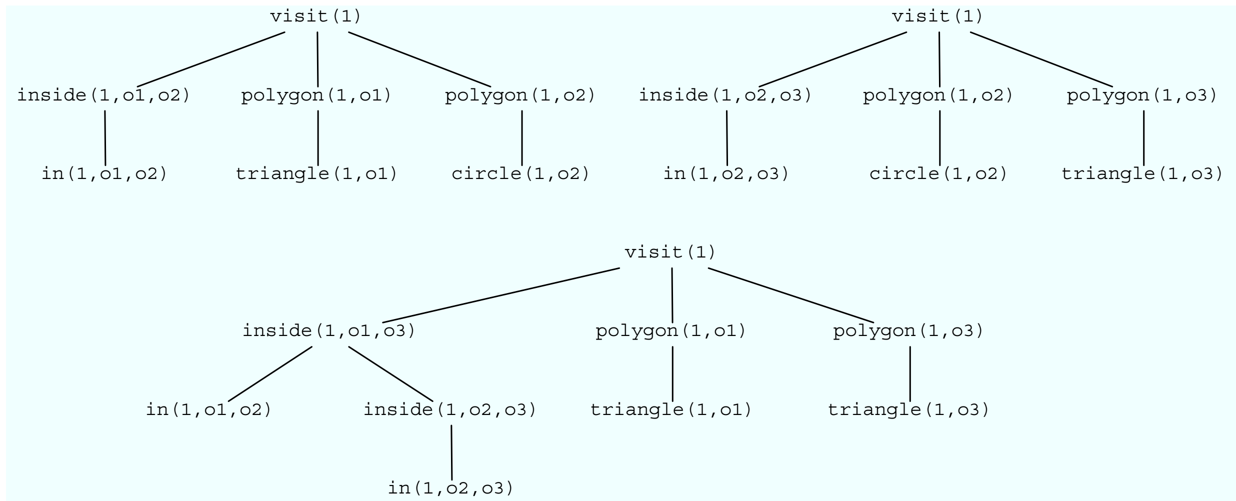

Figure 4. Proof trees obtained by running the visitor on the first Bongard problem in Figure 3.

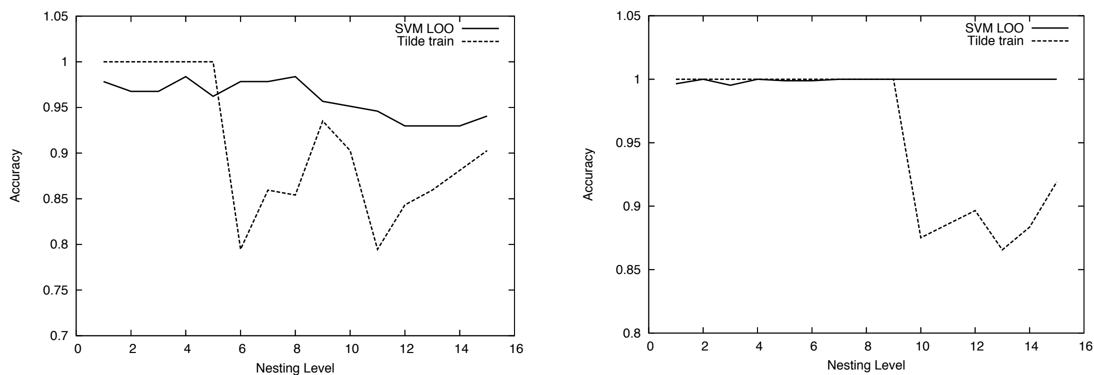

Figure 5. Comparison between SVM and Tilde in learning the *triangle-*X^n*-triangle* for different values of n, for data sets corresponding to $m = 10$ (left) and $m = 50$ (right).

```
term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

The data set was created in the following way: the training set was made of 150 randomly generated lists of length 4 and 150 lists of length 5; the test set was made of 1455 randomly generated lists of length from 6 to 100. This allowed to verify the generalization performances of the algorithm for lengths very different from the ones it was trained on. The area under the ROC curve (Bradley, 1997) on the test set was equal to 1, showing that the concept had been perfectly learned by the algorithm.

5.3. Mutagenicity

The mutagenicity problem described in (Srinivasan et al., 1996) is a standard benchmark for ILP approaches. Background theory is represented as number of clauses looking for functional groups, such as benzene or anthracene, within a molecule. As a baseline we used a visitor looking for paths of different lenghts within the molecule, thus ignoring the notion of functional groups:

```
path(Drug,1,X,Y,M):-
  atm(Drug,X,_,_,_),bond(Drug,X,Y,_),
  atm(Drug,Y,_,_,_),\+ member(Y,M).

path(Drug,L,X,Y,M):-
  atm(Drug,X,_,_,_),bond(Drug,X,Z,_),
  \+ member(Z,M),L1 is L - 1,
  path(Drug,L1,Z,Y,[Z|M]).

visit1(Drug):-
  path(Drug,1,X,_,[X]).
  .
  .
visit5(Drug):-
  path(Drug,5,X,_,[X]).
```

the kernel compared atoms and bonds in corresponding positions for paths of same length:

```
compound_kernel(sum).

type(atm(ignore,ignore,cat,cat,num)).
type(bond(ignore,ignore,ignore,cat)).

term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

A more complex notion of similarity would be to compare atoms belonging to the same type of functional group, according to the background knowledge available. This was implemented with the following set of visitors:

```
atoms(Drug,[]).

atoms(Drug,[H|T]):-
  atm(Drug,H,_,_,_),atoms(Drug,T).

visit_benzene(Drug):-
  benzene(Drug,Atoms),
  atoms(Drug,Atoms).

visit_anthracene(Drug):-
  anthracene(Drug,[Ring1,Ring2,Ring3]),
  atoms(Drug,Ring1),atoms(Drug,Ring2),
  atoms(Drug,Ring3).
  .
  .
visit_ring_size_5(Drug):-
  ring_size_5(Drug,Atoms),
  atoms(Drug,Atoms).

leaf(benzene(_,_)).
leaf(anthracene(_,_)).
  .
  .
leaf(ring_size_5(_,_)).
```

and corresponding kernel configuration:

```
compound_kernel(sum).

type(atm(ignore,ignore,cat,cat,num)).

term_kernel(X,Y,K):-
  functor_equality_kernel(X,Y,K).
```

Note that we are not interested in the way the presence of a functional group is proved, but simply on the characteristics of the atoms belonging to it. Finally, an additional source of information is given by some non structural attributes, which were included using a visitor which simply reads them

```
visit_global(Drug):-
  lumo(Drug,_Lumo),
  logp(Drug,_Logp).
```

and a kernel configuration like

```
type(lumo(ignore,num)).
type(logp(ignore,num)).
```

to be added before the last statement for the default functor equality kernel.

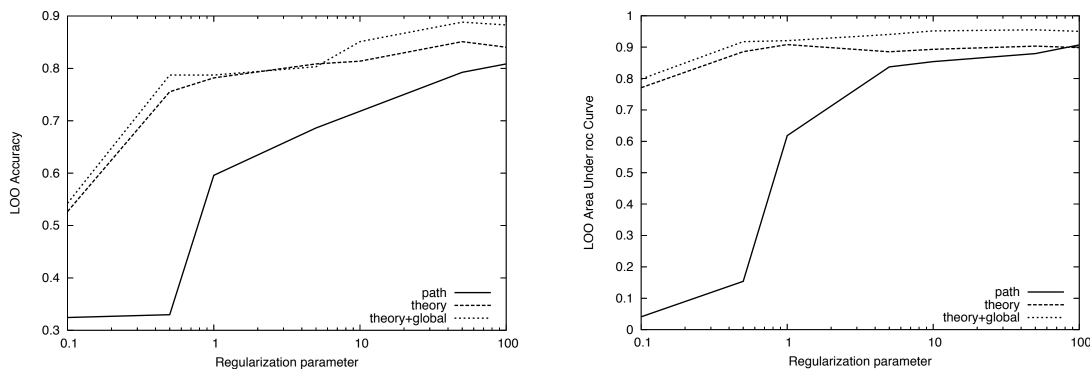

Figure 6. LOO accuracy (left) and AUC (right) for the regression friendly mutagenesis data set using different types of visitors/kernels.

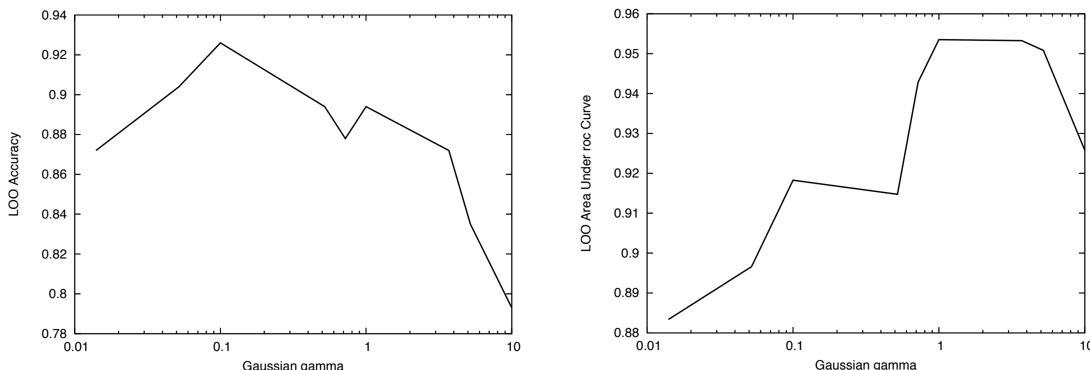

Figure 7. LOO accuracy (left) and AUC (right) for the regression friendly mutagenesis data set using the *theory+global* visitor/kernel, a Gaussian kernel on top of it, $C = 50$ and different values for the Gaussian width.

We used the regression friendly data set of 188 molecules with a LOO procedure to evaluate the methods, and both accuracy and area under the ROC curve (AUC) as performance measures. Figures 6(a) and 6(b) report LOO accuracy and AUC for different values of the regularization parameter C, for *path*, *theory* and *theory+global* visitors and corresponding kernels. Note that performances could be further improved by composing additional kernels on top of the visit program one. As an example, Figure 7(a) and 7(b) report LOO accuracy and AUC when using a Gaussian kernel on top of the *theory+global* kernel, with a fixed parameter $C = 50$ (tuned on the non composed kernel), and different values for the Gaussian width.

6. Conclusions

We have introduced the general idea of kernels over program traces and specialized it to the case of Prolog proof trees in the logic programming paradigm. The theory and the experimental results that we have obtained indicate that this method can be seen as a successful attempt to bridge several important aspects of symbolic and statistical learning, including the ability of working with relational data, the incorporation of background knowledge in a flexible and principled way, and the use of kernel methods. Besides the case of classification that has been studied in this paper, other learning tasks could benefit from the proposed framework including regression, clustering, ranking, and novelty detection. One advantage of ILP as compared to the present work is the intrinsic ability of inductive logic programming to *generate* transparent explanations of the learned function. We are currently investigating the possibility to use the kernel in guiding program synthesis or refinement, for example by learning to change the default order of Prolog resolution looking at the traces of successful and unsuccessful proofs.

Acknowledgements

This research is supported by EU Grant APrIL II (contract n° 508861). PF and AP are also partially sup-

ported by MIUR Grant 2003091149_002.

References

Biermann, A., & Krishnaswamy, R. (1976). Constructing programs from example computations. *IEEE Transactions on Software Engineering*, *2*, 141–153.

Blockeel, H., & Raedt, L. D. (1997). *Top-down induction of logical decision trees* (Technical Report CW 247). Dept. of Computer Science, K.U.Leuven.

Bongard, M. (1970). *Pattern recognition*. Spartan Books.

Bradley, A. (1997). The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, *30*, 1145–1159.

Collins, M., & Duffy, N. (2002). New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. *Proceedings of ACL 2002* (pp. 263–270). Philadelphia, PA, USA.

Cortes, C., Haffner, P., & Mohri, M. (2004). Rational kernels: Theory and algorithms. *Journal of Machine Learning Research*, *5*, 1035–1062.

Cumby, C. M., & Roth, D. (2002). Learning with feature description logics. *Proc. of ILP'02* (pp. 32–47). Springer-Verlag.

Cumby, C. M., & Roth, D. (2003). On kernel methods for relational learning. *Proc. of ICML'03*.

Gärtner, T. (2003). A survey of kernels for structured data. *SIGKDD Explor. Newsl.*, *5*, 49–58.

Gärtner, T., Lloyd, J., & Flach, P. (2004). Kernels and distances for structured data. *Machine Learning*, *57*, 205–232.

Haussler, D. (1999). *Convolution kernels on discrete structures* (Technical Report UCSC-CRL-99-10). University of California, Santa Cruz.

Jaakkola, T., & Haussler, D. (1998). Exploiting generative models in discriminative classifiers. *Proc. of NIPS*.

Kramer, S., Lavrac, N., & Flach, P. (2000). Propositionalization approaches to relational data mining. In *Relational data mining*, 262–286. SV, NY.

Leslie, C., Eskin, E., & Noble, W. (2002). The spectrum kernel: a string kernel for svm protein classification. *Proc. of the Pac. Symp. Biocomput.* (pp. 564–575).

Lloyd, J. (2003). *Logic for learning: learning comprehensible theories from structured data*. Springer-Verlag.

Lodhi, H., Shawe-Taylor, J., Cristianini, N., & Watkins, C. (2000). Text classification using string kernels. *NIPS 2000* (pp. 563–569).

Mitchell, T. M., Utgoff, P. E., & Banerj, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine learning: An artificial intelligence approach*, vol. 1. Morgan Kaufmann.

Passerini, A., & Frasconi, P. (2005). Kernels on prolog ground terms. *Int. Joint Conf. on Artificial Intelligence (IJCAI'05)*. Edinburgh.

Schölkopf, B., & Warmuth, M. (Eds.). (2003). *Kernels and regularization on graphs*, vol. 2777 of *Lecture Notes in Computer Science*. Springer.

Shapiro, E. (1983). *Algorithmic program debugging*. MIT Press.

Shawe-Taylor, J., & Cristianini, N. (2004). *Kernel methods for pattern analysis*. Cambridge University Press.

Srinivasan, A., Muggleton, S., Sternberg, M. J. E., & King, R. D. (1996). Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, *85*, 277–299.

Sterling, L., & Shapiro, E. (1994). *The art of prolog: Advanced programming techniques*. MIT Press. 2nd edition.

Vishwanathan, S., & Smola, A. (2002). Fast kernels on strings and trees. *NIPS 2002*.

Zelle, J. M., & Mooney, R. J. (1993). Combining FOIL and EBG to speed-up logic programs. *Proc. of IJCAI-93* (pp. 1106–1111).

Learning Recursive Prolog Programs with Local Variables from Examples

M. R. K. Krishna Rao KRISHNA@CCSE.KFUPM.EDU.SA
Information and Computer Science Department King Fahd University of Petroleum and Minerals, Dhahran 31261, Saudi Arabia.

Abstract

Logic programs with elegant and simple declarative semantics have become very common in many areas of artificial intelligence such as knowledge acquisition, knowledge representation and common sense and legal reasoning. For example, in Human GENOME project, logic programs are used in the analysis of amino acid sequences, protein structure and drug design etc. In this paper, we investigate the problem of learning logic (Prolog) programs from examples and present an inference algorithm for a class of programs. This class of programs (called one-recursive programs) is based on the divide-and-conquer approach and mode/type annotations. Our class is very rich and includes many programs from Sterling and Shapiro's book [33] including `append, merge, split, delete, insert, insertion-sort, preorder` and `inorder` traversal of binary trees, polynomial recognition, derivatives, sum of a list of natural numbers etc., whereas earlier results can only deal with very simple programs without local variables and at most two clauses and one predicate [4].

1. Introduction

The theory of inductive inference attempts to understand the all pervasive phenomena of learning from examples and counterexamples. Starting from the influential works of Gold [12] and Blum and Blum [5], a lot of effort has gone into the development of a rich theory about inductive inference and the classes of concepts which can be learned from both positive (examples) and negative data (counterexamples) and the classes of concepts which can be learned from positive data alone. The study of inferability from positive data alone is important because negative examples are hard to obtain in practice.

Logic programs with simple and elegant declarative semantics can be used as representations of the concepts to be learned. In fact, the problem of learning logic programs from examples has attracted a lot of attention (a.o. [3,4,7,8,10,11,13-16,18-20,22-25,28,29,35]) starting with the seminal work of Shapiro [30, 31] and many techniques and systems for learning logic programs are developed and used in many applications. See [24] for a recent survey.

The existing literature mainly concerns with either nonrecursive programs or recursive programs without local variables, usually with a further restriction that programs contain a unit clause and at most one recursive clause with just one atom in the body. It is a well-known fact that local variables in logic programs play an important role in *sideways information passage*. However, presence of local variables pose a few difficulties in analyzing and learning programs. Moding annotations and linear inequalities have been successfully applied in the literature (cf. [34, 27, 17, 2]) to tame these difficulties in analyzing logic programs with local variables (in particular, termination and occur-check aspects). In this paper, we demonstrate that moding/typing annotations and linear inequalities are useful in learning logic programs as well.

As established by many authors in the literature, learning recursive logic programs, even with the above restrictions, is a very difficult problem. We approach this problem from a programming methodology angle and propose an algorithm to learn a class of Prolog programs, that use divide-and-conquer methodology. Our endeavour is to develop an inference algorithm that learns a very natural class of programs so that it will be quite useful in practice. We measure the naturality of a class of programs in terms of the number of programs it covers from a standard Prolog book such as [33]. We use the inference criterion proposed by An-

gluin [1]: consistent and conservative identification in the limit from positive data with polynomial time in updating conjectures. That is, the program guessed by the algorithm is always consistent with the examples read so far and changes its guess only when the most recently read example is not consistent with the current guess and it updates its guess in polynomial time in the size of the current sample of examples read so far.

2. Preliminaries

We assume that the reader is familiar with the basic terminology of logic programming and inductive inference and use the standard terminology from [21, 24, 12]. We are primarily interested in programs operating on the following recursive types used in Sterling and Shapiro [33].[1]

```
Nat ::= 0 | s(Nat)
List ::= [ ] | [item | List]
ListNat ::= [ ] | [Nat | ListNat]
Btree ::= void | tree(Btree, item, Btree)
```

Definition 1 A term t is a *generic expression* for type T if for every $s \in T$ disjoint with t the following property holds: *if* s *unifies with* t *then* s *is an instance of* t.

For example, a variable is a generic expression for every type T, and [ ], [X], [H|T], [X, Y|Z], $\cdots$ are generic expressions for the type List. Note that a generic expression for type T need not be a member of T — e.g., term $f(X)$ is a generic expression for the type List.

Notation:

1. We call the terms 0, [ ] and void the constants of their respective types, and call the subterms T_1 and T_2 *recursive subterms of* term (or generic-expression) of the form tree(T_1, X, T_2) of type Btree. Similarly, L is the recursive subterm of term (or generic-expression) of the form [H|L] of type List.

2. The generic-expression 0 (resp. [ ] and void) is called the first generic-expression of type Nat (resp. List and Btree). The generic-expression s(X) (resp. [H|L] and tree(T_1, X, T_2)), which generalizes all the other terms of type Nat (resp. List and Btree) is called the second generic-expression of type Nat (resp. List and Btree).

[1] Though we only consider Nat, List, ListNat and Btree in the following, any other recursive type can be handled appropriately.

Remark: Note that the first and second generic-expressions of a given recursive type are unique upto variable renaming.

Definition 2 For a term t, the *parametric size* $[t]$ of t is defined recursively as follows:

- if t is a variable X then $[t]$ is a linear expression X,
- if t is the empty list [ ] or the natural number 0 or the empty tree void then $[t]$ is zero,
- if $t = f(t_1, \ldots, t_n)$ and $f \in \Sigma - \{0, [\,], \texttt{void}\}$ then $[t]$ is a linear expression $1 + [t_1] + \cdots + [t_n]$.

The parametric size of a sequence $\mathbf{t}$ of terms $t_1, \cdots, t_n$ is the sum $[t_1] + \cdots + [t_n]$.

The *size* of a term t, denoted by $|t|$, is defined as $[t]\theta$, where θ substitutes 1 for each variable. The *size* of an atom $p(t_1, \cdots, t_n)$ is the sum of the sizes of terms $t_1, \cdots, t_n$.

Example 1 *The parametric sizes of terms* [ ], [X], [a] *and* [a, b, c] *are* 0, $X + 1$, 2, *and* 6 *respectively. Their sizes are* 0, 2, 2, *and* 6 *respectively.*

Remark: In general, the size of a list (or binary tree) with n elements is $2n$. This is similar to the measures used in the termination analysis of logic programs by Plümer [27] in the sense that size of a term is proportional to its contents.

3. Linearly-moded programs

Using moding annotations and linear predicate inequalities, Krishna Rao [18] introduced the following class of programs and proved a theoretical result that this class is inferable from positive examples alone.

Definition 3 A *mode* m of an n-ary predicate p is a function from $\{1, \cdots, n\}$ to the set $\{in, out\}$. The sets $in(p) = \{j \mid m(j) = in\}$ and $out(p) = \{j \mid m(j) = out\}$ are the sets of input and output positions of p respectively.

A moded program is a logic program with each predicate having a unique mode associated with it. In the following, $p(\mathbf{s}; \mathbf{t})$ denotes an atom with input terms $\mathbf{s}$ and output terms $\mathbf{t}$.

Definition 4 Let P be a moded program and I be a mapping from the set of predicates occurring in P to sets of input positions satisfying $I(p) \subseteq in(p)$ for each predicate p in P. For an atom $A = p(\mathbf{s}; \mathbf{t})$, the linear inequality

$$\sum_{i \in I(p)} [s_i] \geq \sum_{j \in out(p)} [t_j] \qquad (1)$$

is denoted by $LI(A, I)$.

Definition 5 A moded program P is *linearly-moded w.r.t. a mapping* I such that $I(p) \subseteq in(p)$ for each predicate p in P, if each clause

$$p_0(\mathbf{s_0};\mathbf{t_0}) \leftarrow p_1(\mathbf{s_1};\mathbf{t_1}), \cdots, p_k(\mathbf{s_k};\mathbf{t_k})$$

$k \geq 0$, in P satisfies the following:

1. $LI(A_1, I), \ldots, LI(A_{j-1}, I)$ together imply $[\mathbf{s_0}] \geq [\mathbf{s_j}]$ for each $j \geq 1$, and
2. $LI(A_1, I), \ldots, LI(A_k, I)$ together imply $LI(A_0, I)$,

where A_j is the atom $p_j(\mathbf{s_j};\mathbf{t_j})$ for each $j \geq 0$.
A program P is *linearly-moded* if it is linearly-moded w.r.t. some mapping I.

Example 2 *Consider the following* `reverse` *program.*
`moding: app(in,in, out)` *and* `rev(in, out)`.

```
app([ ], Ys, Ys) ←
app([X|Xs], Ys, [X|Zs]) ← app(Xs, Ys, Zs)

rev([ ], [ ]) ←
rev([X|Xs], Zs) ← rev(Xs, Ys), app(Ys, [X], Zs)
```

This program is linearly-moded *w.r.t. the mapping* $I(\texttt{app}) = in(\texttt{app})$; $I(\texttt{rev}) = in(\texttt{rev})$. *For lack of space, we only prove this for the last clause.* $LI(\texttt{rev(Xs, Ys)}, I)$ *is*

$$Xs \geq Ys, \tag{2}$$

$LI(\texttt{app(Ys, [X], Zs)}, I)$ *is*

$$Ys + 1 + X \geq Zs \tag{3}$$

and $LI(\texttt{rev([X|Xs], Zs)}, I)$ *is*

$$1 + X + Xs \geq Zs. \tag{4}$$

It is easy to see that inequalities 2 and 3 together imply inequality 4 satisfying the requirement 2 of Definition 5. The requirement 1 of Definition 5 holds for atoms `rev(Xs, Ys)` *and* `app(Ys, [X], Zs)` *as follows:* $1 + X + Xs \geq Xs$ *trivially holds for atom* `rev(Xs, Ys)`. *For atom* `app(Ys, [X], Zs)`, *inequality 2 implies* $1 + X + Xs \geq Ys + 1 + X$.

The class of linearly-moded programs is very rich and contains many standard programs such as `split`, `merge`, `quick-sort`, `merge-sort`, `insertion-sort` and various tree traversal programs.

4. One-Recursive Programs

To facilitate efficient learning of programs, we restrict our attention to a subclass of the class of linearly-moded programs. In particular, we consider well-typed programs [6]. The divide-and-conquer approach and recursive subterms are the two central themes of our class of programs. The predicates defined by these programs are recursive on the leftmost argument. The leftmost argument of each recursive call invoked by a caller is a recursive subterm of the arguments of the caller. In the following, *builtins* is a (possibly empty) sequence of atoms with built-in predicates having no output positions.

Definition 6 (**One-recursive programs**)
A linearly-moded well-typed Prolog program without mutual recursion is *one-recursive* if each clause in it is of the form

$p(\mathbf{s}_0;\mathbf{t}_0) \leftarrow builtins,\ p(\mathbf{s}_1;\mathbf{t}_1), \cdots, p(\mathbf{s}_k;\mathbf{t}_k)$
or
$p(\mathbf{s}_0;\mathbf{t}_0) \leftarrow builtins, p(\mathbf{s}_1;\mathbf{t}_1), \cdots, p(\mathbf{s}_k;\mathbf{t}_k), q(\mathbf{s};\mathbf{t})$

such that (a) $\mathbf{s}_i$ is same as $\mathbf{s}_0$ except that the leftmost term in $\mathbf{s}_i$ is a recursive subterm of the leftmost term in $\mathbf{s}_0$ for each $1 \leq i \leq k$, (b) the terms in $\mathbf{s}_0$ are variables or one of the first two generic-expressions of the asserted types and $|\mathbf{s}_0| \geq |\mathbf{t}_0|$ and (c) the terms in $\mathbf{t}_i,\ i \geq 1$ are distinct variables not occuring in $\mathbf{s}_0$.

It is easy to see that all the above conditions can be checked in linear time by scanning the program once.

Theorem 1 *Whether a well-typed program P is one-recursive or not can be checked in polynomial (over the size of the program) time.*

The following example illustrates the divide-and-conquer nature of one-recursive programs.

Example 3 Consider the following program for preorder traversal of binary trees.

```
mode/type:  preorder(in:Btree,out:List) and
       app (in:List, in:List, out:List)

app([ ], Ys, Ys) ←
app([X|Xs], Ys, [X|Zs]) ← app(Xs, Ys, Zs)

preorder(void, [ ]) ←
preorder(tree(T1, X, T2), [X|L]) ←
       preorder(T1, L1), preorder(T2, L2),
       app(L1, L2, L)
```

It is easy to see that this program is well-typed, linearly-moded and one-recursive.

A typical one-recursive clause
$p(\mathbf{s}_0;\mathbf{t}_0) \leftarrow builtins, p(\mathbf{s}_1;\mathbf{t}_1), \cdots, p(\mathbf{s}_k;\mathbf{t}_k), q(\mathbf{s};\mathbf{t})$
satisfies (1) $|\mathbf{s}\sigma| \geq |\mathbf{t}\sigma|$ for every substitution σ such that $p(\mathbf{s}_0;\mathbf{t}_0)\sigma, p(\mathbf{s}_1;\mathbf{t}_1)\sigma, \cdots, p(\mathbf{s}_k;\mathbf{t}_k)\sigma,\ q(\mathbf{s};\mathbf{t})\sigma$ are atoms in the minimal Herbrand model and (2) $[\mathbf{t}_0] \leq [\mathbf{s}_0, \mathbf{t}_1, \cdots, \mathbf{t}_n, \mathbf{t}]$. These properties form the basis for **Step Aux** in the inference algorithm given below.

Remark: The class of one-recursive programs is different from the class of linear-recursive programs studied in Cohen [7, 8]. Linear-recursive programs allow at most one recursive atom in the body of a clause, whereas one-recursive programs allow more than one recursive atoms in the body of a clause.

5. Algorithm for generating one-recursive programs

In this section, we give an inference algorithm to derive one-recursive programs from positive presentations. We only consider programs satisfying the following conditions: (1) programs are deterministic such that the least Herbrand model of a program do not contain two different atoms $p(\mathbf{s};\mathbf{t_1})$ and $p(\mathbf{s};\mathbf{t_2})$ with the same input terms, (2) heads of no two clauses are same (even after renaming) and (3) non-recursive clauses have only builtin atoms in the body. These conditions are obeyed by almost all the programs given in Sterling and Shapiro [33].

We need the following concepts in describing our algorithm. An atom A is a most specific generalization (or *msg*) of a set S of atoms if (a) each atom is in S is an instance of A and (b) A is an instance of any other atom B satisfying condition (a). It is well known that *msg* of S can be computed in polynomial time in the total size of atoms in S [26]. In view of the restrictions placed on the atoms in one-recursive programs, it is some times desirable to have more than one atoms (in a particular form) to cover a set S of atoms.

In the following, we assume that the type of the leftmost argument of the target predicate p has n recursive subterms and our recursive clauses are of the form $p(s,\cdots) \leftarrow builtins,\ p(s_1,\cdots),\cdots,p(s_n,\cdots)$ or $p(s,\cdots) \leftarrow builtins,\ p(s_1,\cdots),\cdots,p(s_n,\cdots),\ q(\cdots)$, where $s_1,\cdots,s_n$ are the recursive subterms of s. Two atoms $Pat_1 \equiv p(u_1,\mathbf{u})$ and $Pat_2 \equiv p(u_2,\mathbf{u})$ are called the first two patterns of the target predicate p if (a) u_1 and u_2 are the first two generic-expressions of the asserted type of the leftmost argument of p and (b) $\mathbf{u}$ is a sequence of distinct variables.

Procedure **Infer-one-recursive**;
begin $P := \phi$; $S := \phi$;
Read examples into S until it contains an atom whose leftmost argument has instances of the second generic-expression as recursive subterms, and all the atoms with recursive subterms of this argument as leftmost arguments. That is, if the asserted type of the leftmost position of the target predicate is `List`, read the examples into S until S contains an atom with a list L of at least two elements in the first argument and all the atoms which have sublists of L as first argument.
If an example $p(\mathbf{s};\mathbf{t})$ with $|\mathbf{s}| < |\mathbf{t}|$ is encountered, exit with error message *no linearly-moded program.*
repeat
  Read example $A \equiv p(\mathbf{s};\mathbf{t})$ into S;
  if $|\mathbf{s}| < |\mathbf{t}|$ **then** exit with error(*no LM program*);
  if A is inconsistent with P **then** $P :=$ **Generate**(S);
  if $P =$ false **then** exit with error(*no LM program*)
forever
end;

We say a ground atom $p(\mathbf{s};\mathbf{t})$ is incompatible with a clause $p(\mathbf{u};\mathbf{v}) \leftarrow builtins$ if there is a substitution σ such that (1) *builtins* hold for substitution σ, (2) $\mathbf{s} \equiv \mathbf{u}\sigma$ and (3) $\mathbf{t} \not\equiv \mathbf{v}\sigma$.

Function **Generate**(S);
begin $P := \phi$;
$S1 := \{B \in S \mid$ B is an instance of $Pat_1\}$;
$S2 := \{B \in S \mid$ B is an instance of $Pat_2\}$;
Step 1: $P := P \cup$**Non-rec**($S1$);
Step 2: $P := P \cup$**Non-rec**($S2$);
Step 3: % Recursive clauses. %
Let $S3$ be the set of atoms in $S2$ which are not covered by the clauses added in **Step 2**;
if $S3 \neq \phi$ **then**
begin
Let *builtin*3 be the sequence of builtin-atoms complementing the builtin-atoms of the clauses added in Step 2;
Compute **the** *msg* $p(\mathbf{s}_0;\mathbf{t}_0)$ of $S3$;
Consider the following one-recursive clause:
  $p(\mathbf{s}_0;\mathbf{t}_0) \leftarrow builtin3, p(\mathbf{s}_1;\mathbf{t}_1),\cdots,p(\mathbf{s}_n;\mathbf{t}_n)$;
Let T be the set of tuples $\{\langle \mathbf{s}_0\sigma, \mathbf{t}_1\sigma,\cdots,\mathbf{t}_n\sigma,\mathbf{t}_0\sigma\rangle$ such that $p(\mathbf{s}_0;\mathbf{t}_0)\sigma \in S3$ and $p(\mathbf{s}_i;\mathbf{t}_i)\sigma \in S$ for each $1 \leq i \leq n\}$;
Get a set $T2$ of *msg*'s of the form $\langle \mathbf{s}_0,\mathbf{t}_1,\cdots,\mathbf{t}_n,\mathbf{t}_0\theta\rangle$ covering **all** the tuples in T such that
  (a) $[\mathbf{t}_0\theta] \leq [\mathbf{s}_0,\mathbf{t}_1,\cdots,\mathbf{t}_n]$ and
  (b) $|\mathbf{s}_0| \geq |\mathbf{t}_0\theta|$;
if $T2$ is a singleton set **then** $P := P \cup \{C\}$ where C is $p(\mathbf{s}_0;\mathbf{t}_0\theta) \leftarrow builtin3, p(\mathbf{s}_1;\mathbf{t}_1),\cdots,p(\mathbf{s}_n;\mathbf{t}_n)\}$
elsif $|T2| = m > 1$ **then** form m clauses with additional built-in atoms and add them to P
elsif $T2 = \phi$ **then**
begin
Step Aux: % Add auxiliary predicate. %
Let $T3$ be the set of atoms of the form $q(\mathbf{u};\mathbf{v})$ such that
  (1) $|\mathbf{u}\sigma| \geq |\mathbf{v}\sigma|$ for each σ such that $p(\mathbf{s}_0;\mathbf{t}_0)\sigma \in S3$ and $p(\mathbf{s}_i;\mathbf{t}_i)\sigma \in S$ for each $1 \leq i \leq n$,
  (2) $LI(A_1,I),\ldots,LI(A_n,I)$ together imply $[\mathbf{s}_0] \geq [\mathbf{u}]$ where $A_i \equiv p(\mathbf{s_i};\mathbf{t_i})$ and
  (3) there is a θ such that $[\mathbf{t}_0\theta] \leq [\mathbf{s}_0,\mathbf{t}_1,\cdots,\mathbf{t}_n,\mathbf{v}]$ and $|\mathbf{s}_0| \geq |\mathbf{t}_0\theta|$;
Flag := false;
while not Flag and $T3 \neq \phi$ **do**
begin
Pick an atom $A \equiv q(\mathbf{u};\mathbf{v}) \in T3$;
$T3 := T3 - \{q(\mathbf{u};\mathbf{v})\}$;
Let $T4$ the set of atoms $\{A\sigma$ such that $p(\mathbf{s}_0;\mathbf{t}_0)\sigma \in S3$ and $p(\mathbf{s}_i;\mathbf{t}_i)\sigma \in S$ for each $1 \leq i \leq n\}$;
AuxP := **Generate**($T4$);
if AuxP $\neq$ false **then** Flag := true

end;
if Flag = false **then** Return(false)
else $P := P \cup AuxP \cup \{C1\}$ where $C1$ is
$p(\mathbf{s_0}; \mathbf{t_0}\theta) \leftarrow builtin3,\ p(\mathbf{s}_1; \mathbf{t}_1), \cdots, p(\mathbf{s}_n; \mathbf{t}_n), q(\mathbf{u}; \mathbf{v})$
end;
end;
Return(P)
end Generate;

Function **Non-rec**(S);
begin $P := \phi$;
Get a set of *msg*'s of the form $p(\mathbf{s}; \mathbf{t})$ for S such that $[\mathbf{t}] \leq [\mathbf{s}]$.
for each msg $p(\mathbf{s}; \mathbf{t})$ **do**
if no atom in S is incompatible with unit clause $p(\mathbf{s}; \mathbf{t}) \leftarrow$
then $P := P \cup \{p(\mathbf{s}; \mathbf{t}) \leftarrow\}$
else try to get a clause $p(\mathbf{s}; \mathbf{t}) \leftarrow builtin\ atoms$ without any incompatible atom in S (if possible) and add it to P;
Return(P)
end Non-rec;

It may be noted that the clauses returned by **Non-rec** for input $S1$ cover all the examples in $S1$ for the following reasons: (1) as the leftmost argument of Pat_1 has no recursive arguments, no recursive clauses can be considered, (2) all the atoms in $S1$ are covered by the clauses of the form $p(\mathbf{s}; \mathbf{t}) \leftarrow builtins$ and (3) since $S1$ is a part of the positive presentation of a linearly-moded program, $[\mathbf{t}] \leq [\mathbf{s}]$ holds.
However, the clauses returned by **Non-rec** for input $S2$ need not cover all the examples in $S2$ as shown by the following example. In fact, this is expected, as **Non-rec** only generates unit clauses or clauses with just built-in atoms in the body, while most of the problems need recursive clauses.

Example 4 Let us consider inference of a program `del` for deleting all the occurrence of a given element from a list. The relevant mode/type annotation is `del(in:List, in:Item; out:List)`. The patterns to consider are: `del([ ], Y; Zs)` and `del([X|Xs], Y; Zs)`.

Consider the invocation of **Generate** with examples: `del([ ], 1; [ ])`, `del([ ], 2; [ ])`, `del([1], 1; [ ])`, `del([2], 1; [2])`, `del([2,1], 1;, [2])`, `del([1,2], 1; [2])`, `del([1,2,3], 1; [2,3])`, `del([1,2,1], 1; [2])`.
From the first pattern and examples `del([ ], 1; [ ])`, `del([ ], 2; [ ])`, we get a unit clause
`del([ ], Y; [ ])` ←.

Consider step 2 now. Only *msg*'s to be considered by **Non-rec**(S2) are `del([X|Xs], X; Xs)`, `del([X|Xs], X; [X|Xs])`, `del([X|Xs], Y; Xs)` and `del([X|Xs], Y; [Y|Xs])`.
There are incompatible examples with each of the unit clauses suggested by these *msg*'s and no sequence of built-in atoms help. Hence step 2 does not generate any clause and $S3 = S2$.

Consider step 3 now. Unlike step 2, step 3 considers the unique *msg* of $S3$ without any restriction. That *msg* is `del([X|Xs], Y; Zs)` and the considered recursive clause is `del([X|Xs], Y; Zs)` ← `del(Xs, Y; Z1s)`. The set of tuples T is $\{\langle[1], 1, [\ ], [\ ]\rangle, \langle[2], 1, [\ ], [2]\rangle, \langle[2,1], 1, [\ \ ], [2]\rangle, \langle[1,2], 1, [2], [2]\rangle\rangle, \langle[1,2,3], 1, [2,3], [2,3]\rangle, \langle[1,2,1], 1, [2], [2]\rangle\}$. Now, $T2$ contains 2 *msg*'s $\langle[X|Xs], X, Z1s, Z1s\rangle$ and $\langle[X|Xs], Y, Z1s, [X|Z1s]\rangle$ and we get the following two recursive clauses after adding apprpriate built-in atoms.
`del([X|Xs], X; Z1s)` ← `del(Xs, X; Z1s)`
`del([X|Xs], Y; [X|Z1s])` ← `X ≠ Y, del(Xs, Y; Z1s)`
and inference algorithm does not invoke **Generate** hereafter as this program is consistent with each example in any positive presentation of `del`.

The following example illustrates the addition of an **auxiliary** predicate by **Generate**.

Example 5 Let us consider inference of a program for `reverse` with mode/type annotations `rev(in : List; out : List)`. The two patterns to consider are `rev([ ]; Ys)` and `rev([X|Xs]; Ys)`.

Consider the invocation of **Generate** with examples: `rev([]; [])`, `rev([a]; [a])`, `rev([b]; [b])`, `rev([a, a]; [a, a])`, `rev([a, b]; [b, a])`. Step 1 generates the unit clause `rev([ ]; [ ])` ← from the first example and step 2 does not add any clauses as in the above Example.

Step 3 computes the *msg*, `rev([X|Xs]; [Y|Ys])` of S3 and considers the following one-recursive clause: `rev([X|Xs]; [Y|Ys])` ← `p(Xs; Zs)`. The set of tuples T is $\{\langle[a], [], [a]\rangle, \langle[b], [], [b]\rangle, \langle[a,a], [a], [a,a]\rangle, \langle[a,b], [b], [b,a]\rangle\}$. There is no set $T2$ of *msg*'s covering all the tuples in T to relate the output terms `[Y|Ys]` and `Zs` and hence Step Aux is executed.

Conditions 1, 2 and 3 force us to consider `q(Zs, [X]; [Y|Ys])`. In particular, condition 1 forces us to use `[X]` rather than `X`. Now the examples for Auxiliary predicate `q` are $T4 =$ `{q([], [a]; [a]), q([], [b]; [b]), q([a], [a]; [a, a]), q([b], [a]; [b, a])}`. From these examples, **Generate**(T4) generates the clauses:
`q([], Ys; Ys)` ←
`q([X|Xs], Ys; [X|Zs])` ← `q(Xs, Ys; Zs)`
which are nothing but the clauses of append. The recursive clause added for `rev` is
`rev([X|Xs]; [Y|Ys])` ← `rev(Xs; Zs), q(Zs, [X]; [Y|Ys])`.
In the post processing, this clause will be rewritten to
`rev([X|Xs]; Z)` ← `rev(Xs; Zs), q(Zs, [X]; Z)`
replacing the term `[Y|Ys]` by `Z` in both the head and body.

The following example is to illustrate that the algorithm learns predicates without any output position as well.

Example 6 *The algorithm considers two patterns* `list([ ])` *and* `list([H|L])` *and generates the following two clauses*
`list([ ])` ←
`list([H|L])` ← `list(L)`
in learning a predicate `list` *which checks whether a given term is a list or not.*

The following theorem establishes correctness of our algorithm.

Theorem 2 *The above procedure* **Infer-one-recursive**

1. *only generates one-recursive programs which are consistent with the examples read so far (consistent),*
2. *changes its guess only when the most recently read example is not consistent with the current guess (conservative) and*
3. *updates its guess in polynomial time in the size of the current sample of examples read so far (polynomial time updates).*

In view of the notorious difficulty in learning recursive clauses mentioned often in the literature, we explain the main reasons for polynomial time complexity of our algorithm. After reading each example, the algorithm checks whether this new example is consistent with the current program. This consistency check can be done in polynomial time as (1) the leftmost argument of a recursive call is a **proper subterm** of the leftmost argument of the caller, (2) the sum of the sizes of the leftmost arguments of all the recursive calls (in the body of the clause) is at most the size of the leftmost argument of the caller (head of the clause) and (3) the sum of the sizes of input terms of the auxiliary predicate is bounded by the sum of the sizes of input terms of the head. In fact, the sum of the sizes of input terms of atoms in any SLD-derivation of a linear-moded program-query pair is bounded by the sum of the sizes of input terms of the query. Further, by enforcing the discipline that the leftmost arguments of all the recursive atoms in the body are recursive subterms of the leftmost argument of the body and the terms in the clauses are either variables, constants or the first two generic-expressions of the annotated types, we drastically reduce the search space for recursive clauses. This is in sharp contrast to the fact that most of the learning algorithms in the literature spend a lot of time in searching for a suitable recursive clause. The above discipline is encouraged in the programming methodologies advocated by Deville [9] and Sterling and Shapiro [33]. Only notable exception is the `even` program for checking whether a given natural number is even or not, which has a clause `even(s(s(X))` ← `even(X)` with a term `s(s(X))` that is not among the first two generic-expressions of the type `Nat`. We can relax our restriction to cover this program by allowing terms of depth more than 2, but then the algorithm will become a bit inefficient. These decisions should be postponed to the implementation time.

6. Conclusion

In this paper, we approach the problem of learning logic programs from a programming methodology point of view and propose an algorithm to learn a class of Prolog programs, that use divide-and-conquer methodology. This class of programs is very natural and rich and contains many programs from chapter 3 (on recursive programs) of Sterling and Shapiro's standard book on Prolog [33]. This indicates that our algorithm will be successful in practical situations as the underlying class of programs is very natural.

We believe that our results can be extended in the following two directions: (1) to consider predicates that have more than one recursive arguments (we call such programs, k-recursive programs) and (2) to cover the programs which uses divide-and-conquer approach but splits the input using a specific (to that data type) splitting algorithm rather than the splitting suggested by the recursive structure of the data type. For example, splitting a list into two lists of almost equal length. This can be done when we are looking for learning algorithms that work on a particular (fixed) data type. Further investigations are needed in these directions.

References

[1] D. Angluin (1980), *Inductive inference of formal languages from positive data*, Information and Control **45**, pp. 117-135.

[2] K.R. Apt and A. Pellegrini (1992), *Why the occur-check is not a problem*, Proc. of PLILP'92, LNCS **681**, pp. 69-86.

[3] H. Arimura and T. Shinohara (1994), *Inductive inference of Prolog programs with linear data dependency from positive data*, Proc. Information Modelling and Knowledge Bases V, pp. 365-375, IOS press.

[4] H. Arimura, H. Ishizaka and T. Shinohara (1992), *Polynomial time inference of a subclass of context-free transformations*, Proc. Computational Learning Theory, COLT'92, pp. 136-143.

[5] L. Blum and M. Blum (1975), *Towards a mathematical theory of inductive inference*, Information and Control **28**, pp. 125-155.

[6] F. Bronsard, T.K. Lakshman and U.S. Reddy (1992), *A framework of directionality for proving termination of logic programs*, Proc. Joint Intl. Conf. and Symp. on Logic Prog., JICSLP'92, pp. 321-335

[7] W.W. Cohen (1995a), *Pac-learning recursive logic programs: efficient algorithms*, Journal of Artificial Intelligence Research **2**, pp. 501-539.

[8] W.W. Cohen (1995b), *Pac-learning recursive logic programs: negative results*, Journal of Artificial Intelligence Research **2**, pp. 541-573.

[9] Y. Deville (1990), *Logic Programming: Systematic Program Development*, Addison Wesley.

[10] S. Dzeroski, S. Muggleton and S. Russel (1992), *PAC-learnability of determinate logic programs*, Proc. of COLT'92, pp. 128-135.

[11] M. Frazier and C.D. Page (1993), *Learnability in inductive logic programming: some results and techniques*, Proc. of AAAI'93, pp. 93-98.

[12] E.M. Gold (1967), *Language identification in the limit*, Information and Control **10**, pp. 447-474.

[13] P. Idestam-Almquist (1993), *Generalization under Implication by Recursive Anti-unification*, Proc. of ICML'93.

[14] P. Idestam-Almquist (1996), *Efficient induction of recursive definitions by structural analysis of saturations*, pp. 192-205 in L. De Raedt (ed.), *Advances in inductive logic programming*, IOS Press.

[15] J.-U. Kietz (1993), *A Comparative Study of Structural Most Specific Generalizations Used in Machine Learning*, Proc. Workshop on Inductive Logic Programming, ILP'93, pp. 149-164.

[16] J.-U. Kietz and S Dzeroski (1994), *Inductive logic programming and learnability*, SIGART Bull. **5**, pp. 22-32.

[17] M.R.K. Krishna Rao, D. Kapur and R.K. Shyamasundar (1997), *A Transformational methodology for proving termination of logic programs*, The Journal of Logic Programming **34**, pp. 1-41.

[18] M.R.K. Krishna Rao (2001), *Some classes of Prolog programs inferable from positive data*, Theoretical Computer Science **241**, pp. 211-234.

[19] S. Lapointe and S. Matwin (1992), *Sub-unification: a tool for efficient induction of recursive programs*, Proc. of ICML'92, pp. 273-281.

[20] N. Lavrac, S. Dzeroski and M. Grobelnik (1991), *Learning nonrecursive definitions of relations with LINUS*, Proc. European working session on learning, pp. 265-81, Springer-Verlag.

[21] J. W. Lloyd (1987), *Foundations of Logic Programming*, Springer-Verlag.

[22] S. Miyano, A. Shinohara and T. Shinohara (1991), *Which classes of elementary formal systems are polynomial-time learnable?*, Proc. of ALT'91, pp. 139-150.

[23] S. Miyano, A. Shinohara and T. Shinohara (1993), *Learning elementary formal systems and an application to discovering motifs in proteins*, Tech. Rep. RIFIS-TR-CS-37, Kyushu University.

[24] S. Muggleton and L. De Raedt (1994), *Inductive logic programming: theory and methods*, J. Logic Prog. **19/20**, pp. 629-679.

[25] S. Muggleton (1995), *Inverting entailment and Progol*, in Machine Intelligence 14, pp. 133-188.

[26] G. Plotkin (1970), *A note on inductive generalization*, in Meltzer and Mitchie, *Machine Intelligence 5*, pp. 153-163.

[27] L. Plümer (1990), *Termination proofs for logic programs*, Ph. D. thesis, University of Dortmund, Also appears as Lecture Notes in Computer Science **446**, Springer-Verlag.

[28] J.R. Quinlan and R.M. Cameron-Jones (1995), *Induction of logic programs: foil and related systems*, New Generation Computing **13**, pp. 287-312.

[29] Y. Sakakibara (1990), *Inductive inference of logic programs based on algebraic semantics*, New Generation Computing **7**, pp. 365-380.

[30] E. Shapiro (1981), *Inductive inference of theories from facts*, Tech. Rep., Yale Univ.

[31] E. Shapiro (1983), *Algorithmic Program Debugging*, MIT Press.

[32] T. Shinohara (1991), *Inductive inference of monotonic formal systems from positive data*, New Generation Computing **8**, pp. 371-384.

[33] L. Sterling and E. Shapiro (1994), *The Art of Prolog*, MIT Press.

[34] J.D. Ullman and A. van Gelder (1988), *Efficient tests for top-Down termination of logical rules*, JACM **35**, pp. 345-373.

[35] A. Yamamoto (1993), *Generalized unification as background knowledge in learning logic programs*, Proc. of ALT'93, LNCS **744**, pp. 111-122.

Incremental discovery of sequential patterns for grammatical inference

Ramiro Aguilar RAMIRO@TEJO.USAL.ES
Instituto de Investigaciones en Informática, Universidad Mayor de San Andrés
Av. Villazón 1995, Monoblock Central. La Paz, Bolivia

Luis Alonso, Vivian López, María N. Moreno {LALONSO, VIVIAN, MMG}@USAL.ES
Departmento de Informática y Automática, Universidad de Salamanca,
Plaza de la Merced S/N, 37008 Salamanca, Spain

Abstract

In this work a methodology is described to generate a grammar from textual data. A technique of incremental discovery of sequential patterns is presented to obtain production rules simplified production rules, and compacted with bioinformatics criteria that make up a grammar that recognizes not only the initial data set but also extended data.

1. Introduction

The growing quantity of documentary information makes its analysis complex and tedious, so that automatic and intelligent methods for their processing are needed. To understand, "what say the data" is necessary to know the structure of the data language. In order to this, in (López & Aguilar, 2002) a general plan is defined that proposes a data mining method on text, to discover the syntactic-semantic knowledge of it. As part of all the proposed process a method is presented to obtain the grammar from the sequential patterns obtained in the text. This is the grammatical inference (GI) of the language of the text.

In this work a novel data mining process is described that combines hybrid techniques of association analysis and classical sequentiation algorithms of genomics to generate grammatical structures from a specific language. Subsequently, these structures are converted to *context-free grammars*. Initially the method applies to context-free languages with the possibility of being applied to other languages: structured programming, the language of the book of life expressed in the genome and proteome and even the natural languages.

1.1. Problem of grammatical inference

Grammatical inference (GI) is transversal to a number of fields including machine learning, formal languages theory, syntactic and structured pattern recognition, computational biology, speech recognition, etc. (de la Higuera, 2004).

Problem of GI is the learning of a language description from language data. The problem of context-free languages inference involves practical and theoretical questions. Practical aspects includes pattern and speech recognition; an approach of pattern recognition is the context-free grammatical (CFG) inference that built a set of patterns (Fu, 1974); another approach search the ability to infer CFGs from natural that would enable a speech recognizer to modify its internal grammar on the fly, thus allowing it to adjust to individual speakers (Horning, 1969). Theoretical aspects have importance as for the serious limitations of context-free languages (Lee, 1996) and, actually, to construct feasible algorithms of learning that imitate the model of the human language (this last point, can be problematic, but was one of the principal motivations for the early work in grammar inference (Horning, 1969)).

1.2. Language learning

The learning of a language also has to do with its identification. In the literature of the grammar inference, the attention focuses on the *identification in the limit*, this way, in each time t the machine that learns receives a information unit i_t on a language and output a hypothesis $H(i_1, ..., i_t)$; the learning algorithm is successful if after a finite amount of time, all its guesses are the same and are all a correct description in the language of the question. Another learning criteria

is the *exact identification using queries in polynomial time*, in this framework, the learning machine have access to the oracles that can answer questions, and must halt in polynomial time with a correct description of the language (Lee, 1996).

In the last 20 years, the inherent complexity present in the problem of grammatical inference, made unsuccesfull all the approaches (Miclet, 1986). The paper detailed in (de la Higuera, 2002) states that actually the algorithms with mathematical properties obtain better results than the algorithms with heuristic properties, but is when finite automata are used or, on the other hand, when algorithms of GIC learning are constructed, also it emphasizes that for the heuristic approach common "benchmark" does not exist and is then more difficult to compare and to evaluate the effectiveness of these methods. With the mentioned thing previously, our proposal has a data set available with which it is validated and we are open to other comparisons to improve or to ratify our work.

2. Techniques for the association analysis

Association analysis involves techniques that are different in its operations but all of them search relations among the attributes of a data set. Some techniques are:

- Association rules
- Discovery of sequential patterns, and
- Discovery of associations

2.1. Association rules

The association rules (AR) describe the relations of certain attributes with regard to others attributes in a database (DB). These rules identify cause-effect implications between the different attributes of the DB. For example, in the registers of products purchases, what article of purchase is identified as related to another; for instance: "the 80% of the people that buys diapers for baby, also buys talcum".

A rule have the form "if X then Y" or $X \Rightarrow Y$. X is called *antecedent* of the rule (in the example, "buys diapers"); Y is called *consequent* of the rule (in the example, "buys talcum").

The generation of the rule is supported by statistical and probabilistic aspects such as the support factor (f_s), confidence factor (f_c) and the expected confidence factor (f_e) defined as: $f_s = \frac{nr_times_rule}{nr_total_registers}$, $f_c = \frac{nr_times_rule}{nr_times_X}$ and $f_e = \frac{nr_times_Y}{nro_total_registers}$.

Table 1. Data set for association rules.

A	B	C	D	E	F
2	2	6	0	1	0.2
2	2	5	0	1	0.2
2	2	6	1	1	0.2
3	2	7	1	0	0.8
2	3	8	1	0	0.8
3	3	8	1	0	0.8
3	3	7	1	0	0.8

The minimum value of the support factor for the rules should be greater than a given threshold. If the confidence factor is greater than 0.5, then the rule appears, at least, in half the number of instances that means that the rule has certain sense. The difference between the support factor and the expected confidence factor should be minimum to assure the effectiveness of the rule.

For example, we consider the data of the table 1, a rule obtained is: $A = 2 \Rightarrow B = 2$ with $f_s = 0.43$, $f_c = 0.75$ and $f_e = 0.57$, means that the 75% of items whit $A = 2$ imply $B = 2$, besides in the 43% of all items complies that rule and $B = 2$ complies in the 57% of all items.

2.2. Discovery of associations

Similarly to the AR, the discovery of associations (DA) tries to find implications between different couples attribute-value so that the appearance of these determine a present association in a good quantity of the registers of the DB. To discover associations the following steps are carried out:

1. Associate an identifier to each transaction
2. Order sequentially the transactions according to its identifier
3. Count the occurrences of the articles creating a vector where each article is counted. The elements where the account is below of a "threshold", are eliminated
4. Combine in a matrix the transactions attribute-value and carry out the count of occurrences eliminating those elements that do not surpass the threshold
5. Repeat successively the steps 3 and 4 until no more transaction combinations are possible

Whit the data of the table 1, whit $threshold = 2$, the technique is applied as is observed in the figure 1 and the following associations are generated:

1. $A2 \Rightarrow B2 \Rightarrow E1 \Rightarrow F0.2$
2. $A3 \Rightarrow D1 \Rightarrow E0 \Rightarrow F0.8$
3. $B3 \Rightarrow D1 \Rightarrow E0 \Rightarrow F0.8$

The association 1 means: if the value of A and B is 2 and the value of E is 1 and the value of F is 0.2, then the registers with those characteristics can belong to a class. The other associations show the possible characteristics of the registers to belong to another class or behavior.

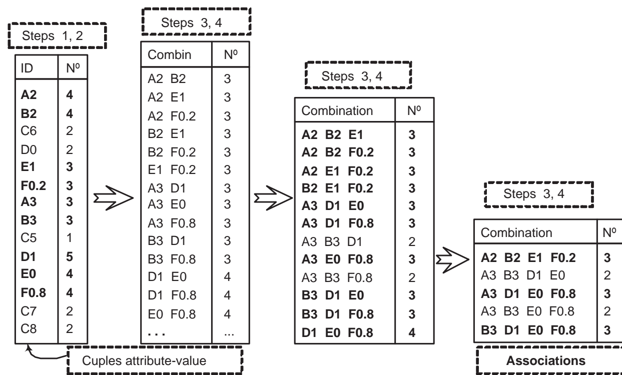

Figure 1. Discovery of associations in data of the table 1.

2.3. Discovery of sequential patterns

Discovery of sequential patterns (DSP) is very similar to the AR but search for patterns between transactions so that the presence of a set of items precede another set of items in a DB during a period of time. For example, if the data correspond to registers of articles purchased by clients, a description of what articles buys frequently a client can be obtained, and above all, which is the sequence of its purchase. Thus, the next time, the profile of the client would be known, and it will be able to predict the sequence of its purchase. This criteria can apply to another data control, for example, in the Bioinformatics context, when the data to treat correspond to the chain of nucleotides of the genome and sequences are discovered as the patterns that codify genes conform some protein (Fayyad et al., 1996) (Aguilar, 2003).

DSP have the following operation:

1. Identify the time related attribute
2. Considering the period of time when the sequential patterns are to be discovered, create an array ordered by the identifier of the transtaction
3. Create another array linking the articles of purchase of each client
4. According to the "support percent", infer the sequential patterns

The discovered patterns show instances of articles that appear in consecutive form in the data as is appreciated in the example of the figure 2.

3. Grammars, languages and bioinformatics

3.1. Context-free grammar

A grammar $\mathcal{G}$ is defined like $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$, where $\mathcal{N}$ is the set of non terminals symbols, $\mathcal{T}$ is the set of terminals symbols or syntactic categories, $\mathcal{P}$ is the set of production rules and $\mathcal{S}$ is the initial symbol. The language of a grammar $\mathcal{L}(\mathcal{G})$ is the set of all terminal strings w that have derivations from the initial symbol. This is: $\mathcal{L}(\mathcal{G}) = \{w\ is\ in\ \mathcal{T}^* \mid \mathcal{S} \Rightarrow^* w\}$

A Context-Free Grammar (CFG) has production rules like $A \rightarrow \alpha$ where $A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \mathcal{T})^*$. The substitution of A by α is carried out independently of the place in which appear A (Louden, 1997). The majority of the programming languages are generated by grammars of this type (enlarged with some contextual elements necessary for the language semantics)

3.2. Grammars and bioinformatics

Bioinformatics employs computational and data processing technologies to develop methods, strategies and programs that permit to handle, order and study the immense quantity of biological data that have been generated and are currently generated. For example, for the human genome (HG), the bioinformatics seeks to find meaning to the language of the more than 37.000 million peers A, C, T and G that have been compiled and stored in the "book of life".

They offer us the opportunity to understand the gigantic DB that contain the details of the circumstances of time and place in which the genes are activated, the conformation of the proteins that specify, the form in which they influence some proteins on others and the role that such influences can play in the diseases. Besides, what are the relations of the HG with the genomes of the model organisms, like the fly of the fruit, the mice and the bacteria? Will it be able to discover sequential patterns that show how are related

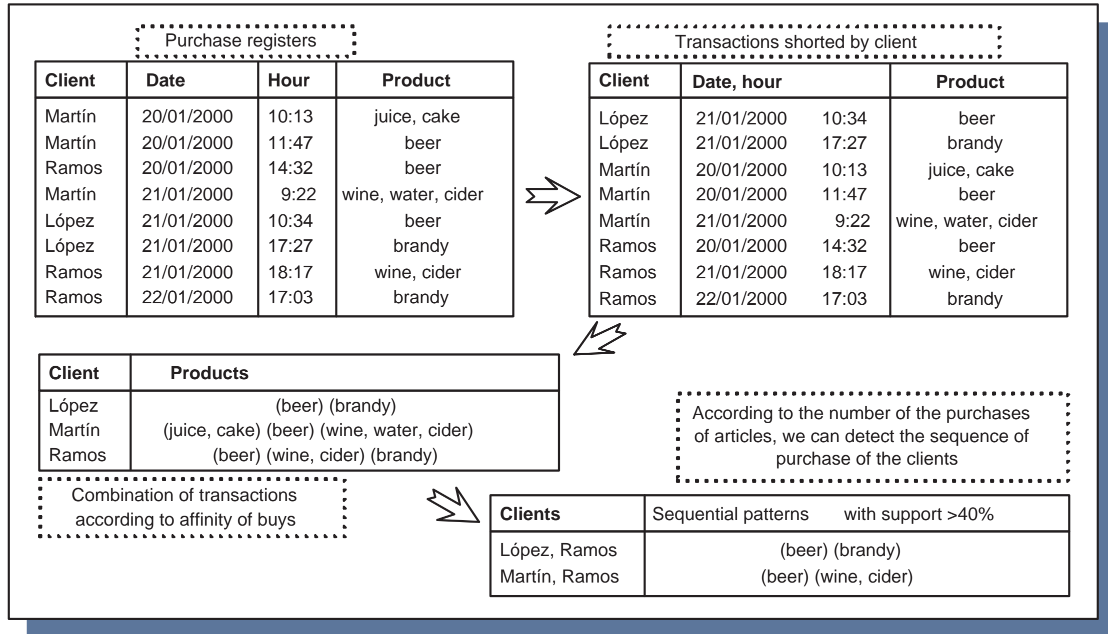

Figure 2. Discovery of sequential patterns in registers of products purchases (elaborated according to (Cabena et al., 1998)).

between itself the fragments of information? and will it be able to conform a grammatical structure that show the interpretation of the resultant set? If we are able to infer that structure for this type of language we will contribute to understand the real function of the structure of the DNA and we will understand slightly more than the questions presented.

One of the applications of the bioinformatics is the farmacology, offering reviving solutions to the old model for the creation of new medicines. It is worth to note that, one of the more elementary bioinformatics operations consists of the search of resemblances between a fragment of DNA recently arranged and the already available segments of diverse organisms (remember and associate this with the DSP). The finding of approximate alignments permits to predict the type of protein that will specify such sequence. This not only provides trails on pharmacological designs in the initial phases of the development of medicines, but suppresses some that will constitute un resolving “puzzles”. A popular series of programs to compare sequences of DNA is BLAST (Basic Local Alignment Search Tool) (Altschul et al., 1990) (Altschul et al., 1997) whose mechanism of comparison applied in the development of the new medicine is shown in the plan of the figure 3.

4. Data mining procedure for the grammatical inference

The idea considers the experiences acquired (Aguilar, 2003), the literature and the existing theories (Mitra & Acharya, 2003) (Louden, 1997) (Moreno, 1998), carrying out the prosecution on data that are not structured in relations or tables with differentiated attributes but those are codified as a finite succession of sentences.

The data mining procedure has the following phases:

- Language generation by means of an context-free grammar. This language will be the source of data

- Codification of the strings of the language regarding its syntactic categories

- Dispensing with the initial grammar, discovery of sequential patterns on the codified language. This discovery, called “incremental”, is a combination of the operation of the DSP and of the operation of the search of identical sequences. With this, patterns of sequences will be found that then will be replaced by an identifier symbol

- Replace the discovered sequences by their identifiers. With the previous thing the identifier is stored and the sequence as a production rule

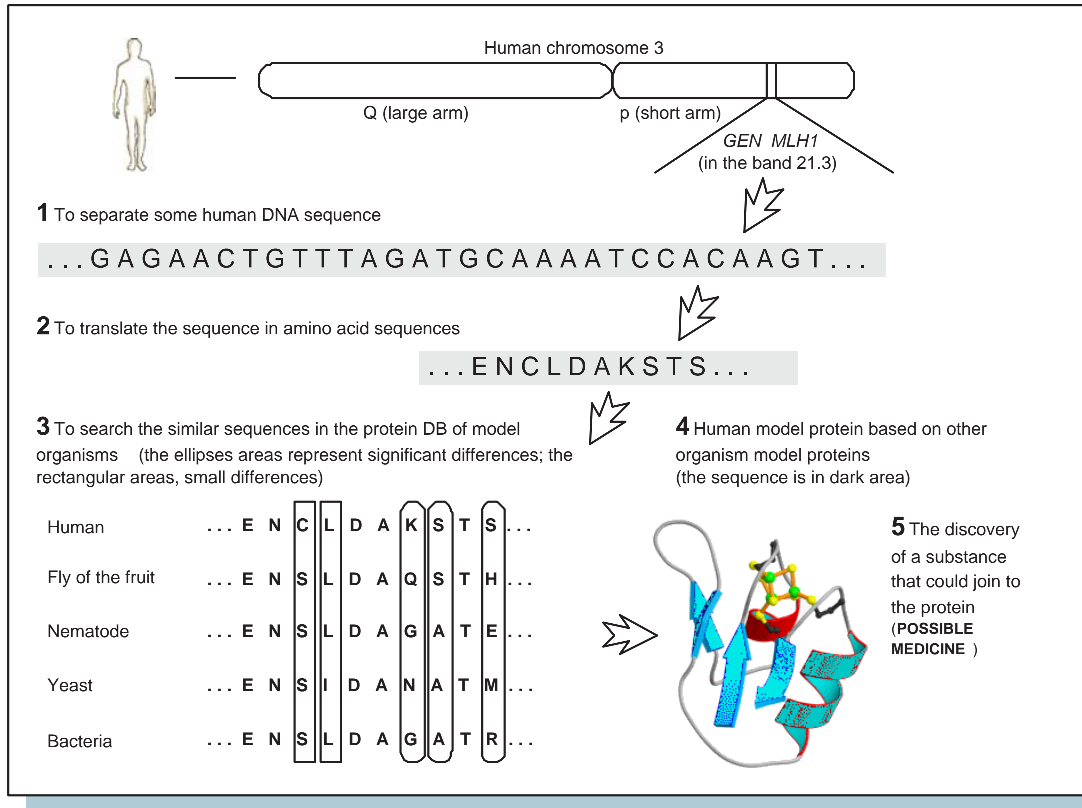

Figure 3. Utilization of the bioinformatics in the farmacology (elaborated according to (Howard, 2004)).

- Repeat the two previous steps until all the sentences of the language are replaced by identifiers

4.1. Language generation

We consider the CFG $\mathcal{G}_{æ}$ proposed in (Louden, 1997) about the generation of arithmetic expressions $\mathcal{G}_{æ} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ where $\mathcal{N} = \{\text{Exp, Num, Dig, Op}\}$, $\mathcal{T} = \{0, 1, +, *\}$,

$\mathcal{P}:$ Exp $\rightarrow$ Exp Op Exp | (Exp) | Num

Num $\rightarrow$ Dig$^+$

Dig $\rightarrow$ 0 | 1

Op $\rightarrow$ + | *

and $\mathcal{S}$ =Exp.

We can modify the formalism of this CFG of the following form:

$\mathcal{G}_{æ} = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$ where $\mathcal{N} = \{\text{E, d, b, o, a, c}\}$, $\mathcal{T} = \{0, 1, +, *, (,)\}$,

$\mathcal{P}:$ E $\rightarrow$ E o E | a E c | n

d $\rightarrow$ b$^+$

b $\rightarrow$ 0 | 1

o $\rightarrow$ + | *

a $\rightarrow$ (

c $\rightarrow$)

and $\mathcal{S}$ =E, what does not change in essence the character of the original grammar.

With the previous criteria, a sample of the language generated by $\mathcal{G}_{æ}$ can be seen in the figure 4, point (i). It is noted that each line corresponds to a sentence accepted by the grammar.

4.2. Language codification

Considering the language that is generated with $\mathcal{G}_{æ}$, all the symbols of $\mathcal{T}$ can be codified with the symbols of $\mathcal{N}$, only for this particular case the symbols to be used are {b, o , a, c} as syntactic categories. See the figure 4, point (ii).

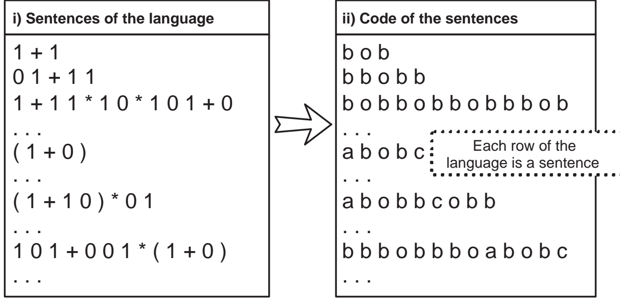

Figure 4. Language of arithmetic expressions on which its grammar is inferred.

4.3. Incremental discovery of sequential patterns and associations

The hybrid discovery of sequential patterns applied to codified languages seeks key subsequences in the sentences of the language. Each subsequence $\mathbf{q}$ has a length $\mathbf{w_q}$ that indicates the number of symbols that possesses. In this particular case $1 \leq \mathbf{w_q} \leq 5$ and $\mathbf{Q}$ is defined as a string of length $\mathbf{w_Q}$. By convention, in the codified language many sentences exist that conform the population of the language. The idea consists of finding subsequences, to identify them with a symbol and to replace with that symbol the appearances of the subsequences in the sentences of the population, all the previous procedure of repetitive form until each sentence is identify by a single symbol.

The detailed steps are:

1. For all the sentences, While $\mathbf{w_Q} > 1$ do:

- **1.1.** For $\mathbf{w_q} = 1..5$ do:
  - **1.1.1.** For all the sentences:
    - **(a)** Make $\mathbf{q}$ from then $\mathbf{w_q}$ first symbols of $\mathbf{Q}$
    - **(b)** Compute the global scoring $\mathbf{g_q}$ of $\mathbf{q}$ defined as $\mathbf{g_q} = \sum_{i=1}^{cant} \mathbf{p_q^i}$, where $\mathbf{p_q} = \frac{\mathbf{w_q} * nr_apparitions_of_q_in_Q}{\mathbf{w_Q}}$ is the scoring of $\mathbf{q}$ in $\mathbf{Q}$
  - **1.1.2.** End For
- **1.2.** End For
- **1.3.** Selecting the subsequence $\mathbf{q}_*$ of greater global scoring
- **1.4.** If $\mathbf{q}_*$ has one symbol, then replacing all the consecutive appearances of that symbol by itself. Thus the production rule is created $\alpha \rightarrow \alpha^+$ (in this particular case, this are replaced all the *bb* by *d*, to see figure 5)
- **1.5.** If $\mathbf{q}_*$ has more than one symbol, then replace all the appearances of $\mathbf{q}_*$ in the sentences $\mathbf{Q}$ creating the production rule $A \rightarrow contained_of(\mathbf{q}_*)$. The symbol A is generated consecutively so that the following time that another rule production is created, is utilized $B, C, ...$ and so on (to see figure 5)

2. Returning to step 1 noting that with 1.4 and 1.5 changes the size of the sentences of the population of the language

With the previous procedure production rules are generated that recognize the sentences of the language. The production rules number can be considerable so that we apply a particular method of simplification of grammar.

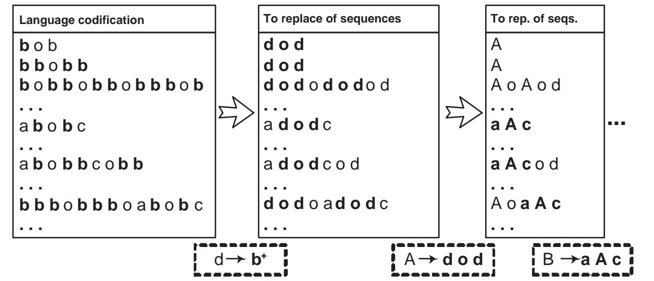

Figure 5. Hybrid discovery of sequential patterns for the context-free languages.

5. Experiments

5.1. Rules similarity

Considering the language $\mathcal{L}_{æ}$ of arithmetic expressions[1], we apply the hybrid algorithm of DSP and the production rules of the figure 6 were obtained. With the right hand of rules, that conform the sequential patterns of the language, a *substitution matrix* is computed that it is observed in figure 7, this matrix shows the similarity values between terminal symbols. Similarity among a pair of consecutive symbols is related with the apparition frequency of the symbols in the language (is a matrix like BLOSUM matrix (Henikoff & Henikoff, 1992)). Subsequently, is possible to make alignments among those sequences by compact them.

In the substitution matrix $m(i, j)$ each row i and each column j correspond with a non terminal symbol of the production rules generated. The symbols are put

[1] The corpus can be observed in `http://www.geocities.com/ramirohp/corpusae.html`

Rules generated
S → C G
R → F o F o A
Q → C F
P → a a N c c
O → C D
N → a a F c c
M → C A
L → F o D
K → C I
J → d o F
I → C E
H → F o E
G → C B
F → a d c
E → C d
D → B o B
C → A o
B → a A c
A → d o d

In the rules, replace **J** by **A**

Iteration 1
S → C G
R → F o F o A
Q → C F
P → a a N c c
O → C D
N → a a F c c
M → C A
L → F o D
K → C I
I → C E
H → F o E
G → C B
F → a d c
E → C d
D → B o B
C → A o
B → a A c
A → d o d | d o F

In the rules, replace **F** by **B**

Iteration 2
S → C G
R → **B o B** o A
Q → C B
P → a a N c c
O → C D
N → a a B c c
M → C A
L → B o D
K → C I
I → C E
H → B o E
G → C B
E → C d
D → B o B
C → A o
B → a A c | a d c
A → d o d | d o B

Figure 6. Production rules generated and some iterations in its simplification.

according its apparition frequency, this is, first d, after A, C, o and so on. For $\mathcal{L}_{æ}$ themselves it generated 19 symbols A, B,..., S that join with the symbols of the codification d, o, c and a they conform 23 non terminal symbols (in the bioinformatics context, the symbols would correspond to the amino acids). The values of the matrix denote the importance of the alignment among the not terminal symbols; for example, $m(d, d) = 23$ denotes a degree of high similarity between both symbols; $m(d, A)$ denotes a degree of similarity of -1.

5.2. Rules simplification and compaction

With the right parts of the productions rules (where the first rules generated have greater importance) we search *similar sequences* to compact them.

The steps are:

- The sequence β that can be compacted with the sequence α is activated with the similarity function f; $f(\alpha, \beta) = \begin{cases} 1 & si\ \frac{\sum_{i=1}^{n} m(\alpha_i, \beta_i)}{\sum_{i=1}^{n} m(\alpha_i, \alpha_i)} > \theta; \\ 0 & si\ e.o.c. \end{cases}$ where n is the minimal length between the sequences α and β, θ is a threshold or *similarity factor* with value 0.4 in this particular case
- The similar sequences are compacted and will be derived by a single non terminal symbol. The remaining non terminal symbol should be replaced for the previous one in all the right parts of the rules

Substitution matrix

	d	A	C	o	E	B	F	D	. . .	P	R
d	23	-1	-2	-3	-4	-5	-6	-7	. . .	-21	-22
A	-1	22	-1	-2	-3	-4	-5	-6	. . .	-20	-21
C	-2	-1	21	-1	-2	-3	-4	-5	. . .	-19	-20
o	-3	-2	-1	20	-1	-2	-3	-4	. . .	-18	-19
E	-4	-3	-2	-1	19	-1	-2	-3	. . .	-17	-18
B	-5	-4	-3	-2	-1	18	-1	-2	. . .	-16	-17
F	-6	-5	-4	-3	-2	-1	17	-1	. . .	-15	-16
D	-7	-6	-5	-4	-3	-2	-1	16	. . .	-14	-15
.	..	..	..	..	..	..	..	..	. . .	..	..
.	..	..	..	..	..	..	..	..	. . .	..	..
.	..	..	..	..	..	..	..	..	. . .	..	..
P	-21	-20	-19	-18	-17	-16	-15	-14	. . .	2	-1
R	-22	-21	-20	-19	-18	-17	-16	-15	. . .	-1	1

Figure 7. Substitution matrix for the rules generated.

- Repeat the previous steps until there are no similar sequences

For example, for the language $\mathcal{L}_{æ}$ the rules **dod** and **aAc** are not similar since $f(\textbf{dod}, \textbf{aAc}) = 0$ since $\frac{\sum m(\textbf{dod},\textbf{aAc})}{\sum m(\textbf{dod},\textbf{dod})} = \frac{-10-2-11}{23+20+23} = \frac{-23}{66} = -0.35$ is not greater than 0.40. Nevertheless, the rules **dod** and **doF** are similar since, $\frac{\sum m(\textbf{dod},\textbf{doF})}{\sum m(\textbf{dod},\textbf{dod})} = \frac{23+20-6}{23+20+23} = \frac{37}{66} = 0.56$. This way, the generated rules are simplifying and compacting iteratively (figures 6 and 8) until a grammar is built $\mathcal{G}'_{æ} = (\mathcal{N}', \mathcal{T}', \mathcal{P}', \mathcal{S}')$ where $\mathcal{N}' = \{$S, R, E, D, B, A, d, b, o, a, c$\}$, $\mathcal{T}' = \{0, 1, +, *, (,)\}$,

$\mathcal{P}'$: S → R | E | D | B | A | d

R → DoA

E → Cd | CB | CE | CA | CD

D → BoB | BoE | BoD

C → Ao

B → aAc | adc

A → dod | doB

d → b$^+$

b → 0 | 1

o → + | *

a → (

c →)

and $\mathcal{S}'$ =S.

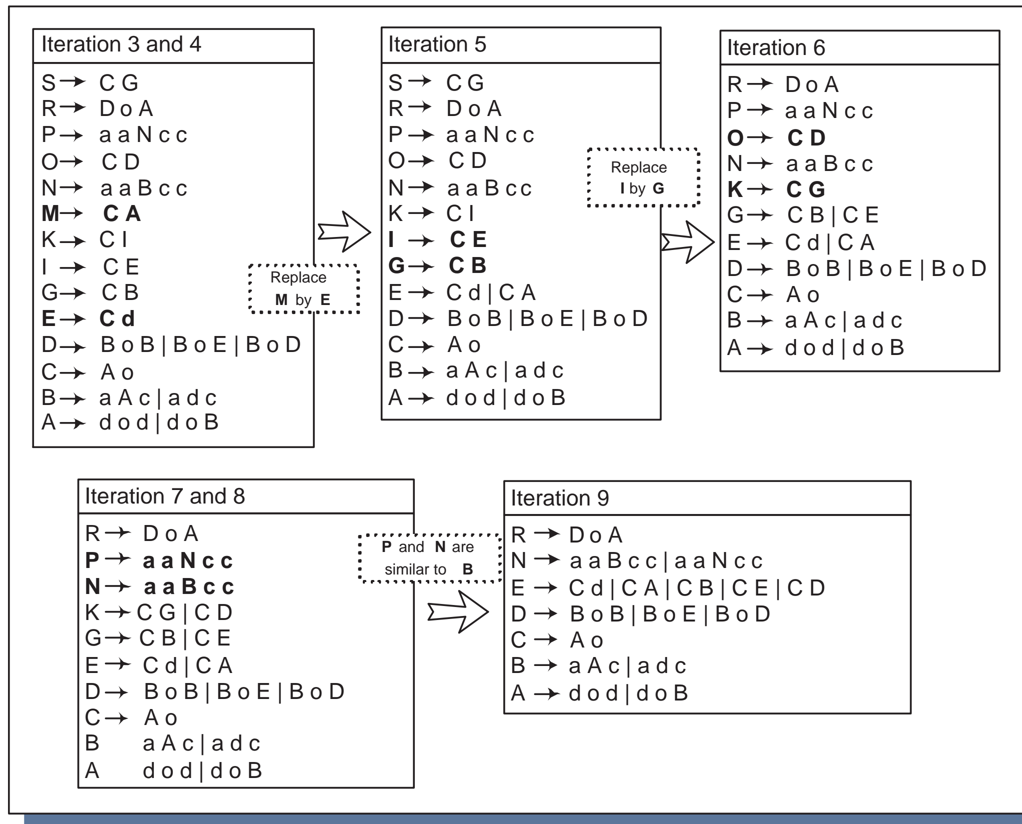

Figure 8. Simplification and compaction of the production rules generated.

6. Conclusions

In the experiments, a language $\mathcal{L}_{æ}$ has been considered generated by predetermined context-free grammar $\mathcal{G}_{æ}$ and the syntactic categories **b**, **o**, **a** and **c** were known beforehand; but later none of the properties of that grammar were utilized to generate the set of production rules that then conformed the grammar $\mathcal{G}'_{æ}$. The approach extends to processing of data that are believed to have a grammatical structure that could be generated automatically. We could imagine to find somewhat similar for the genome, for the proteome or for the natural languages, the doubt is served.

References

Aguilar, R. (2003). *Minería de datos. Fundamentos, técnicas y aplicaciones.* Salamanca: University of Salamanca.

Altschul, S. F., Gish, W., Miller, W., Meyers, E. W., & Lipman, D. J. (1990). Basic local alignment search tool. *Molecular Biology*, *215*, 403–410.

Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, J., Zhang, Z., Miller, W., & Lipman, D. J. (1997). Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Research*, *25*, 3389–3402.

Cabena, P., Hadjinian, P., Stadler, R., Verhees, J., & Zanasi, A. (1998). *Discovering data mining. From concept to implementation.* Prentice Hall.

Corlett, L. (2003). *Web content mining: a survey* (Technical Report). Department of Computer Science, California State University.

de la Higuera, C. (2002). Current trends in grammatical inference. *Proceedings of Joint Iapr International Workshops Sspr 2000 and Spr 2000* (pp. 130–135). Lecture Notes in Artificial Intelligence, Springer-Verlag.

de la Higuera, C. (2004). A bibliographical study of grammatical inference. *Pattern Recognition.*

Fayyad, U., Piatetsky-Shapiro, G., Smyth, P., & Uthurusamy, R. (1996). *Advances in knowledge discovery and data mining.* Salamanca: MIT Press.

Fu, K.-S. (1974). *Syntactic methods in pattern recognition.* Academic Press.

Henikoff, S., & Henikoff, J. G. (1992). Amino acid substitution matrices from protein blocks. *Proc. National Academic Science*, *89*, 10915–10919.

Horning, J. J. (1969). *A study of grammatical inference* (Technical Report 139). Computer Science Department, Stanford University.

Howard, K. (2004). La fiebre de la bioinformática. *Investigación y ciencia: nueva genética*, *38*, 79–82.

Lee, L. (1996). *Learning of context-free languages: a survey of the literature* (Technical Report). Computer Science Department, Stanford University.

López, V., & Aguilar, R. (2002). Minería de datos y aprendizaje automático en el procesamiento del lenguaje natural. *Workshop de minería de datos y aprendizaje, IBERAMIA 2002* (pp. 209–216). Sevilla: University of Sevilla.

Louden, K. C. (1997). *Compiler construction. Principles and practice.* International Thomsom Publishing Inc.

Lucas, S. (1994). Structuring chromosomes for context free grammar evolution. *Proceedings of first IEEE International Conference on Evolutionary Computation* (pp. 130–135).

Miclet, L. (1986). *Structural methods in pattern recognition.* Chapman and Hall.

Mitra, S., & Acharya, T. (2003). *Data mining. Multimedia, soft computing and bioinformatics.* John Wiley and sons.

Moreno, A. (1998). *Linguística computacional.* Madrid: Editorial Síntesis.

Data-dependencies and Learning in Artificial Systems

Palem GopalaKrishna KRISHNA@CSE.IITB.AC.IN
Research Scholar, Computer Science & Engineering, Indian Institute of Technology - Bombay, Mumbai, India

Abstract

Data-dependencies play an important role in the performance of learning algorithms. In this paper we analyze the concepts of data dependencies in the context of artificial systems. When a problem and its solution are viewed as points in a system configuration, variations in the problem configurations can be used to study the variations in the solution configurations and vice versa. These variations could be used to infer solutions to unknown instances of problems based on the solutions to known instances, thus reducing the problem of learning to that of identifying the relations among problems and their solutions. We use this concept in constructing a formal framework for a learning mechanism based on the relations among data attributes. As part of the framework we provide metrics – *quality* and *quantity* – for data samples and establish a knowledge conservation theorem. We explain how these concepts can be used in practice by considering an example problem and discuss the limitations.

1. Introduction

Two instances of a function can only differ in their arguments, i.e. the input data. When a function is sensitive to the data it is operating upon, even a slight variation in the nature of data can cause large variations in the path of execution. This property of being sensitive to data is termed as *data-dependency* which poses critical restrictions on the applicability of algorithms themselves.

The success of any data-dependent learning algorithm highly depends on the nature of the data samples it learns from. A well designed algorithm with mismatched data is unlikely to succeed in generalization. Thus a careful analysis of the size and quality of the input data samples is vital for the success of every learning algorithm. While there exists sufficient number of metrics for learning in traditional systems in this regard (Kearns, 1990; Angluin, 1992), there exists almost none for learning in artificial systems, where the typical requirements would be *action selection* and *planning* implemented through agents (Wilson, 1994; Bryson, 2003). These agents would act as deterministic systems and thus demand non-probabilistic metrics with data-independent algorithms.

Data-independence essentially means that the path of execution (the series of instructions carried out) is independent of the nature of the input data. In other words, when an algorithm is said to be data-independent, all instances of the algorithm would follow the same execution path no matter what the input data is. We can understand this with the following example. Consider an algorithm to search a number in a given array of numbers. Such an algorithm would typically look like below.

```
int Search(int Array[], int ArrLen, int Number) {
    for( int i=0; i < ArrLen; ++i )
        if( Array[i] == Number)
            return i;
    return -1;
}
```

The above procedure sequentially scans a given array of numbers to find if a given number is present in the array. It returns the index of the number if it finds a match and -1 otherwise. The time complexity of this algorithm is $O(1)$ in the best case and $O(n)$ in the average and worst cases. However, if we change the iterator construct from $for(i = 0; i <$ArrLen$; ++i)$ to $for(i=$ArrLen-1$;\ i \geq 0; - -\ i)$, then the performances would vary from best to worst and vice versa.

On the other hand consider the following data-independent version of the same code.

```
int Search1(int Array[], int ArrLen, int Number) {
    int nIndex = -1;
    for(int i=0; i < ArrLen; ++i) {
        int bEqual = (Number == Array[i]);
        nIndex = bEqual * i + !bEqual * nIndex;
    }
    return nIndex;
}
```

Search1 is same as *Search* with the mere exception that we have replaced the non-deterministic *if* statement with a series of deterministic arithmetic constructs that in the end produce same results. The advantage with this replacement is that the path of execution is deterministic and independent of the input array values, thus facilitating us to reorder or even parallelize the individual iterations. This is possible because no $(i+1)$th iteration depends on the results of ith iteration, unlike the case of *search* where the $(i+1)$th iteration would be processed only if the ith iteration fails to find a match.

Demanding a time complexity of $O(n)$ in all cases, it might appear that *Search1* is inferior to *Search* in performance. However, for this small cost of performance we are gaining two invaluable properties that are crucial for our present discussion: *stability* and *predictability*.

It is a well-known phenomenon in the practice of learning algorithms that the performance of learner is highly affected by the order of the training data samples, making the learner unstable and at times unpredictable. In this regard, what relation could one infer between the stability of the learner and the dependencies among data samples? How does such relation affect the performance of learner? Can these dependencies be analyzed in a formal framework to assist the learning? These are some of the issues that we try to address in the following.

2. Learning in Artificial Systems

By an *artificial system* we essentially mean a man-made system that has a software module, commonly known as *agent*, as one of its components. The artificial system itself could be a software program such as a simulation program in a digital computer, or it could be a hardware system such as an autonomous robot in the real world. And there could be more than one agent in an artificial system. The system can use the agents in many ways as to steer the course of simulation or to process the environmental inputs (or events) and take the necessary action etc.... Additionally, the functionality of agents could be static, i.e. does not change with experience, or it could be dynamic, varying with experience. The literature addressing these can be broadly classified into two classes, namely the theories that study the agents as pre-programmed units (such as (Reynolds, 1987; Ray, 1991)), and the theories that consider the agents as *learning units* which can adjust their functionality based on their experience (e.g. (Brooks, 1991; Ramamurthy et al., 1998; Cliff & Grand, 1999)). The present discussion falls into the second category. We discuss a learning mechanism for agents based on the notion of data-dependencies.

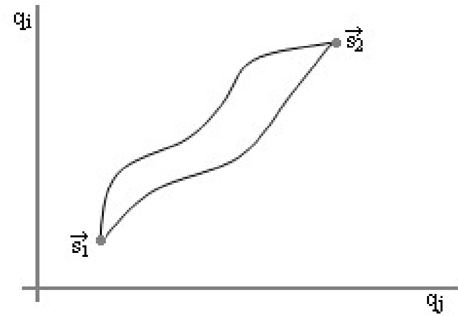

Figure 1. Different paths indicate different algorithms to solve a task instance in configuration space

Consider an agent that is trying to accomplish a task, such as solving a maze or sorting the events based on priority etc..., in an artificial system.

Assume that the instantaneous configuration (the current state) of artificial system is described by n generalized coordinates $q_1, q_2, \ldots, q_n$, which corresponds to a particular point in a Cartesian hyperspace, known as the *configuration space*, where the q's form the n coordinate axes. As the state of the system changes with time, the system point moves in the configuration space tracing out a curve that represents "the path of motion of the system".

In such a configuration space, a task is specified by a set of system point pairs representing the initial and final configurations for different instances of the task. An instance of the task is said to be *solvable* if there exists an algorithm that can compute the final configuration from its initial configuration. The task is said to be solvable if there exists an algorithm that can solve all its instances.

Each instance of the algorithm solving an instance of the task represents a path of the system in the configuration space between the corresponding initial and final system points. If there exists more than one algorithm to solve the task then naturally there might exist more than one path between the two points.

The goal of an agent that is trying to learn a task in such a system is to observe the algorithm instances and infer the algorithm. In this regard, all the information that the agent would get from an algorithm instance is just an initial-final configuration pair along with a series of configuration changes that lead from initial configuration to final configuration. The agent would not be aware of the details of the underlying process

that is responsible for these changes, and has to infer the process purely based on the observations.

The agent is said to have "learned the task" if it can perform the task on its own, by moving the system from any given initial configuration to the corresponding final configuration in the configuration space. It should be noted that the procedure used (the algorithm inferred) by the agent may not be the same as the original algorithm from whose instances it has learned.

It should also be noted that the notion of learning the task, as described above, does not allow any probabilistic or approximate solutions. The agent should be able to perform the task correctly under all circumstances. An additional constraint that we put on the agent is that it should infer the algorithm from as few algorithmic instances as possible. This is important for agents of both real world systems and simulation systems alike, for in case of agents observing samples from real world environment it may not be possible to pickup as many samples as they want, and in case of simulated environments each sample instance incurs an execution cost in terms of time and other resources and hence should be traded sparingly.

We formalize these concepts in the following.

2.1. A Formal Framework

Consider an agent that it trying to learn a task T in a system S whose configuration space is given by

$$\mathcal{C}(S) = \{\vec{s_1}, \vec{s_2}, \ldots, \vec{s_N}\},$$

where each $\vec{s_i}$ is a system point represented with $n-$coordinates $\{q_{i_1}, q_{i_2}, \ldots, q_{i_n}\}$.

Let A be an algorithm to solve the task T, and $A_1, A_2, \ldots, A_k$ be the instances of A solving the instances $T_1, T_2, \ldots, T_k$ of T respectively.

In the configuration space each T_i is represented by a pair of system points $(\vec{s}_{i_1}, \vec{s}_{i_2})$, and the corresponding A_i by a path between those system points.

Let I, F be two operators that when applied to an algorithm instance A_i, yield the corresponding initial and final system points respectively, such as $I(A_i) = \vec{s}_{i_1}$ and $F(A_i) = \vec{s}_{i_2}$. We also define the corresponding set versions of these operators $\vec{I}$ and $\vec{F}$ as following. For all $A' \subseteq \{A_1, A_2, \ldots, A_k\}$,

$$\vec{I}(A') = \{I(A_i) \mid A_i \in A'\},$$

and

$$\vec{F}(A') = \{F(A_i) \mid A_i \in A'\}.$$

The goal of the agent is to perform T, by mimicking or modeling A, inferring A's details from a subset of its instances.

By following the tradition of learning algorithms, let us call A as the *target concept*, and the subset of its instances $D = \{D_1, D_2, \ldots, D_d\} \subseteq \{A_1, A_2, \ldots, A_k\}$ as the *training set* or *data samples*, and the agent as the *learner*. We use the symbol L to denote the learner.

At any instance during the phase of learning the set D can be partitioned into two subsets O, O' such that

$$(O \cup O' = D) \ \wedge \ (O \cap O' = \emptyset).$$

The set $O \subseteq D$ denotes the set of data samples that the learner has already seen, and the set $O' \subseteq D$ denotes the set of data samples the learner has yet to see. Learning progresses by presenting the learner with an unseen sample $D_i \in O'$, and marking it as seen, by moving it to the set O. Starting from $O = \emptyset, O' = D$, this process of transition would be repeated till it becomes $O = D, O' = \emptyset$.

In this process, each data sample D_i decreases the ignorance of the learner L about the target concept A, and hence could be assigned some specific *information content* value that indicates how much additional information L can gain from D_i about A.

We can determine the information content values of data samples by establishing the concept of a *zone,* where we treat an ensemble of system points that share a common relation as a single logical entity.

Definition. A set $Z \subseteq \mathcal{C}(S)$ defines a zone if there exists a function $f : \mathcal{C}(S) \rightarrow \{0, 1\}$ such that for each $\vec{s_i} \in \mathcal{C}(S)$:

$$f(s_i) \quad = \quad \begin{cases} 1 & \text{if } \vec{s_i} \in Z, \\ 0 & \text{if } \vec{s_i} \notin Z. \end{cases}$$

The function f is called the characteristic function of Z.

For the configuration space $\mathcal{C}(S) = \{\vec{s_1}, \vec{s_2}, \ldots, \vec{s_m}\}$, we can construct an equivalent *zone-space* $\mathcal{Z}(S) = \{Z_1, Z_2, \ldots, Z_r\}$, such that the following holds.

$$\forall \vec{s_i}\, \exists Z_i\, [\vec{s_i} \in Z_i] \ \wedge \ \forall_{i=1}^{r} \forall_{j=i+1}^{r} [Z_i \cap Z_j = \emptyset] \ \wedge \ \forall_{i=1}^{r} [Z_i \neq \emptyset] \ \wedge \ \cup_{i=1}^{r} Z_i = \mathcal{C}(S).$$

The zone-space can be viewed as a $m-$dimensional space with each Z_i being a point in it, where m is some function of n whose value depends upon and hence would be decided by the nature of T. Let the range of ith coordinate of this $m-$dimensional space be $[0, r_i]$.

If we use the notation $|P|$ to indicate the size of any set P, then we could represent the volume of the zone-space as

$$vol(\mathcal{Z}(S)) = |\mathcal{Z}(S)| = \prod_{i=1}^{m} r_i.$$

Define an operator $\nabla : \mathcal{C}(S) \rightarrow \mathcal{Z}(S)$ that when applied to a system point in the configuration space yields the corresponding zone in the zone-space. Similarly, let $\vec{\nabla}$ be the corresponding set version of this operator defined as, for all $S' \subseteq \mathcal{C}(S)$,

$$\vec{\nabla}(S') = \{\nabla(\vec{s_i}) \mid \vec{s_i} \in S'\}.$$

At any instance the *knowledge* of L about A depends on the set of samples it has seen till then, and the information content of a data sample depends on whether the sample has already been seen by L or not.

To define formally, the knowledge of the learner, after having seen a set of samples $O \subseteq D$, is given by

$$\mathcal{K}_O(L) \quad = \sum_{Z \in \vec{\nabla}(\vec{I}(O))} |Z|\,.$$

The information content of any data sample $D_i \in D$, after the learner has seen a set of samples $O \subseteq D$, is given by

$$IC_O(D_i) = \begin{cases} |\nabla(I(D_i))| & \text{if } \forall D_j \in O \quad [\nabla(I(D_i)) \neq \nabla(I(D_j))]; \\ 0 & \text{otherwise;} \end{cases}$$

and the information content of all data samples would be given by

$$\overrightarrow{IC}_O(D) \quad = \sum_{Z \in \vec{\nabla}(\vec{I}(D-O))} |Z|\,.$$

It should be noted that the above definitions measure the information content of data samples relative to the state of the learner and satisfy the limiting conditions $\mathcal{K}_\emptyset(L) = 0$, $\overrightarrow{IC}_D(D) = 0$.

The process of learning is essentially a process of transfer of information from data samples to the learner, resulting in a change in the state of the learner. When these changes are infinitesimal, spanning many steps, the transformation process satisfies the condition that the line integral

$$\mathcal{L} = \int_{\emptyset}^{D} E \;\; do, \qquad (2.1)$$

where $E = \overrightarrow{IC}_O(D) - \mathcal{K}_O(L)$, has a stationary value.

This is known as the Hamilton's principle, which states that out of all possible paths by which the learner could move from $\mathcal{K}_\emptyset(L)$ to $\mathcal{K}_D(L)$, it will actually travel along that path for which the value of the line integral (2.1) is stationary. The phrase "stationary value" for a line integral typically means that the integral along the given path has same value to within first-order infinitesimals as that along all neighboring paths (Goldstein, 1980; McCauley, 1997).

We can summarize this by saying that the process of learning is such that the *variation* of the line integral $\mathcal{L}$ is zero.

$$\delta\mathcal{L} = \delta \int_{\emptyset}^{D} E \;\; do = 0.$$

Thus we can formulate the following conservation theorem.

Theorem 1. *The sum* $\mathcal{K}_O(L) + \overrightarrow{IC}_O(D)$ *is conserved for all* $O \subseteq D$.

Proof. We shall prove this by establishing that $\mathcal{K}_{O_i}(L) + \overrightarrow{IC}_{O_i}(D) = \mathcal{K}_{O_j}(L) + \overrightarrow{IC}_{O_j}(D)$ for all $O_i, O_j \subseteq D$.

Consider $O_1, O_2 \subseteq D$ such that $|O_2| - |O_1| = 1$. Let $O_2 - O_1 = \{D_i\}$. To calculate the information content value of D_i, we need to consider two cases.

Case 1. $\nabla(I(D_i)) = \nabla(I(D_j))$ for some $D_j \in O_1$. In such case, $\vec{\nabla}(\vec{I}(O_2)) = \vec{\nabla}(\vec{I}(O_1))$, and hence $IC_{O_1}(D_i) = IC_{O_2}(D_i) = 0$.

$$\begin{aligned} \mathcal{K}_{O_2}(L) &= \textstyle\sum_{Z \in \vec{\nabla}(\vec{I}(O_2))} |Z| \\ &= \textstyle\sum_{Z \in \vec{\nabla}(\vec{I}(O_1))} |Z| \\ &= \mathcal{K}_{O_1}(L). \\ \overrightarrow{IC}_{O_2}(D) &= \overrightarrow{IC}_{O_1}(D) - IC_{O_1}(D) \\ &= \overrightarrow{IC}_{O_1}(D). \\ \mathcal{K}_{O_2}(L) + \overrightarrow{IC}_{O_2}(D) &= \mathcal{K}_{O_1}(L) + \overrightarrow{IC}_{O_1}(D). \end{aligned}$$

Case 2. $\nabla(I(D_i)) \neq \nabla(I(D_j))$ for all $D_j \in O_1$. In such case, $IC_{O_1}(D_i) = |\nabla(I(D_i))|$.

$$\begin{aligned} \overrightarrow{IC}_{O_2}(D) &= \overrightarrow{IC}_{O_1}(D) - IC_{O_1}(D_i) \\ &= \overrightarrow{IC}_{O_1}(D) - |\nabla(I(D_i))|\,. \\ \mathcal{K}_{O_2}(L) &= \textstyle\sum_{Z \in \vec{\nabla}(\vec{I}(O_2))} |Z| \\ &= \textstyle\sum_{Z \in \vec{\nabla}(\vec{I}(O_1 + \{D_i\}))} |Z| \\ &= \sum_{Z \in \vec{\nabla}(\vec{I}(O_1))} |Z| + \sum_{Z \in \{\nabla(I(D_i))\}} |Z| \\ &= \mathcal{K}_{O_1}(L) + |\nabla(I(D_i))|\,. \\ \mathcal{K}_{O_2}(L) + \overrightarrow{IC}_{O_2}(D) &= \mathcal{K}_{O_1}(L) + |\nabla(I(D_i))| + \\ &\quad\; \overrightarrow{IC}_{O_1}(D) - |\nabla(I(D_i))| \\ &= \mathcal{K}_{O_1}(L) + \overrightarrow{IC}_{O_1}(D). \end{aligned}$$

Thus whenever $|O_2| - |O_1| = 1$, it holds that

$$\mathcal{K}_{O_2}(L) + \overrightarrow{IC}_{O_2}(D) = \mathcal{K}_{O_1}(L) + \overrightarrow{IC}_{O_1}(D).$$

Now consider two sets $O_i, O_j \subseteq D$ such that $|O_j| - |O_i| = l$, $l > 1$. Let $O_j - O_i = \{D_{j_1}, \ldots, D_{j_l}\}$. We can construct sets $P_1, \ldots, P_{l-1}$ such that

$$P_1 = O_i \cup \{D_{j_1}\}, \ldots, P_{l-1} = O_i \cup \{D_{j_1}, \ldots, D_{j_{l-1}}\}.$$

Then it holds that

$$|P_1| - |O_i| = |P_2| - |P_1| = \cdots = |O_j| - |P_{l-1}| = 1.$$

However, we have proved that

$$\mathcal{K}_{O_2}(L) + \overrightarrow{IC}_{O_2}(D) = \mathcal{K}_{O_1}(L) + \overrightarrow{IC}_{O_1}(D)$$

whenever $|O_2| - |O_1| = 1$, and hence it follows that

$$\begin{aligned}
\mathcal{K}_{O_i}(L) + \overrightarrow{IC}_{O_i}(D) &= \mathcal{K}_{P_1}(L) + \overrightarrow{IC}_{P_1}(D), \\
\mathcal{K}_{P_1}(L) + \overrightarrow{IC}_{P_1}(D) &= \mathcal{K}_{P_2}(L) + \overrightarrow{IC}_{P_2}(D), \\
&\vdots \\
\mathcal{K}_{P_{l-1}}(L) + \overrightarrow{IC}_{P_{l-1}}(D) &= \mathcal{K}_{O_j}(L) + \overrightarrow{IC}_{O_j}(D),
\end{aligned}$$

and thereby, $\mathcal{K}_{O_i}(L) + \overrightarrow{IC}_{O_i}(D) = \mathcal{K}_{O_j}(L) + \overrightarrow{IC}_{O_j}(D)$.

Hence the sum $\mathcal{K}_O(L) + \overrightarrow{IC}_O(D)$ is conserved for all $O \subseteq D$. □

An important consequence of this theorem is that irrespective of the order of individual samples that L chooses to learn from, the gain in its knowledge would always be equal to the corresponding loss in the information content of the data samples.

$$\mathcal{K}_{O_j}(L) - \mathcal{K}_{O_i}(L) = \overrightarrow{IC}_{O_i}(D) - \overrightarrow{IC}_{O_j}(D).$$

We now define two metrics – *quality* and *quantity* – for the data samples to denote the notions of *necessity* and *sufficiency*.

The metric *quality* measures the relative information strength of individual samples, defined as

$$quality(D) = \frac{|D| - \left|\vec{\nabla}(\vec{I}(D))\right|}{|D|} \times 100\%.$$

Ideally a data sample set should have this value to be 100%. Smaller values indicate the presence of unnecessary samples that do not contribute to learning.

Similarly we define the *quantity* of data samples as

$$quantity(D) = \frac{\left|\vec{\nabla}(\vec{I}(D))\right|}{|\mathcal{Z}(S)|} \times 100\%.$$

This is a sufficiency measure and hence a value less than 100% indicates the insufficiency of data samples to complete the learning.

Theorem 2. *The knowledge of the learner, after completing the learning over data samples D having $quantity(D) = 100\%$, would be equal to the volume of the configuration space $|\mathcal{C}(S)|$.*

Proof. When the $quantity(D) = 100\%$,

$$\left|\vec{\nabla}(\vec{I}(D))\right| = |\mathcal{Z}(S)|.$$

Since $\vec{\nabla}(\vec{I}(D)) \subseteq \mathcal{Z}(S)$,

$$\left|\vec{\nabla}(\vec{I}(D))\right| = |\mathcal{Z}(S)| \Rightarrow \vec{\nabla}(\vec{I}(D)) = \mathcal{Z}(S).$$

From theorem 1 we have,

$$\mathcal{K}_D(L) + \overrightarrow{IC}_D(D) = \mathcal{K}_\emptyset(L) + \overrightarrow{IC}_\emptyset(D).$$

Since $\mathcal{K}_\emptyset(L) = 0$ and $\overrightarrow{IC}_D(D) = 0$,

$$\begin{aligned}
\mathcal{K}_D(L) &= \overrightarrow{IC}_\emptyset(D) \\
&= \textstyle\sum_{Z \in \vec{\nabla}(\vec{I}(D-\emptyset))} |Z| \\
&= \textstyle\sum_{Z \in \vec{\nabla}(\vec{I}(D))} |Z| \\
&= \textstyle\sum_{Z \in \mathcal{Z}(S)} |Z| \\
&= |\mathcal{C}(S)|.
\end{aligned}$$

Thus when $quantity(D) = 100\%$, $\mathcal{K}_D(L) = |\mathcal{C}(S)|$. □

Theorem 3. *The target concept can not be learnt with less than $|\mathcal{Z}(S)|$ number of data samples.*

Proof. Consider a data sample set $D = \{D_1, \ldots, D_d\}$ having $quantity(D) = 100\%$ and $|D| < |\mathcal{Z}(S)|$.

Let $P = \mathcal{Z}(S) - \vec{\nabla}(\vec{I}(D)) = \{P_1, \ldots, P_l\}$, $l \geq 1$. Assume that L has learned the target concept completely from D. Then, by theorem 2, $\mathcal{K}_D(L) = |\mathcal{C}(S)|$, and by theorem 1,

$$\mathcal{K}_D(L) + \overrightarrow{IC}_D(D) = \mathcal{K}_\emptyset(L) + \overrightarrow{IC}_\emptyset(D).$$

Since $\mathcal{K}_\emptyset(L) = 0$ and $\overrightarrow{IC}_D(D) = 0$, it leads to

$$\begin{aligned}
|\mathcal{C}(S)| &= \overrightarrow{IC}_\emptyset(D) \\
&= \textstyle\sum_{Z \in \vec{\nabla}(\vec{I}(D-\emptyset))} |Z| \\
&= \textstyle\sum_{Z \in \vec{\nabla}(\vec{I}(D))} |Z| \\
&= \textstyle\sum_{Z \in (\mathcal{Z}(S)-P)} |Z| \\
&= \textstyle\sum_{Z \in \mathcal{Z}(S)} |Z| - \sum_{Z \in P} |Z| \\
&= |\mathcal{C}(S)| - \textstyle\sum_{Z \in P} |Z|.
\end{aligned}$$

This is not possible unless $P = \emptyset$, in which case it would become $\mathcal{Z}(S) = \vec{\nabla}(\vec{I}(D))$, and $|D| \geq |\mathcal{Z}(S)|$. Hence proved. □

2.2. A Learning Mechanism Based on Data-dependencies

Consider a task instance T_1 with end points $(\vec{s}_1, \vec{s}_2)$ in the configuration space. If we express T_1 as a point function f, then we could write $\vec{s}_2 = f(\vec{s}_1)$. An algorithm instance A_1 that solves T_1 would typically implement the functionality of f thereby representing a path between $\vec{s}_1$ and $\vec{s}_2$. If there exists more than one way to implement f, then there exists more than one path between $\vec{s}_1$ and $\vec{s}_2$. Such a set of paths might be denoted by $f(\vec{s}_1, \alpha)$ with $f(\vec{s}_1, 0)$ representing some arbitrary path chosen to be treated as reference path.

Further, if we select some function $\eta(\vec{x})$ that vanishes at $\vec{x} = \vec{s}_1$ and $\vec{x} = \vec{s}_2$, then a possible set of varied paths is given by

$$f(\vec{x}, \alpha) = f(\vec{x}, 0) + \alpha\, \eta(\vec{x}).$$

It should be noted that all these varied paths terminate at the same end points, that is, $f(\vec{x}, \alpha) = f(\vec{x}, 0)$ for all values of α.

However, when we try to consider another task instance T_2 to be represented with these variations, we need to make them less constrained. The tasks T_1 and T_2 would not have the same end points in the configuration space and hence there would be a variation in the coordinates at those points. We can, however, continue to use the same parameterization as in the case of single task instance, and represent the family of possible varied paths by

$$f_i(\vec{x}, \alpha) = f_i(\vec{x}, 0) + \alpha\, \eta_i(\vec{x}),$$

where α is an infinitesimal parameter that goes to zero for some assumed reference path. Here the functions η_i do not necessarily have to vanish at the end points, either for the reference path or for the varied paths. Upon close inspection, one could realize that the variation in these family of paths is composed of two parts.

1. Variations within a task instance due to different algorithmic implementations.
2. Variations across task instances due to different initial system point configurations.

The learner can overcome the first type of variations by observing that the end points, and their corresponding zones, are invariant to the paths between them. In this regard, all the system points that belong to the same initial, final zone pair could be learned with a single algorithm instance. However, for the second type of variations, the learner may not be able to overcome them without any prior knowledge of the task. All the different instances of the task would have different corresponding zones for their end points and hence they need to be remembered as they are.

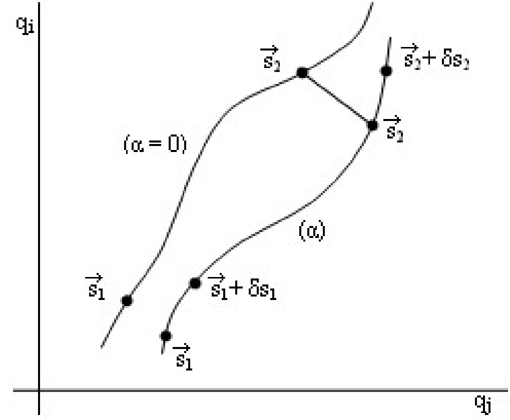

Figure 2. Schematic illustration of path variations across task instances in configuration space

Below we present a mechanism that uses these concepts of variations in the configurations paths to infer solutions to the unknown problem instances based on the solutions to the known problem instances. This results in a learning like behavior where the known problem-solution configuration pairs form the training set samples. Such samples could be collected by the following procedure.

1. For a given problem identify the appropriate configuration space and number of dimensions.
2. Express the solution as a logical relation R_s in terms of coordinates of the configuration space.
3. Use R_s to identify an appropriate characteristic function F_s to form a solution zone.
4. Use F_s in deciding the characteristic functions for other zones and the number of dimensions for zone-space.
5. Define the operator ∇ to map the system points from configuration space to the zones in zone-space.
6. Define an appropriate variation operator δ in the configuration space such that variations in the known problem configurations would give clue to the variations in the solution configurations, such as, $solution(x + \delta x) = solution(x) + \delta x$.
7. Construct the sample problem-solution configuration pairs by using any traditional algorithm. The

samples should be such that all zones are represented.

Once we have all the required data samples with us, the training procedure is simple and straightforward in that all that is needed is to mark each of the sample problem configurations as the reference configuration for the corresponding zone and remembering the respective solution configurations for those references. We can use a memory lookup table to store these reference solutions. The procedure is as follows.

For each data sample $D_i = (\vec{p}_i, \vec{s}_i)$
{
  Let $Z = \nabla(p_i)$;
  RefProbConfig[Z] $= \vec{p}_i$;
  RefSolConfig[Z] $= \vec{s}_i$;
}

Once the training is over, we can compute the solution configuration $\vec{s}$ for any given problem configuration $\vec{p}$ in the configuration space as follows.

1. For the given problem configuration $\vec{p}$, apply the operator ∇ and find the zone $Z = \nabla(\vec{p})$;
2. Get the reference problem configuration $\vec{p}_i$ = RefProbConfig[Z], and compute the variation $\delta(\vec{p}, \vec{p}_i)$;
3. Compute the required solution configuration from the reference solution configuration by applying the variation parameter as:

$$\vec{s} = \text{RefSolConfig}[Z] + \delta(\vec{p}, \vec{p}_i);$$

2.3. An Example Problem

To explain how these concepts of variations in the configuration paths could be used in practice, we consider an example problem of *sorting*. We outline a procedure that implements sorting based on the concepts we have discussed till now.

The reason behind choosing sorting as opposed to any other alternative is that the problem of sorting has been well studied and well understood, and requires no additional introduction. However, it should be noted that our interest here is, rather to explain how zones can be constructed and used for the sorting problem, than to propose a new sorting algorithm; and hence we do not consider any performance comparisons. In fact, the procedure we outline below runs with $O(n^2)$ time complexity requiring $O(2^{n^2})$ memory, thus any performance comparisons would be futile.

To start with, we can consider the task of sorting as being represented by its instances such as $\{(3,5,4),\ (3,4,5)\}$, where the second element $(3,4,5)$ represents the *sorted result* of first element $(3,5,4)$. We can consider these elements as points in a $3-$dimensional space.

Thus in general given an array of n integers to be sorted, we can form a system with $n-$coordinate axes resulting in an $n-$dimensional configuration space. If we assume that each element of the array can take a value in the range $[0, N]$, where N is some maximum integer value, then there would be a total of N^n system points in the configuration space. That is, following our notation from section 2.1, $|\mathcal{C}(S)| = N^n$. To construct the corresponding zone-space for this configuration space, consider the following mathematical specification for sorting,

$$\forall_{i=1}^{n} \forall_{j=1}^{n} \ [\, i < j \ \Rightarrow \ a[i] < a[j]\,],$$

where a is an array with n integers. This specification represents a group of conditions that need to be satisfied by the array if it has to considered as being in sorted order. Now, we can use this specification in identifying the following.

1. Number of dimensions of zone-space: The specification involves two quantifiers $\forall_{i=1}^{n}$ and $\forall_{j=1}^{n}$, with an additional constraint $i < j$. Thus the valid values could be $i = 1, \ldots, n, j = i+1, \ldots, n$, resulting in a group of $n \times (n-1)/2$ conditions to be accounted for. Each condition would form one coordinate axis in the zone-space and hence we have $n \times (n-1)/2$ axes.
2. Range of each axis of zone-space: Since each axis is formed out of the condition $(a[i] < a[j])$, with various values of i, j representing various axes, the range of each axis would be defined by the number of possible conditions $(a[i] < a[j]), (a[i] = a[j])$ and $(a[i] > a[j])$, which is *three*. Hence the range of each axis $r_i = 3$.
3. Operator ∇ : Each zone is a point in zone-space with $n \times (n-1)/2$ coordinates. To find these coordinate values we need to evaluate $n \times (n-1)/2$ conditions (one for each axis) as below.

```
for(int i=0,r=0; i<n; ++i)
    for(int j=i+1; j<n; ++j,++r)
        ZCoord[r]=((a[i]=a[j])?0:((a[i]<a[j])?1:2));
```

4. Variation operator δ : We implement the variation operator using the differences between relative array positions of numbers before and after sorting. We can use the array indexing and de-indexing

operations for this purpose. For example, sorting $a = \{3, 5, 4\}$ produces $\vec{a} = \{3, 4, 5\}$, which gives us a variation in the indices of elements from $(0, 1, 2)$ to $(0, 2, 1)$. Thus we can use our variation operator to express $\vec{a}$ as, $\vec{a} = \{\, a[0],\ a[2],\ a[1]\, \}$.

Once we have these necessary operators with us, we can start assigning the reference $(unsorted, sorted)$ configuration pairs for each zone by using any traditional sorting algorithm such as *heapsort* or *quicksort*, as shown below.

```
for(int i=0; i < nSamples; ++i)
{
    GetZCoord(Unsorted[i], ZCoord);
    quicksort(Unsorted[i], Sorted[i]);
    SetRefConfig(ZCoord, Unsorted[i], Sorted[i]);
}
```

It should be noted that we have $3^{n\times(n-1)/2}$ zones in the zone-space and hence we need so many sample $(unsorted, sorted)$ pairs as well. However, once we complete the training with all those samples, we can use the following procedure to sort any of the N^n possible arrays.

```
void LSort(int nArray[], int nSize, int nSorted[])
   {
      GetZCoord( nArray, ZCoord );
      GetRefConfig( ZCoord, RefProb, RefSol);
      for(int i=0; i < nSize; ++i)
         nSorted[i] = nArray[RefSol[i]];
   }
```

2.4. Limitations

Having presented the mechanism for learning based on the concepts of variations in the system configuration paths, here we discuss the limitations of this approach.

Disadvantages:

1. As could be easily understood, the concepts of *configuration space* and *zone-space* form the central theme of this approach. However, it may not be always possible to come up with appropriate configuration space or zone-space for any given problem. In fact, for many tasks such as face recognition etc... we readily do not have any clues for logical relations among the data attributes. This is one of the biggest limitations of this approach.
2. To present the learner with some sample configurations, we assumed the existence of an algorithm that could solve the task at hand. However, this assumption may not hold at all times. Once again, face recognition is an example.
3. The memory requirements are too high. We have already seen that we need $3^{n\times(n-1)/2}$ samples to correctly learn the sorting task.

However, given the goal of mimicking a human being and the scope of abstract concepts the agents have to learn from human beings, and given the virtually unlimited number of problem instances that could be solved by this learning mechanism, the memory requirements should not become a problem at all (note that the memory requirements do not depend on N but on n, so there is no upper limit to the number of problem instances that can be solved correctly). Further advantages are as follows.

Advantages:

1. Independent of the order of training data samples. In this method, the learner is invariant to the order in which it receives the data samples. All that a learner does with a data sample is, compute the corresponding zone and mark the sample as a reference for that zone. This process clearly is independent of the order of the data samples and hence gives the same results in all circumstances. It should be noted that the traditional learning algorithms does not guarantee any such invariance.
2. Additional samples do not create any bias. If there exists more than one sample per zone, the characteristic functions of zones guarantee that they all would produce the same results as that of first sample. Hence the training would not be biased by the presence of additional samples. Further, a sample could be repeated as many times as one wants without affecting the training results. This is useful for situations where a robot might be learning from real world, where some typical observations (such as the changing traffic lights, flow of vehicles etc...) would get repeated more frequently compared with some rare observations (such as earth quakes or accidents etc...). Traditional learning algorithms fail to provide unbiased results in such situations.
3. Non-probabilistic metrics and accurate results. To meet the demands of artificial systems, the metrics we have devised are completely deterministic and are void of any probabilistic assumptions and thus can be adapted to any suitable system.

4. Expandable to multi-task learning. Though we have concentrated on learning a single task in this discussion, there is nothing in this method that could prevent the learner from learning more than one task at the same time. For example, once an agent learns to sort in ascending order ($SASC$), it can further learn to sort in descending order ($SDSC$) simply by computing the new variation operator δ_{SDSC} directly from (δ_{SASC}), instead of from new sample problem-solution configuration pairs. This saves the training time and cost for $SDSC$. However, to implement this feature the agent should be informed of the relation between the tasks a priori. Smart agents that can automatically recognize the relation among tasks based on their configuration spaces should be an interesting option to explore further in this direction.
5. Knowledge transfer. All the knowledge of the learner is represented in terms of reference configurations for individual zones. Any learner who has access to these reference configurations can perform equally well as the owner of the knowledge itself, without the need to go through all the training again. This could lead to the concept of *tradable knowledge* resources for agents.
6. Perfect partial learning. Just as additional samples do not create bias, lack of samples also would not create problems for learning. A training set with *quantity* less than 100% would still give correct results as long as the problem instance at hand is from one of the learnt zones. That is, whatever the agent learns, it learns perfectly. This feature comes handy to implement low cost *bootstrapping robots* with reduced features and functionality which can be used as "data sample suppliers" for other full-blown implementaions. This concept of bootstrapping robots is one of the fundamental concepts of artificial life study in that it might invoke the possibility of self-replicating robots (Freitas & Gilbreath, 1980; Freitas & Merkle, 2004).

3. Conclusions

The notion of data-independence for an algorithm speaks for constant execution paths across all its instances. A variation in execution path is generally attributable to the variations in the nature of data. When a problem and its solution are viewed as points in a system configuration, variations in the problem configurations can be used to study the variations in the solution configurations and vice versa. These variations could be used to infer solutions to unknown instances of problems based on the solutions to the known instances.

This paper analyzed the problem of data-dependencies in the learning process and presented a learning mechanism based on the relations among data attributes. The mechanism constructs a Cartesian hyperspace, namely the configuration space, for any given task, and finds a set of paths from the initial configuration to final configuration that represents different instances of the task. As part of the learning process the learner gradually gains information from data samples one by one, till all data samples were processed. Once such a transfer of information is complete, the learner can solve any instance of the task without any restrictions.

The mechanism presented is independent of the order of data samples and has the flexibility to be expandable to multi-task learning. However, the practicality of this approach may be hindered by the lack of appropriate algorithms that could provide sample instances. Further study to eliminate such bottlenecks could make this a perfect choice to implement learning behavior in artificial agents.

References

Angluin, D. (1992). Computational learning theory: survey and selected bibliography. *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing* (pp. 351–369). New York: ACM Press.

Balmer, M., Cetin, N., Nagel, K., & Raney, B. (2004). Towards truly agent-based traffic and mobility simulations. *AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems* (pp. 60–67). Washington, DC, USA: IEEE Computer Society.

Brooks, R. A. (1991). Intelligence without reason. *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)* (pp. 569–595). San Mateo, CA, USA: Morgan Kaufmann publishers Inc.

Brugali, D., & Sycara, K. (2000). Towards agent oriented application frameworks. *ACM Computing Surveys*, *32*, 21–27.

Bryson, J. J. (2003). Action selection and individuation in agent based modelling. *Proceedings of Agent 2003: Challenges of Social Simulation.*

Cliff, D., & Grand, S. (1999). The creatures global digital ecosystem. *Artificial Life*, *5*, 77–93.

Collins, J. C. (2001). *On the compatibility between physics and intelligent organisms* (Technical Report DESY 01-013). Deutsches Elektronen-Synchrotron DESY, Hamburg.

Decugis, V., & Ferber, J. (1998). Action selection in an autonomous agent with a hierarchical distributed reactive planning architecture. *AGENTS '98: Proceedings of the second international conference on Autonomous agents* (pp. 354–361). New York, NY, USA: ACM Press.

Franklin, S. (2005). A "consciousness" based architecture for a functioning mind. In D. N. Davis (Ed.), *Visions of mind*, chapter 8. IDEA Group INC.

Freitas, R. A., & Gilbreath, W. P. (Eds.). (1980). *Advanced automation for space missions*, Proceedings of the 1980 NASA/ASEE Summer Study. National Aeronautics and Space Administration and the American Society for Engineering Education. Santa Clara, California: NASA Conference Publication 2255.

Freitas, R. A., & Merkle, R. C. (2004). *Kinematic self-replicating machines*. Georgetown, TX: Landes Bioscience.

Goldstein, H. (1980). *Classical mechanics*. Addison-Wesley Series in Physics. London: Addision-Wesley.

Kamareddine, F., Monin, F., & Ayala-Rincón, M. (2002). On automating the extraction of programs from proofs using product types. *Electronic Notes in Theoretical Computer Science*, *67*, 1–21.

Katsuhiko, T., Takahiro, K., & Yasuyoshi, I. (2002). Translating multi-agent autoepistemic logic into logic program. *Electronic Notes in Theoretical Computer Science*, *70*, 1–18.

Kearns, M. J. (1990). *The computational complexity of machine learning*. ACM Distinguished Dissertation. Massachusetts: MIT Press.

Lau, T., Domingos, P., & Weld, D. S. (2003). Learning programs from traces using version space algebra. *K-CAP '03: Proceedings of the international conference on Knowledge capture* (pp. 36–43). New York, USA: ACM Press.

Laue, T., & Röfer, T. (2004). A behavior architecture for autonomous mobile robots based on potential fields. *RoboCup 2004*. Springer-Verlag.

Littlestone, N. (1987). Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, *2*.

Lopez, R., & Armengol, E. (1998). Machine learning from examples: Inductive and lazy methods. *Data & Knowledge Engineering*, *25*, 99–123.

Maes, P. (1989). How to do the right thing. *Connection Science Journal*, *1*.

McCauley, J. (1997). *Classical mechanics*. Cambridge University Press.

Moses, Y. (1992). Knowledge and communication: A tutorial. *TARK '92: Proceedings of the 4th conference on Theoretical aspects of reasoning about knowledge* (pp. 1–14). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Raedt, L. D. (1997). Logical settings for concept-learning. *Artificial Intelligence*, *95*, 187–201.

Ramamurthy, U., Franklin, S., & Negatu, A. (1998). Learning concepts in software agents. *From Animals to Animats 5: Proceedings of The Fifth International Conference on Simulation of Adaptive Behavior*. Cambridge: MIT Press.

Ray, T. S. (1991). *Artificial life ii*, chapter An Approach to the Synthesis of Life. Newyork: Addison-Wesley.

Ray, T. S. (1994). Evolution, complexity, entropy and artificial reality. *Physica D*, *75*, 239–263.

Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, *21*, 25–34.

Schmidhuber, J. (2000). *Algorithmic theories of everything* (Technical Report IDSIA-20-00 (Version 2.0)). Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, Manno-Lugano, Switzerland.

Wilson, S. W. (1994). Zcs: a zeroth level classifier system. *Evolutionary Computation*, *2*, 1–18.

Zurek, W. H. (1989). Algorithmic randomness and physical entropy. *Physical Review A*, *40*, 4731–4751.

Author Index

Aguilar, Ramiro 59
Alonso, Luis 59
De Raedt, L. 37
Frasconi, P. 37
GopalaKrishna, Palem 69
Kitzelmann, Emanuel 15
López, Vivian 59
Monakhova, Emilia 29
Monakhov, Oleg 29
Moreno, María N. 59
Muggleton, Stephen 9
Passerini, A. 37
Rao, M. R. K. Krishna 51
Schmidhuber, Jürgen 11
Schmid, Ute 15
Wysotzki, Fritz 13