TLP **15** (6): 783–817, 2015. © Cambridge University Press 2014
doi:10.1017/S147106841400009X First published online 14 May 2014

Predicate logic as a modeling language: modeling and solving some machine learning and data mining problems with IDP3

MAURICE BRUYNOOGHE, HENDRIK BLOCKEEL, BART BOGAERTS, BROES DE CAT, STEF DE POOTER, JOACHIM JANSEN, ANTHONY LABARRE, JAN RAMON and MARC DENECKER

Department of Computer Science, KU Leuven, Heverlee, Belgium
(*e-mail:* Maurice.Bruynooghe@cs.kuleuven.be, Hendrik.Blockeel@cs.kuleuven.be, Bart.Bogaerts@cs.kuleuven.be, Broes.DeCat@cs.kuleuven.be, Stef.De.Pooter@cs.kuleuven.be, Joachim.Jansen@cs.kuleuven.be, labarre.anthony@gmail.com, Jan.Ramon@cs.kuleuven.be, Marc.Denecker@cs.kuleuven.be)

SICCO VERWER

Institute for Computing and Information Sciences, Radboud Universiteit Nijmegen, Toernooiveld, Nijmegen, the Netherlands
(*e-mail:* siccoverwer@gmail.com)

submitted 22 March 2013; revised 3 February 2014; accepted 6 March 2014

Abstract

This paper provides a gentle introduction to problem-solving with the IDP3 system. The core of IDP3 is a finite model generator that supports first-order logic enriched with types, inductive definitions, aggregates and partial functions. It offers its users a modeling language that is a slight extension of predicate logic and allows them to solve a wide range of search problems. Apart from a small introductory example, applications are selected from problems that arose within machine learning and data mining research. These research areas have recently shown a strong interest in declarative modeling and constraint-solving as opposed to algorithmic approaches. The paper illustrates that the IDP3 system can be a valuable tool for researchers with such an interest. The first problem is in the domain of stemmatology, a domain of philology concerned with the relationship between surviving variant versions of text. The second problem is about a somewhat related problem within biology where phylogenetic trees are used to represent the evolution of species. The third and final problem concerns the classical problem of learning a minimal automaton consistent with a given set of strings. For this last problem, we show that the performance of our solution comes very close to that of the state-of-the art solution. For each of these applications, we analyze the problem, illustrate the development of a logic-based model and explore how alternatives can affect the performance.

KEYWORDS: knowledge representation and reasoning, declarative modeling, logic programming, knowledge base systems, FO(·), IDP system, stemmatology, phylogenetic tree, deterministic finite state automaton

1 Introduction

In his seminal paper, Kowalski (1974) proposed to use first-order predicate logic (FO) as a programming language. He argued that it is possible to use deduction for computation by associating a procedural interpretation to the Horn clause subset of first-order logic. These ideas found their incarnation in the language Prolog.

Whereas Prolog uses deduction as an inference method, other inference methods also exist. Most prominent is the model generation as used in propositional SAT solvers. Also, the inference method of Constraint Programming (CP) can be considered as model generation; indeed, its solvers attempt to assign values to variables while satisfying a set of constraints. The last decades have witnessed tremendous progress in solver technology for Constraint Programming and SAT solving. In Constraint Programming, this progress is at the basis of a shift from Constraint Programming to Constraint Modeling.[1] Notorious examples are Essence (Frisch *et al.* 2008) and Zinc (Marriott *et al.* 2008). Within logic programming, the introduction of stable semantics (Gelfond and Lifschitz 1988) eventually led to the Answer Set Programming (ASP) paradigm (Brewka *et al.* 2011) that, similar to SAT, uses model generation instead of deduction for inference. Many ASP-based systems exist. Examples are DLV (Leone *et al.* 2002), `clasp` (Gebser *et al.* 2007) and Smodels (Syrjänen and Niemelä 2001).

All this progress raises the question as to what is the status of logic as a modeling language. SAT is restricted to propositional logic. It can be considered as the assembler language for modeling. Indeed, there are many examples of programs that generate SAT encodings to obtain the state-of-the-art solvers for various classes of problems. One can find examples in the areas of planning and generating deterministic finite automata, to name just a few. However, SAT is not suited as a language for developing models. For what concerns ASP, it is an expressive high-level language, but it is not based on predicate logic. Today, many intricacies of stable model semantics are hidden in high-level ASP constructs such as constraints and choice rules; however, its two forms of negation ("not" and strong negation) (Brewka *et al.* 2011) clearly distinguish it from the first-order logic; the deviation from first-order-logic semantics could be an obstacle for newcomers.

Historically, predicate logic was always viewed as a very expressive modeling language. This is remarkable, given that anyone who used it for modeling a practical domain will have experienced its inconvenience in expressing certain common propositions. A clear weakness is in expressing inductively definable concepts such as the transitive closure of a binary relation. Another deficiency is in expressing bounds on the cardinality or the sum of sets. Practical modeling languages in Constraint Programming or ASP therefore support some of these propositions. While ASP can express inductive definitions, it is built on radically different foundations than FO. A more conservative solution that preserves FO's foundations is to extend it

[1] In this paper, we use the word model in two different meanings. First, a model is a structure that satisfies the theory, as in "model generation." Second, a model is the result of modeling a problem domain. It is a theory in logic, a formal specification of the problem domain. It should be clear from the context what is intended.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

with suitable language constructs. For instance, it was argued in several works, for example by Denecker and Ternovska (2008) and Denecker and Vennekens (to appear), that a rule set formalism under an extension of the well-founded semantics (Van Gelder *et al.* 1991) is a natural formalism to express the most common forms of inductive definitions. Such a formalism can be integrated with FO in a conceptually clean way. The resulting logic was named FO(ID) by Denecker and Ternovska (2008). The link between FO(ID) and ASP was recently studied by Denecker *et al.* (2012). Below, we use the notation FO(·) to denote the family of extensions of first-order logic.

In this paper, we explore the use of $\text{FO}(\cdot)^{\text{IDP3}}$, the instance of the FO(·) family that is supported by IDP3, the current version of the IDP Knowledge Base System (De Pooter *et al.* 2011). $\text{FO}(\cdot)^{\text{IDP3}}$ extends first-order logic with inductive definitions, partial functions, types and aggregates. The IDP system supports model generation and model expansion (Mitchell and Ternovska 2005; Wittocx *et al.* 2013) as inference methods and is one of the fastest such systems (Calimeri *et al.* 2011). Particular to the modeling language used here is the combination of a purely declarative modeling language with a procedural language that handles interaction with the outside world. Indeed, in contrast to Prolog, the control of the search can be left to the solver, and the user can concentrate on modeling. As we will illustrate in the paper, this does not mean that any correct model will do; when performance matters, models have to be designed with care.

As for the organization of the paper, we start Section 2 with recalling the FO(·) family of extensions of predicate logic. Next, we introduce the $\text{FO}(\cdot)^{\text{IDP3}}$ instance of FO(·) and the IDP knowledge base system that supports $\text{FO}(\cdot)^{\text{IDP3}}$ as a modeling language. The section continues with the shortest path problem as an illustrative example. After describing a very basic model, it explores how alternative models affect the performance of the underlying solver.

The other sections explore the use of the IDP system for solving some real-world problems encountered in the domain of machine learning and data mining by some of the authors. Researchers in this domain have become increasingly aware of the fact that data analysis problems come in many different variants, which do not always fit the standard algorithms well. It is infeasible to develop algorithms for each specific variant, but recently it has been shown that some standard data mining problems, as well as their variants, can be modeled as constraint problems, and solved by general-purpose solvers with performance comparable to that of dedicated algorithms (Guns *et al.* 2011). Our discussion on the use of IDP for three different tasks adds support for the claim that declarative modeling may have an important role to play in machine learning and data mining.

The first task, addressed in Section 3, is in the domain of a stemmatology, a part of philology that studies the relationship between surviving variant versions of a text. A stemma is a family tree that shows how different copies of the same text relate to each other. These copies – manuscripts – are not identical, but evolve. Manuscripts often do not have a single parent, different parts can be copied from different parents, so a stemma is in fact a directed acyclic graph. A typical task is to analyze the plausibility of a stemma hypothesis. For this task, the philologist

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

collects datasets describing features of the text. The values of a feature represent variant readings of a fragment of the text. A common assumption is that the stemma has, for each variant, a unique manuscript that is the source of the variant. The feature value is unknown for some manuscripts, and the question is whether these unknown values can be assigned such that there is indeed a unique source for each feature value. In working out this task, we also illustrate how the procedural side of FO(·)$^{\text{IDP3}}$ allows the user to organize a complete workflow.

The second task (Section 4), although in the very different domain of biology, is somewhat related to the previous one as it is concerned with phylogenetic trees. Phylogeny is an area in which many problems arise. Several problems have been tackled by means of ASP, see Erdem (2011) for an overview. Here we address a new problem in this area. Phylogenetic trees have in their leaves a set of current species and the tree represents the evolutionary relationship between them. Often there are several equally plausible evolutionary explanations and hence different phylogenetic trees. The question addressed here is: what is the minimal supergraph that represents each of the individual trees? What makes the problem difficult is that the correspondence between the internal nodes of different trees is unknown and has to be guessed. Different guesses result in different supergraphs.

In Section 5, we study the well-known problem of learning a minimal deterministic finite state automaton (DFA) that is consistent with a given set of accepted and rejected strings. This is a classical machine learning task for which competitions are organized. The state-of-the-art method (Heule and Verwer 2010; Heule and Verwer 2012), winner of the 2010 Stamina competition (Stamina 2010), solves it by a problem-specific program that iteratively creates a SAT encoding and applies a SAT-solver for an increasing number of states until a model is found. Here we explore to what extent a high level FO(·) formalization can compete with a laboriously constructed encoding as a propositional SAT problem.

These three problems can be abstracted as graph problems and are nondeterministic polynomial time (NP)-complete. Solving them inherently involves search; heuristics are needed to guide the search toward solutions. Developing an algorithm in a procedural language is time-consuming, error-prone and challenging. The use of a declarative modeling language liberates the programmer from the task and allows him to devote more time to proper formalization. Moreover, the default heuristics of the underlying solvers are often sufficient to obtain adequate solutions.

In Section 6, we reflect on our achievements and discuss where there is potential for further improvement. Some of the material in this paper is based on the work of Blockeel *et al.* (2012), and for stemmatology, on the work of Andrews *et al.* (2012).

2 FO(·) and the IDP system

First-order logic has a long tradition and a well-understood semantics but also some limitations with regard to its expressiveness, which makes it not so well suited as a language for knowledge representation. The most notorious problem is that it cannot naturally express transitive closures such as "x is reachable from y if either x and y are connected or there exists a z such that x and z are connected and x is

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

reachable from z." Note that Prolog programmers can cope with transitive closure and that the least Herbrand interpretation captures its meaning; actually, in the first years of logic programming, many Prolog programmers did not realize that it was an issue in the knowledge representation community; at the same time, many in the latter community were ignorant about Prolog's expressiveness. In the knowledge representation community, there are two ways to work around the limitation. On the one hand, one can introduce knowledge representation languages with a semantics different from first-order logic; on the other hand, one can enhance first-order logic with additional constructs. The former approach is taken by the ASP community; the latter approach has been advocated in Denecker and Ternovska (2008), where first-order logic was extended with (not necessarily monotone) inductive definitions. It was argued that this extension resulted in a very natural and expressive language whose meaning was captured by a generalization of the well-founded semantics introduced by Van Gelder *et al.* (1991). This extension was named FO(ID) and later work used the notation FO(·) for a family of languages extending first-order logic.

The FO extension used as a modeling language throughout this paper includes not only inductive definitions but also partial functions, types and aggregates. It is the extension supported by the IDP3 version of the IDP Knowledge Base System that was for the first time described by De Pooter *et al.* (2011).[2] We denote this extension as $\mathrm{FO}(\cdot)^{\mathtt{IDP3}}$.

Functions, of which constants are a special case, are very convenient in modeling. They are used in all the modeling examples of the paper. While n-ary functions can be considered as syntactic sugar for predicates with $n + 1$ arguments, the use of functions makes models more concise and readable. Indeed, when functions are represented as predicates, the functional dependency between the input arguments and the result needs to be represented as a separate constraint. Partial functions give extra flexibility to the modeler. An example can be found in the model of finite state automata in Section 5, where a state doesn't need a transition for all symbols in the automaton's input alphabet.

In almost all applications, the universe is not uniform but contains different types. Relations are typed. Quantification is "naturally" typed, namely, we quantify over objects of a type. It is not difficult to make typing explicit in untyped predicate logic. Yet, it requires an extra discipline of the user to make the types in her quantifications and her relations and functions explicit. By introducing an explicit type system, even a simple many-sorted type system, and a type checking and inference system (to discover type clashes and to guess the types of variables and/or parameters), theories become more compact and graceful. Moreover, a number of bugs can be detected: syntactic errors in variable names, swapped or missing arguments, unintended reuse of variables etc. This is common wisdom. Indeed, well-typed theories go wrong less often (Milner 1978). The type system of IDP3 is not needed from a computational point of view, but for above-mentioned reasons, we often find it convenient.

[2] The examples used throughout the paper make use of IDP version 3.2.0.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

Aggregates are another extensions that contribute to the readability and conciseness of models. Consider, for example, the constraint expressing the functional dependency. One needs to express that there is exactly one value (or in the case of a partial function at most one value) for each combination of input arguments. The availability of aggregates makes it a lot more convenient to express such constraints. A study about the semantics of aggregates in definitions, including the case of recursion, has been made by Pelov *et al.* (2007).

2.1 *The logical components of an* $\mathbf{FO}(\cdot)^{\mathbf{IDP3}}$ *model*

In this section, we introduce the basic notions of an $\mathrm{FO}(\cdot)^{\mathtt{IDP3}}$ model. We restrict ourselves to what is needed to understand the examples later on in the paper.

An $\mathrm{FO}(\cdot)^{\mathtt{IDP3}}$ model comprises a number of logical components, namely vocabularies, structures, terms and theories. A *vocabulary* declares the symbols to be used.[3] A *structure* is used to specify the domain and the data; it can be viewed as a sort of database, it provides a partial (three-valued) interpretation of the symbols in the vocabulary. In the context of optimization problems, a *term* component declares the numerical cost term to be optimized. A *theory* comprises FO formulas and definitions. A *definition* is a set of *rules* of the form $\forall \bar{x} : P(\bar{x}) \leftarrow \varphi(\bar{x})$. where φ is an $\mathrm{FO}(\cdot)^{\mathtt{IDP3}}$ formula.[4] An $\mathrm{FO}(\cdot)^{\mathtt{IDP3}}$ *formula* differs from FO formulas in two ways. First, $\mathrm{FO}(\cdot)^{\mathtt{IDP3}}$ is a many-sorted logic: every variable has an associated *type* and every type has an associated domain. Moreover, it is order-sorted: types can be subtypes of others. Second, besides the standard terms in FO, $\mathrm{FO}(\cdot)^{\mathtt{IDP3}}$ formulas can also have aggregate terms: functions over a set of domain elements and associated numeric values that map to the sum, product, cardinality, maximum or minimum value of the set.

We write $\mathscr{M} \models T$ to denote that structure $\mathscr{M}$ satisfies theory T. With $x^{\mathscr{M}}$, we denote the interpretation of x under $\mathscr{M}$, where x can be a formula or a term. Without going in full details, $\mathscr{M}$ satisfies T when (i) every FO formula F of T is satisfied in $\mathscr{M}$ ($F^{\mathscr{M}}$ is true), and (ii) every definition of T is satisfied in $\mathscr{M}$. A structure $\mathscr{M}$ satisfies a definition D when the well-founded model construction on D (Van Gelder *et al.* 1991) that starts from O, the restriction of M to the predicates not defined in D, results in $\mathscr{M}$. See De Cat *et al.* (2014) for more details.

2.2 *The* IDP3 *system*

The IDP3 system (De Pooter *et al.* 2011) is a *Knowledge Base System* (KBS) that intends to offer the user a range of inference methods, such as model expansion, optimization, verification, symmetry breaking and grounding, and to make use of different state of the art technologies, including SAT, SAT Modulo Theories (Nieuwenhuis *et al.* 2006), Constraint Programming and various technologies from logic programming.

[3] Contrary to Prolog and ASP, the first character of a symbol has no bearing on its kind.
[4] Definitions have a lot in common with pure Prolog rules.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

In this paper, we make use of the inference methods model expansion, satisfiability checking and model minimization. The most important inference method is *model expansion* discussed by Mitchell and Ternovska (2005) and further extended by Wittocx *et al.* (2013). The idea of model expansion is to extend a partial structure (an interpretation) into a full structure that satisfies all constraints specified by the FO(·)$^{\text{IDP3}}$ model. More formally, the task of model expansion is, given a vocabulary V, a theory T over V and a partial structure S over V (at least interpreting all types), to find a structure $\mathcal{M}$ that satisfies T and expands S, i.e., $\mathcal{M}$ is a model of the theory and the input structure S is a subset of $\mathcal{M}$. In the IDP3 system, this task is executed by modelexpand(T,S). The result of the modelexpand procedure is a list of models of T that expands S. If the option nbmodels is set to a value n different from 0, IDP3 will stop searching for more models once it has found n models.

Satisfiability checking is related to model expansion. Calling sat(T,S) in the IDP3 system will return true if and only if modelexpand(T,S) would have returned at least one model. However, since we are not interested in the actual models, some optimizations can be done to speed up this inference.

In case of *model minimization*, also a numerical cost term t is given. The task is to find a model $\mathcal{M}$ of T that expands S such that, for all other models $\mathcal{M}$' expanding S, $t^{\mathcal{M}} \leqslant t^{\mathcal{M}'}$. Model minimization is activated by minimize(T,S,t) with t referring to the term component defining the term.

The IDP3 system allows users to specify FO(·)$^{\text{IDP3}}$ problem descriptions. The basic overall structure of the logical components is as in the following schema.

```
vocabulary V  { ... }        theory T: V    { ... }
term t : V    { ... }        structure S: V { ... }
```

This schema defines a vocabulary V, which is then used as a context in the theory T, the term t and the structure S. In general, several vocabularies can be defined, even extending other vocabularies.

We use IDP syntax in the examples throughout the paper. Each IDP operator has an associated logical operator, the main (non-obvious) operators being: &(∧), |(∨), ~(¬), !(∀), ?(∃), <=>(≡), ~=(≠).

A distinguishing feature of FO(·)$^{\text{IDP3}}$ models is that they not only comprises logical components but also have one or more procedural components. These procedural components comprises procedural code that can perform actions. Actions include the execution of an inference method on a particular logical theory, but also the presentation of results to the user. Procedures allow to glue together a sequence of actions in a process that performs a task for the user. The convention is that the user's task is performed by invoking the procedure main(). Such a task can start with procedural code to prepare one or more structures from input files or databases, continue with performing a number of inference task on combinations of theories with structures and end with presenting the results to the user. The procedural language has to be a flexible and extensible scripting language that offers a smooth integration with the C++ solvers of the IDP system. The IDP system (De Pooter *et al.* 2011) makes use of the Lua (Ierusalimschy *et al.* 1996) scripting language for

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

this purpose. It allows us to treat the various logical components of an FO(·)$^{\text{IDP3}}$ theory as objects that can be manipulated from within the procedures.

More information on the IDP system and in particular its IDP3 version as given by Wittocx *et al.* (2008) and De Pooter *et al.* (2011) can be found at `http://dtai.cs.kuleuven.be/krr/software/idp3`.

2.3 An example: the shortest path problem

As an illustration, we model the shortest path problem (Listing 1). The *vocabulary* comprises a single type, two constants and three predicates. The *structure* specifies the given graph: the interpretation of the type `node` (the domain elements `A`, `B`, `C` and `D`) and the predicate `edge(node,node)` (the domain atoms `edge(A,B)`, `edge(B,C)`, `edge(C,D)` and `edge(A,D)`) as well as the constants `from` (the domain element `A`) and `to` (the domain element `D`), which identify the begin- and endpoint of the path searched for. The predicate `edgeOnPath(node,node)` is used to represent the edges that participate in the shortest path. It provides the base case of the transitive relation `reaches(node,node)`, which is defined in the *theory* component. Definitions are given between "{" and "}." Note that we use the most basic definition for transitive closure: we join the `reaches` relation with itself.

Besides this inductive (recursive) definition, the theory also specifies the constraints expressing that the `edgeOnPath/2` atoms included in a model of the theory indeed compose a simple path from `from` to `to`. The first constraint, a universally quantified implication, ensures that `edgeOnPath/2` atoms are indeed `edge/2` atoms. This constraint explicitly mentions the type of the quantified variables; however, this is optional; these types can be inferred from the type declarations of the predicates of the formula. The types of the quantified variables are omitted in the following constraints. The second constraint, a simple fact, imposes that `reaches/2` includes the pair `(from,to)`. The third constraint, a conjunction of negated formulas, states that the `edgeOnPath/2` atoms should neither include an edge arriving in `from` nor an edge leaving `to`. The fourth constraint is a universally quantified conjunction of two cardinality constraints; it expresses that every node has less than two incoming edges and less than two outgoing edges (i.e., the path is simple). The notation `?<2 y : edgeOnPath(y,x)` means that there are strictly less than two y's that have an edge to x in the path. This is syntactic sugar for the aggregate `#{ y : edgeOnPath(y,x)} < 2`. This aggregate is a more concise formulation of the FO constraint `! y1 y2 : edgeOnPath(y1,x) & edgeOnPath(y2,x) => y1=y2`. Finally, the last constraint, another universally quantified implication, states that the endpoints of selected edges are reachable from `from` (i.e., no edges are selected that do not contribute to the path).

The *term* component of the model defines the term `lengthOfPath` as an aggregate expression counting the number of tuples in the `edgeOnPath` relation of a model of the theory. Minimizing this term ensures that the path described in a model of the theory is indeed the shortest path.

The *procedure* component shows the Lua code invoking the model minimization task and printing the result. The Lua code treats the logical components as first-class

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

citizens and uses them as parameters in the method call activating the solver. The annotation `[1]` directs the solver to return at most one solution.

Listing 1. Calling main() solves the shortest path problem for the given data.

```
vocabulary sp_voc {
    type node
    from, to: node
    edge(node, node)
    edgeOnPath(node, node)
    reaches(node, node)
}
theory sp_theory1 : sp_voc {
    {! x y : reaches(x,y) <- edgeOnPath(x,y).
     ! x y z : reaches(x,y) <- reaches(x,z) & reaches(z,y).}

    ! x[node] y[node] : edgeOnPath(x,y) => edge(x,y). // (1)
    reaches(from, to). // (2)
    ~(? x : edgeOnPath(x,from)) & ~(? x : edgeOnPath(to,x)).//(3)
    ! x : (?<2 y : edgeOnPath(y,x)) &
          (?<2 y : edgeOnPath(x,y)). // (4)
    ! x y : edgeOnPath(x,y) => reaches(from,y). // (5)
}
structure sp_struct : sp_voc {
    node = {A..D} // shorthand for A,B,C,D
    edge = {A,B; B,C; C,D; A,D} // ';' separated list of tuples
    from = A
    to = D
}
term lengthOfPath : sp_voc { #{ x y : edgeOnPath(x,y) } }
procedure main() {
    /* Search a minimal model */
    sol = minimize(sp_theory1, sp_struct, lengthOfPath)[1]
    /* If no result is returned, no models exist */
    if (sol == nil)
    then print("No models exist.\n")
    else print(sol)
    end
}
```

The IDP3 system performs model expansion and model minimization by first reducing the problem to extended CNF, using the grounder GIDL (Wittocx *et al.* 2010) and subsequently calling the solver MINISAT(ID) (Mariën *et al.* 2008). The grounding process can be fine-tuned using options for symmetry breaking (Devriendt *et al.* 2012), grounding with bounds (Wittocx *et al.* 2010), lazy grounding (De Cat *et al.* 2012) etc. The solver is an extension of MiniSat (Eén and Sörensson 2003) with support for aggregate expressions, inductive definitions and branch-and-bound optimization. Recently, MINISAT(ID) was extended to offer support for finite domain constraints, using the propagation techniques described in Schulte and Stuckey (2008), or, alternatively, interfacing with the GECODE Constraint Programming engine.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

2.4 Exploring the space of models for the shortest path problem

The above is a correct IDP3 model that provides a declarative solution to the problem at hand. However, if performance matters, or the instances are that large that the grounding cannot fit in memory, then other models can be preferable. In the long run, perhaps an optimizer can transform a simple model in a model for which model generation has a better performance, but, with the current state of affairs, it is up to the user to explore the design space and to look for better performing models. We do so in this section for the shortest path problem.

Listing 2. Another definition for `reaches`.

```
theory sp_theory2 : sp_voc {
  { reaches(x,y) <- edgeOnPath(x,y).
    reaches(x,y) <- edgeOnPath(x,z) & reaches(z,y). }
  /* ... constraints as in sp_theory1 ... */
}
```

Prolog programmers would never define the `reaches/2` predicate as in theory `sp_theory1` of Listing 1 but rather as in theory `sp_theory2` of Listing 2. Indeed, apart from the risk of entering an infinite loop, `reaches(x,z)` can have many more solutions than `edgeOnPath(x,z)` (which is bounded by the number of `edges`), and hence the search space for Prolog's proof procedure can be substantially larger.[5] While model generation cannot loop in the IDP system, the search space argument remains valid. Comparing the runtime of both systems (see Figure 1) reveals that the heuristics of the underlying SAT solver do not compensate for the larger search space and that the version of Listing 2 is substantially faster.[6] However, there is another factor contributing for the difference between both versions. Figure 1 splits runtime in grounding time (the vertical bar) and solving time (above the bar). For larger problems, one can observe that the grounding also takes more time. The explanation is the difference in grounding size (see Figure 2). The size difference must be attributed to the grounding of the recursive `reaches(x,z)` rule. With n nodes, there are n^3 instances of the recursive `reaches/2` rule in `sp_theory1`. As for `sp_theory2`, it follows from constraint (1) that `edgeOnPath(x,z)` is false whenever `edges(x,z)` is false, hence with e the number of edges, the number of instances is limited to $e * n$, which is typically substantially smaller than n^3.

Although the performance has improved quite a bit, we see that it is still rapidly increasing with the size of the graph. Moreover (see Figure 1), most of the runtime is the grounding time. Can we do better? The recursive `reaches/2` rule has three variables and remains expensive. The grounding has n^2 atoms `reaches(n1,n2)`, expressing whether $n1$ and $n2$ are connected while we are only interested in paths going from `from` to `to`. Hence, we should better use a unary `reachable` predicate and define the points that are reachable from `from`. The new versions of the `vocabulary`

[5] When using a system with tabling, the order in the body of the recursive clause is better inverted (Swift and Warren 2012).

[6] Using an Intel^R Core^{TM} i5-3550 CPU at 3.30 GHz with 7.8-GB RAM running Ubuntu with the IDP3 default options.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

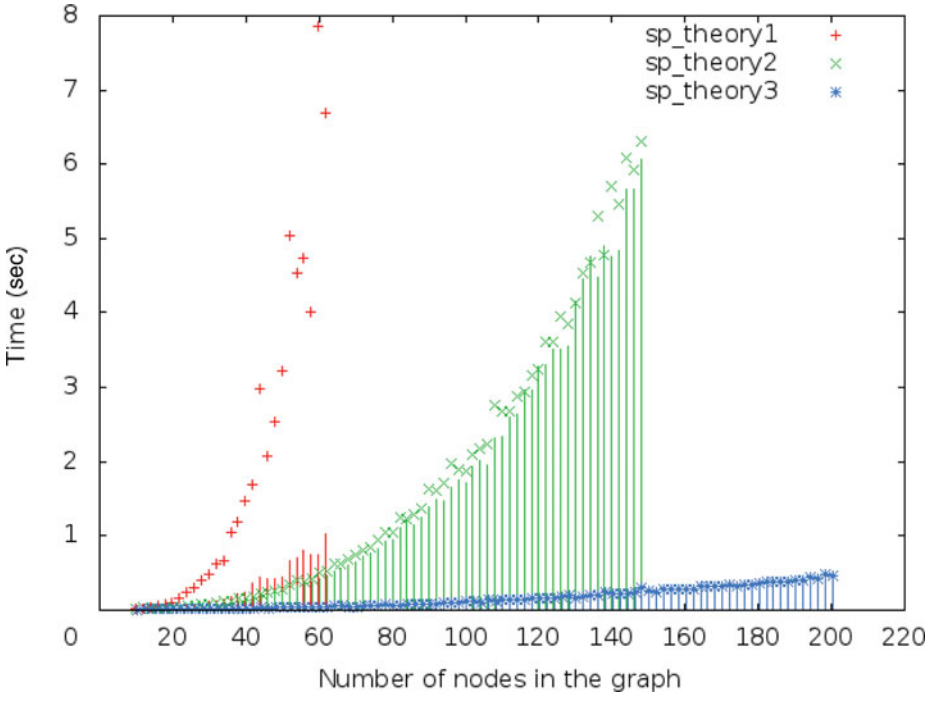

Fig. 1. (Colour online) Different grounding and total running times for sp_theory1, sp_theory2 and sp_theory3. Experiments are performed on graphs with an increasing number of nodes but a constant edge density. The vertical bar shows the grounding time, the part above the vertical bar is the solving time.

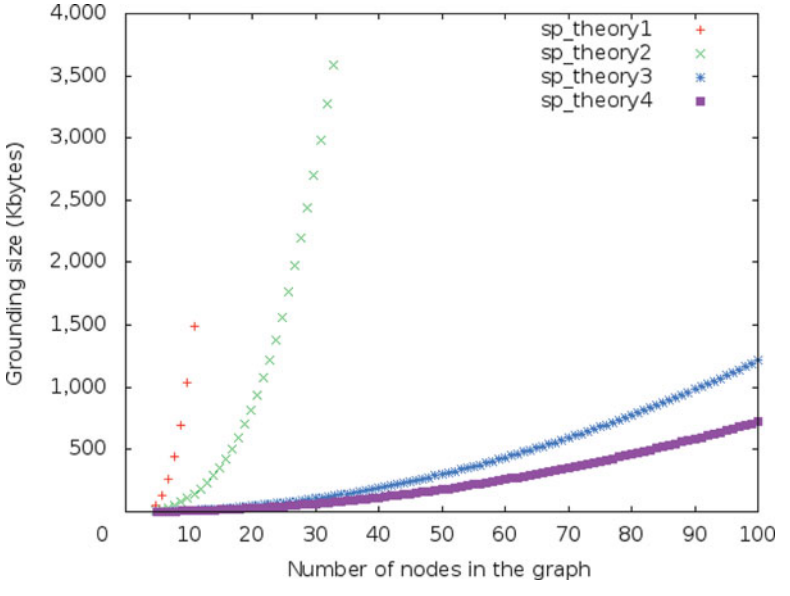

Fig. 2. (Colour online) Grounding size for sp_theory1, sp_theory2, sp_theory3 and sp_theory4. Experiments are performed on graphs with an increasing number of nodes but a constant edge density.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

and the `theory` are shown in Listing 3. The `term` and `procedure` parts are as in Listing 1.

Listing 3. A unary `reachable` relation instead of the binary `reaches relation`.

```
vocabulary sp_voc2 {
    type node
    from, to: node
    edge(node, node)
    edgeOnPath(node, node)
    reachable(node)
}
theory sp_theory3: sp_voc2 {
    { reachable(from).
      reachable(y) <- edgeOnPath(x,y) & reachable(x). }

    ! x[node] y[node] : edgeOnPath(x,y) => edge(x,y). //(1)
    reachable(to). //(2)
    ~(? x : edgeOnPath(x,from)) & ~(? x : edgeOnPath(to,x)).//(3)
    ! x : (?<2 y : edgeOnPath(y,x)) &
          (?<2 y : edgeOnPath(x,y)). //(4)
    ! x y : edgeOnPath(x,y) => reachable(y). //(5)
}
```

As Figure 1 shows, this modification results in a dramatic speed-up. Also, the grounding size (Figure 2) is substantially reduced.

Constraints (3) and (4) are cardinality constraints on the number of edges connected to the same node that can participate in a path. They are redundant with respect to the minimization of the `lengthOfPath` term. Indeed, paths of minimal length satisfy both constraints. Dropping them, as in Listing 4, spoils the clarity of the model; however, it further reduces the size of the grounding as can be seen in Figure 2. The effect on the runtime is negligible for small graphs and rather negative for larger ones as one can observe in Figure 3. The explanation is that the removal of these constraints increases the search space. Indeed, partial solutions violating constraints (3) and (4) are not immediately rejected. This causes a lot more variance in the solving times.

Listing 4. Removing redundant constraints.

```
theory sp_theory4: sp_voc {
    { reachable(from).
      reachable(y) <- reachable(x) & edgeOnPath(x,y). }

    ! x[node] y[node] : edgeOnPath(x,y) => edge(x,y). // (1)
    reachable(to).
}
```

The above example, in which we solved a classical problem with IDP3, illustrates the basic features of FO(·)$^{\text{IDP3}}$. It shows that problem-solving with IDP3 is a quite different endeavor from problem-solving in other languages. This holds not only for procedural languages but also for a declarative language such as Prolog. It also

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

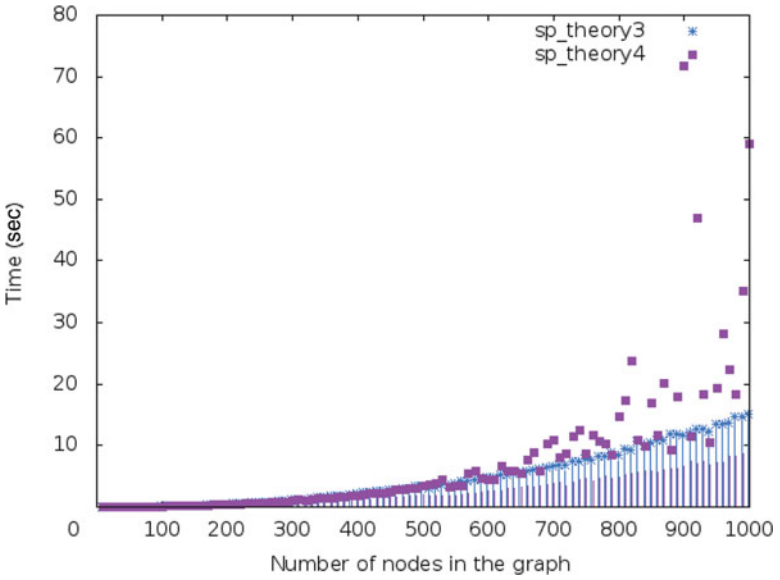

Fig. 3. (Colour online) Different grounding and total running times for sp_theory3 and sp_theory4. Experiments are performed on graphs with an increasing number of nodes but a constant edge density.

shows the importance of exploring various models when performance and memory use matters.

3 Stemmatology

Before the invention of the printing press, texts were copied manually by scribes. This copying process was not perfect; scribes often modified texts, either accidentally or intentionally. As a result the surviving copies of many old texts vary significantly. No text written before the invention of the printing press, and even up to the end of the 18th century, when the habit of circulating texts in manuscript form practically disappeared, can be read without a preliminary critical analysis of its material witnesses. This is the purpose of stemmatology. The Oxford English Dictionary defines the field as "the branch of study concerned with analyzing the relationship of surviving variant versions of a text to each other, especially so as to reconstruct a lost original."

A stemma is a kind of "family tree" of a *tradition*, a set of related manuscripts. It indicates how manuscripts ("children") have been copied from other manuscripts ("parents"), and the manuscript that is the original source. It may include both extant (currently existing and available) and non-extant ("lost") manuscripts. The stemma is not necessarily a tree: sometimes a manuscript has been copied partially from one manuscript, and partially from another, in which case the manuscript has multiple parents.

More formally, a stemma can be defined as a CRDAG, a Connected Directed Acyclic Graph with a single Root (Andrews and Macé 2013). A dataset contains

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

the manuscripts from one tradition. Each manuscript is described by a fixed set of features $F_1, \ldots, F_n$, each of which has a nominal domain $Dom(F_i)$ (variant readings of feature F_i). Typically, a feature refers to a particular location or a section in a text, although it can also be the spelling of a particular word, e.g., the dwelling of "Van den Vos Reynaerde" can be spelled as Malpertuis, Malpertus or Malpertuus.

The 19th century philologist Karl Lachmann was among the first to apply a principled method for reconstructing stemmata from sets of manuscripts (Timpanaro 2005). Nowadays, a variety of methods exist. Many are borrowed from biology, where a similar problem, reconstruction of phylogenetic trees, is well studied. However, these methods do not always fit the stemmatological context well. First, they assume that phylogenies are tree-shaped, while stemmata are DAGs.[7] Second, these trees contain only bifurcations, while stemmata can have multifurcations. Third, in most methods, the trees are such that each extant copy is at a leaf of the tree, whereas in stemmatology one extant copy may be an ancestor of another (and hence should be an internal node). Fourth, stemmatologists often have additional information, for instance, about the time or place of origin of a manuscript, which ideally should be taken into account. Research continues to develop new algorithms better suited for the stemmatological context (Baret *et al.* 2006).

3.1 The task

Apart from reconstructing stemmata from data, stemmatologists are also interested in other types of analyses, which may, for instance, use a known stemma or a manually constructed best-guess stemma as an input. These types of analysis can be very diverse. The data mining tasks that we address in this section belong to this category.

The problem studied here assumes that a CRDAG representing a stemma of a tradition is given, as well as feature data about the manuscripts from the tradition. More specifically, the data include a feature for each location where variation is observed in the tradition represented by the stemma. For each extant manuscript in the tradition, the feature data describe its variant reading; the variant reading is unknown for the non-extant ones. For most features, it seems rather unlikely that the same variant reading originated multiple times independently; i.e., it is reasonable to assume that there is one ancestor where the variant reading occurred for the first time (the "source" of the variant). Therefore, we say that *the feature is consistent with the stemma* if it is possible to indicate for each variant a single manuscript that may have been the origin of that variant. Since for some manuscripts the value of the feature is not known, checking consistency boils down to assigning a variant to each node in the CRDAG in such a way that, for each variant, the nodes having that variant form a CRDAG themselves. Note that one can imagine exceptions to the above, e.g., a new spelling of a word can be independently introduced in different copies.

[7] Some methods return phylogenetic networks, but these represent uncertainty about the real tree, which is different from claiming that the network represents the actual phylogeny.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

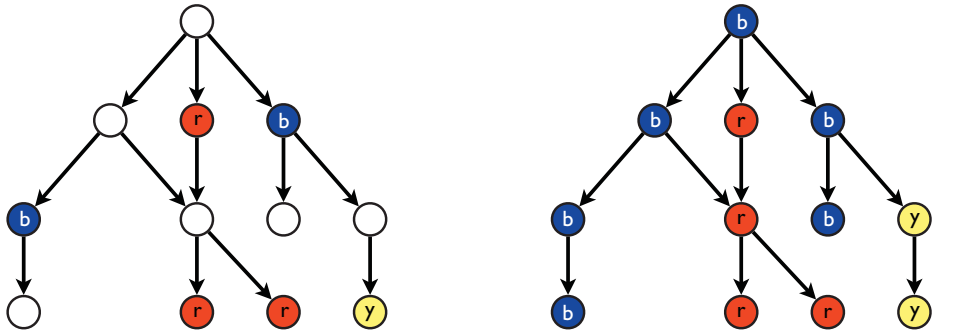

Fig. 4. (Colour online) Left: A partial labeling showing for a given feature which manuscripts have which variant readings/colors. Right: A complete extension of that labeling where each variant reading is a CRDAG. Because such an extension exists, the feature is consistent with the stemma.

3.2 Consistency checking of a stemma is NP-complete

We learned about this problem through contacts with researchers in stemmatology. One of them had developed an algorithm (implemented with a program of about 370 lines of Perl using a graph library as a back end) to solve the basic task, did several iterations to handle yet uncovered cases and was still worried about the completeness of their approach (does the algorithm always find a solution when a solution exists?). The algorithm attempts not to make wrong decisions by initially assigning several variant readings to the non-extant manuscripts and, in the second phase, remove variant readings while preserving consistency. Once understood, the problem was formalized as a graph problem and shown to be NP-complete by one of the authors of this paper. In this formalization, the variant reading of a text is represented as a color, and checking a stemma is a color-connected problem.

Definition 1 (*Color-connected*)
Two nodes x and y in a colored CRDAG are *color-connected* if a node z exists (z can be one of x and y) such that there is a directed path from z to x, and one from z to y, and all nodes on these paths (including z, x, y) have the same color.

Given a partially colored CRDAG, the *color-connected problem* is to complete the coloring such that every pair of nodes of the same color is color-connected.

An illustration is given in Figure 4. A candidate coloring can be checked in polynomial time, hence proving that the color-connected problem is NP-hard implies it is NP-complete.

Theorem 1
The color-connected problem is NP-hard.

Proof
The proof is by showing a polynomial reduction from SAT to color-connectedness.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

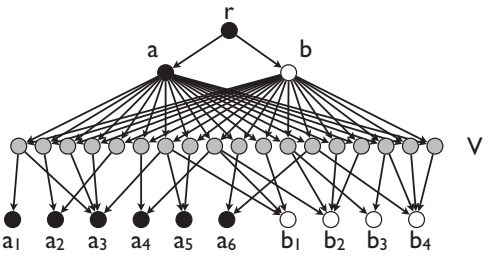

Fig. 5. A partially colored CRDAG constructed from a CNF theory T. The black a_i nodes represent the positive clauses, the white b_i nodes the negative clauses. The grey nodes represent the propositional variables; they are linked to the clauses in which they participate and have to be colored.

There exists a polynomial reduction from a conjunctive normal form (CNF) formula to one with all clauses either positive (all literals are positive) or negative (all literals are negative). Indeed, replace all occurrences of a negative literals $\neg x$ by a new positive literals *notx* and add for every such *notx* literal the clauses $x \vee notx$ and $\neg x \vee \neg notx$.

So we assume without loss of generality a CNF formula T comprising positive clauses T^+ and negative clauses T^-. We construct a color-connected problem whose solutions correspond to the models of T. Let $T^+ = C_1^+ \wedge C_2^+ \wedge \ldots \wedge C_m^+$ and $T^- = C_1^- \wedge C_2^- \wedge \ldots \wedge C_n^-$ with $C_1^+, \ldots, C_m^+$ positive clauses and $C_1^-, \ldots, C_n^+$ negative clauses. Let V be the set of propositional variables in T.

Now we construct a DAG G comprising the nodes $V(G) = r \cup A \cup B \cup V$ where $A = \{a, a_1, a_2, \ldots, a_m\}$ (a_i stands for clause C_i^+; a is an extra node) and $B = \{b, b_1, b_2, \ldots, b_n\}$ (b_i stands for clause C_i^-; b is an extra node). The directed edges are given by $E(G) = \{(v, a_i) | i \in [1..m], v \in C_i^+\} \cup \{(v, b_i) | i \in [1..n], v \in C_j^-\} \cup \{(a, v) | v \in V\} \cup \{(b, v) | v \in V \cup \{(r, a), (r, b)\}$. Next we color r, a and all nodes a_i black, and b and all nodes b_i white. We obtain a partially colored CRDAG (see Figure 5 for an example). Moreover, a solution to the color-connected problem encodes a solution to the original SAT problem. Indeed, each a_i node, representing a positive clause, is connected with at least one black variable. Hence, making all black variables true satisfies all positive clauses. Also, each b_i node, representing a negative clause, is connected with at least one white variable. Hence, making the white variables false satisfies all negative clauses. It follows that the color-connected problem is NP-hard. □

The problem being NP-complete, it is unlikely it can be solved by a procedural program without search. The proof suggests the problem becomes hard when nodes can have multiple parents. This situation is not dealt with (and therefore usually abstracted away) in traditional stemmatological methods. However, it does occur in the datasets we analyzed. Constructing small examples where several nodes have multiple parents, we quickly obtained an example for which the procedural code erroneously claimed no connected coloring exists. So the worries of the developer about the completeness of the code were grounded.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

3.3 An FO(·)$^{\text{IDP3}}$ *Solution*

A first FO(·)$^{\text{IDP3}}$ solution used a binary relation `SameVariant` for representing that two manuscripts have the same variant reading and imposed two constraints: (i) transitivity of `SameVariant` relation, and (ii) manuscripts with the same variant reading have a common ancestor with that variant reading and are connected to that ancestor through manuscripts with that same variant reading. This resulted in a working version that could serve as a golden standard for the procedural code, but was much slower than the latter.

As we already noted in the shortest path problem, modeling transitive closures results in large grounding sizes and runtime. Hence, a major improvement can be expected when that can be avoided. Representing the variant reading as a function from manuscripts to variants allowed us to drop the transitivity constraint. The final improvement, resulting in the program below, came from learning more about the procedural code: It checks for connectedness by following a path to the original source manuscript of the variant reading and checks that there is a single such source for the variant reading. Expressing the latter as a single constraint resulted in a version that turned out to be faster than the incomplete procedural algorithm. The IDP3 model is shown in Listing 5 and explained below. We also show most of the procedural code so that the reader can see how a number of satisfiability-checking tasks can be embedded in a single process.

Listing 5. Checking the consistency between stemma and features.

```
procedure main() {
  process("besoin")
  process("parzival")
  process("florilegium")
  process("sermon158")
  process("heinrichi")
}

/* ——— Knowledge base ——————————————————————————— */
vocabulary V {
  type Manuscript
  type Variant
  CopiedBy(Manuscript,Manuscript)
  VariantReading(Manuscript): Variant
}
vocabulary Vtask {
  extern vocabulary V
  SourceOf(Variant): Manuscript
}
theory Ttask : Vtask {
  ! x : (x ~= SourceOf(VariantReading(x))) =>
     ? y : CopiedBy(y,x) & VariantReading(y) = VariantReading(x).
}

/* ——— Check consistency between feature and stemma ———— */
procedure check(feature) {
  setvocabulary(feature,Vtask)
```

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

```
  return sat(Ttask,feature)
}

/* ------ Procedures for processing ------------------------------ */
procedure process(tradition) {
  io.write("Processing ",tradition,".\n")
  local path = "data/"
  local stemmafilename = path..tradition..".dot"
  local featurefilename = path..tradition..".json"
  processFiles(stemmafilename,featurefilename)
}
procedure processFiles(stemmafilename,featurefilename) {
  local stemma,nbnodes,nbedges = readStemma(stemmafilename)
  io.write("Stemma has ",nbnodes," nodes and
                        ",nbedges," edges.\n")
  local nbp,nbs,time = processFeatures(stemma,featurefilename)
  io.write("Found ",nbp," positive out of ",nbs," groupings ")
  io.write("in ",time," sec.\n")
}
procedure readStemma(stemmafilename) {
  /* 19 lines of lua code */
}
procedure processFeatures(stemma,featurefilename) {
  /* 23 lines of lua code
    a loop iterating over the features,
    -- compute feature as stemma extended with
        the feature specific data
    -- call check(feature)
    -- process the results
    finally, return the overall results */
}
```

The logical model is described in the "Knowledge base" section of the code. The vocabulary has been split in two parts. The vocabulary `V` is used to represent the input data: the stemma and the feature. It introduces the types `Manuscript` and `Variant`, the binary relation `CopiedBy` representing the parent–child pairs in the given structure of the stemma and the function `VariantReading` representing the known data about variant readings of manuscripts. The vocabulary `Vtask` extends `V` with the task-specific vocabulary. Only one extra function is needed, namely `sourceOf` which maps a variant reading to the manuscript that is the source of that variant reading. The theory `Ttask` comprises a single constraint; it states that a manuscript that is not the source of its own variant reading must have a parent with the same variant reading.

The remainder is procedural code. The procedure `main` iterates over all traditions to be analyzed and calls the procedure `process` for each of those. The latter procedure uses concatenation to construct two filenames from the name of the tradition and passes these file names to the `processFiles` procedure; the ".dot" file contains the stemma data; the ".json" file the feature data. The `readStemma` procedure (code omitted) returns the input structure describing the stemma as well as the number of manuscripts (nodes) and parent–child pairs (edges). The

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

Table 1. *The five traditions used in this work*

Name	#Manu-scripts	#Parent–child pairs	#Features	#Variant readings	
				maximum	average
Notre Besoin	13	13	44	5	2.18
Parzival	21	20	122	6	2.59
Florilegium	22	21	547	5	2.19
Sermon 158	34	33	270	3	2.12
Heinrichi	48	51	1042	17	4.84

`processFeatures` procedure (code omitted) iterates over the features in the file. For each feature, it constructs a feature structure by extending the stemma structure with the feature-specific data. It then calls the `check` procedure. This procedure extends the feature structure with the symbols from the `Vtask` vocabulary (`setvocabulary(feature,Vtask)`) and then checks the color-connectedness of the feature (`sat(Ttask,feature)`). The yes/no result is returned to the `processFiles` procedure which collects and returns the global results: number of consistent (positive) features, total number of features and time. The `processFiles` procedure prints these global data and returns to `main`.

As can be seen in the `main()` procedure, we used the code to perform consistency checking for the features of five traditions; two of them, Sermon 158 and Florilegium are real traditions, with stemmata that have been constructed according to current philological best practice; the other three are artificial traditions, produced under test conditions by volunteers for the purposes of empirical research into stemmatological methods. We received the data from Tara Andrews (at the time of writing employed at the KULeuven). A website where such stemma data can be found is `http://byzantini.st/stemmaweb/`. Some information about the stemma we used is given in Table 1.

The IDP program determines consistency for all features and datasets in a matter of seconds[8]:

```
> main()
Processing besoin.
Stemma has 13 nodes and 13 edges.
Found 26 positive out of 44 groupings in 0 sec.
Processing parzival.
Stemma has 21 nodes and 20 edges.
Found 45 positive out of 122 groupings in 1 sec.
Processing florilegium.
Stemma has 22 nodes and 21 edges.
Found 431 positive out of 547 groupings in 2 sec.
Processing sermon158.
Stemma has 34 nodes and 33 edges.
Found 64 positive out of 270 groupings in 2 sec.
Processing heinrichi.
Stemma has 48 nodes and 51 edges.
Found 1 positive out of 1,042 groupings in 12 sec.
>
```

[8] Using an Intel^{R} Core^{TM} 2 Duo CPU at 3.00 GHz with 3.7 GB of RAM running Ubuntu with the IDP3 options stdoptions.groundwithbounds = false (disabling bounded grounding) and stdoptions.liftedunitpropagation = false (disabling lifted unit propagation).

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

Our largest benchmark is the heinrichi data set (Roos and Heikkilä 2009). This stemma about old Finnish texts includes 48 manuscripts, 51 `copiedBy` tuples and information about 1,042 features. Processing all features takes 12 sec with the IDP system, while it took 25 sec with the original procedural code.

One can observe that rather few features are consistent with the stemma. This raises the question, what is the minimal number of sources needed to explain the data. To solve that inference task, it suffices to replace the vocabulary extension `Vtask` and the theory `Ttask` in the knowledge base and to introduce the term to be minimized. As core procedure, `Check` is replaced by `minSources` and the processing of results has to be adjusted. The most relevant new parts are shown in Listing 6. The `IsSource` predicate is defined as manuscripts that do not have a parent with the same variant reading.

Listing 6. Minimize the number of sources.

```
/* ——— new parts of Knowledge base ——————————————— */
vocabulary Vms {
  extern vocabulary V
  IsSource(Manuscript)
}
theory Tms : Vms {
  {! x : IsSource(x) <- ~? y : CopiedBy(y,x) &
                        VariantReading(y) = VariantReading(x).}
}
term NbOfSources : Vms {
  #{ x : IsSource(x) }
}

/* ——— the core procedure ——————————————————————— */
procedure minSources(feature) {
  setvocabulary(feature,Vms)
  return minimize(Tms,feature,NbOfSources)[1]
}
```

Although this is a minimization problem, processing the traditions is still a matter of seconds, except for the larger Heinrichi dataset, which now requires about 5 min to process its 1,042 features.

Other variations are of interest to the researchers. One variation, mentioned by Andrews *et al.* (2012), considers the possibility that the scribe has copied from an older ancestor than the direct parent, thus reintroducing a variant. Playing with the relative penalty of introducing a new variant versus reverting to an older variant, one can obtain various explanations of interest to the stemmatologist. All these can be achieved with modifying a handful of lines in the model. Interesting about the above variant is that it uses a predicate `IndirectAncestor` that is defined in terms of the stemma data, so it can be computed once and reused when processing each of the features. As illustrated in Listing 7, the tight integration of the knowledge

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

base with the procedural code makes this very easy.[9] The procedure `readStemma`, which constructs the stemma structure from the inputfile, is extended with the call `modelexpand(T,stemma)[1]`. The resulting model is the stemma structure extended with the true `IndirectAncestor` atoms. This structure, together with the other outputs of `readStemma`, is returned to the procedure `processFiles` which uses it to handle the features one by one.

Listing 7. Materializing a definition once and using the materialization many times.

```
vocabulary V {
  /* ... as in Listing 5 ... */
  IndirectAncestor(Manuscript,Manuscript)
}

theory T : V {
  {! x y : IndirectAncestor(x,y) <-
           ? z : CopiedBy(x,z) & IndirectAncestor(z,y).
   ! x y : IndirectAncestor(x,y) <-
           ? z : CopiedBy(x,z) & CopiedBy(z,y).}
}

procedure readStemma(stemmafilename) {
  local stemma = newstructure(V,"stemma")
  /* ... reading the stemma data ... */
  return modelexpand(T,stemma)[1], #nodes, #edges
}
```

4 Minimum common supergraphs of partially labeled trees

Phylogenetic trees, extensively surveyed by Felsenstein (2004), are the traditional tool for representing the evolution of a given set of species. However, there exist situations in which a tree representation is inadequate. One reason is the presence of evolutionary events that cannot be displayed by a tree: genes may be duplicated, transferred or lost, and recombination events (i.e., the breaking of a DNA strand followed by its reinsertion into a different DNA molecule) as well as hybridization events (i.e., the combination of genetic material from several species) are known to occur. The second reason is that even when evolution is indeed tree-like, there are cases in which a relatively large number of tree topologies are "equally good" according to the chosen criterion, and that not enough information is available to discriminate between those trees. One solution that has been proposed to address the latter issue is the use of *consensus trees*, where the idea is to find a tree that represents a compromise between the given topologies. Another approach, the focus of this section, consists in building a network that is compatible with all topologies of interest. A somewhat loose description of the variant we are interested in, which will be stated in a more formal way below, is to find the smallest graph that contains

[9] With a more recent version of IDP3, the user can leave this optimization to the system (Jansen *et al.* 2013).

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

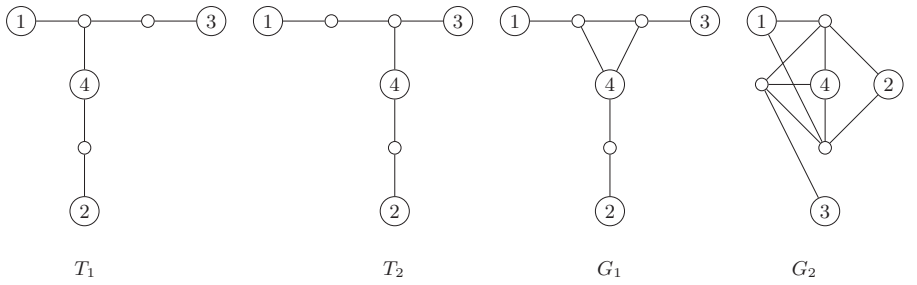

Fig. 6. Two 7-graphs, T_1 and T_2, and two of their common supergraphs. G_1 is a minimum common supergraph.

a given set of evolutionary trees. For more information about *phylogenetic networks*, see the recent book by Huson *et al.* (2010) and the online, up-to-date annotated bibliography maintained by Gambette (2010).

4.1 The problem

The studied problem is about the evolution of a fixed set of m species. The input is a set of phylogenetic trees, each tree showing a plausible relationship between the m species. All trees have n ($> m$) nodes, m of them are labeled with the name of the species (typically, in the leaves, but also internal nodes can be labeled). Given $n - m$ extra names, the labeling of each tree can be extended to a full labeling. Now we can consider the union of these full labelings: a network with m labeled nodes and edges which are induced by the bijections between the fully labeled trees and the network. Obviously, the number of edges of the network depends on the chosen full labelings of the trees. The task is to find a network with a minimum number of edges. Below, we formulate the problem as a slightly more general graph problem where we do not fix the size of the initial labeling.

Definition 2 (*Common supergraph of partially labeled n-graphs*)
Given is a set N of n names and a set of graphs $\{G_1, G_2, \ldots, G_t\}$ where each graph $G_i = (V, E_i, \mathscr{L}_i)$ has n vertices (the set V), edges connecting pairs of vertices (the set E_i) and where some of the vertices are labeled by names (an injective partial function $\mathscr{L}_i : V \to N$). A graph (N, EN) is a *common supergraph* of $\{G_1, G_2, \ldots, G_t\}$ if there exists, for each i, a bijection $\mathscr{L}'_i : V \to N$ that extends $\mathscr{L}_i$ and such that $\{v, w\} \in EN$ iff there exists an i such that $\{v', w'\} \in E_i$ and $\{v, w\} = \{(\mathscr{L}'_i(v'), \mathscr{L}'_i(w'))\}$.

A common supergraph (N, EN) is a *minimum* common supergraph if no other common supergraph (N, EN') exists for which $|EN'| < |EN|$.

Note that every labeling function $\mathscr{L}'_i$ induces an injection $E_i \to EN$, hence the name common supergraph. Figure 6 shows two partially labeled 7-graphs, along with two of their common supergraphs. G_2 is not a minimum common supergraph since it has more edges than G_1; G_1 is a minimum common supergraph since T_1 and T_2 are not isomorphic and G_1 has only one more edge than each of T_1 and T_2.

Now we can consider the following decision problem: Given a set of partially labeled n-graphs, can the labelings be completed such that the n-graphs have a

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

common supergraph with at most k edges? Labarre and Verwer (to appear) prove that this problem is NP-hard, even if the n-graphs are trees with all leaves labeled.

4.2 An FO(·)$^{\text{IDP3}}$ *solution*

Listing 8 shows a simple model inspired by Labarre and Verwer (to appear). It makes use of three types, `tree`, `vertex` and `name`. The latter two types have the same number of elements in a correct input structure. The structure of the given trees is described by the ternary predicate `edge` (the first argument refers to the tree to which the edge belongs), the structure of the common supergraph (over the names themselves) by the predicate `arc`. The labeling is described by the function `label` from the nodes of the given trees to the names. It is partially given in the input structure and is completed during model expansion. The constraint in the theory, stating that, for each name `nm` and each tree `t`, there exists *exactly one* node `nd` (denoted `? 1 nd`) such that its label is `nm`, ensures that the labeling is bijective. The `arc` atoms can be defined as the pairs of names induced by the labels on the nodes of an edge of the tree (the definition in the theory). However, as the minimization is on the number of `arc` atoms in a model, some care is required. One should ensure that either `arc` is a symmetric relation or there is at most one `arc` atom for each pair of names. The latter approach is taken as it gives a somewhat smaller grounding. It is achieved by exploiting the total order that exists over each domain (the tests `label(t,x) < label(t,y)`).

Listing 8. Modeling CS-PLT in FO(·)$^{\text{IDP3}}$.

```
vocabulary CsPltVoc {
  type tree
  type vertex
  type name // Isomorphic to vertex
  edge(tree,node,node)  // trees, given in input structure
  arc(name,name) // the induced network
  label(tree,node): name // the labeling,
                    // partially given in the input structure
}
theory CsPltTheory : CsPltVoc {
  { // induced network; arc is anti-symmetric
   ! t x y : arc(label(t,x),label(t,y)) <- edge(t,x,y) &
                                           label(t,x) < label(t,y).
   ! t x y : arc(label(t,x),label(t,y)) <- edge(t,y,x) &
                                           label(t,x) < label(t,y).
  }
  ! t nm : ?1 nd : label(t,nd) = nm. // label is bijective
}
term SizeOfSupergraph : CsPltVoc { #{ x y : arc(x,y) } }
procedure main() {
  print(minimize(CsPltTheory,CsPltStructure,SizeOfSupergraph)[1])
}
```

In this solution, each rule of the `arc` definition has two occurrences of the terms `label(t,x)` and `label(t,y)`. The current grounder naively associates a distinct symbol

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

with each occurrence, which boils down to grounding a clause of the following form:

Listing 9. One of the `arc` rules after initial processing by the grounder.

```
! t x y lx1 lx2 ly1 ly2 : arc(lx1,ly1) <- lx1=label(t,x) &
      ly1=label(t,y) & lx2=label(t,x) & ly2=label(t,y) &
      edge(t,x,y) & lx2 < ly2.
```

This approach creates extra variables and very large groundings. To avoid this behavior, on can rewrite the definition as follows:

Listing 10. Better performing definition of `arc`.

```
{ // induced network
 ! t x y lx ly : arc(lx,ly) <- lx=label(t,x) & ly=label(t,y) &
                               edge(t,x,y) & lx < ly.
 ! t x y lx ly : arc(lx,ly) <- lx=label(t,x) & ly=label(t,y) &
                               edge(t,y,x) & lx < ly.
}
```

While the formulation is less elegant, the effect on the size of the grounding and the solving time is dramatic; e.g., the grounding is reduced from 620,798 to 6,024 propositional clauses, and the solving time from 144 sec to 8 sec on a problem with five trees of eight vertices and four initial labels.

One can explore several other variations. As mentioned above, one could use a symmetric `arc` relation. Also, as the `arc` definition is free of recursion, one could replace it with the two implications of the completion. Then, exploiting the minimization on the number of `arc` atoms, one could drop the only-if part of the completion. The effect on solving time of all these variations is rather marginal.

4.3 An approximate solution

The solving time is exponential in the number of nodes and, if several trees are involved, the program becomes impractical on real-world problems, even if the best solution found so far is returned when some time budget is exceeded. However, the versatility of the IDP system allowed us to experiment with various strategies for greedily searching an approximate solution. This led to the following quite natural solution that performed very well with respect to both running time and quality of the solution.

1. Find a minimum common supergraph (MCS) for every pair of trees.
2. Pick an MCS with minimum size (say G) and remove the two trees that are the input for G.
3. Find an MCS between G and every remaining tree.
4. Replace G[10] by an MCS with minimum size, remove the tree that is the input for this MCS and go back to step 3 if any tree remains.

[10] This way, the MCS is assembled by each time incorporating one additional original tree.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

Table 2. *Randomly generated instances of the minimum common subgraph problem solved with a time bound of 2,000 sec. Sizes of MCS (average over four runs) for exact and greedy approach*
*Approximate solution due to time out

#Trees	#Nodes	#Initial labels	Exact #edges	Greedy #edges
5	55	5	**130**	131.25
5	60	10	**128**	132.75
5	75	25	207.75*	**184.75**
10	55	5	183.75*	**154.50**
10	60	10	177.75*	**154.75**
10	75	25	270.00*	**269.25**
20	55	5	241.50*	**171.75**
20	60	10	232.00*	**152.25**
20	75	25	346.25*	**279.00**

Steps 1 and 3 of this simple procedure are performed by IDP3 using a model very similar to that of Listing 8 (see Labarre and Verwer (to appear) for the actual model).[11] This greedy approach works very well. Indeed, for large instances and a fixed time budget, the exact method runs out of time and returns a suboptimal solution, while the greedy method completes and returns a solution that, although suboptimal, is typically much smaller. Table 2 shows some experimental results on randomly generated data with various parameters. A timeout was set to 2,000 sec,[12] and the average number of edges were recorded over four runs for each instance for both exact and greedy methods.

5 Learning deterministic finite state automata

The third task is about learning a DFA. The goal is to find a (non-unique) smallest DFA that is consistent with a given set of positive and negative examples. It is one of the best studied problems in grammatical inference (de la Higuera 2005), has many application areas and is known to be NP-complete (Gold 1978). Interestingly, one of the first algorithms proposed to solve this problem was based on a translation to constraint programming (Biermann and Feldman 1972). Much later, translations of this problem to graph coloring (Coste and Nicolas 1997; Costa Florêncio and Verwer 2012) and satisfiability (Grinchtein *et al.* 2006; Heule and Verwer 2010) were proposed. Although the DFA learning problem is typically tackled using greedy approaches (de la Higuera 2005), Heule and Verwer (2012) recently won the 2010 Stamina DFA learning competition (Stamina 2010) by an improved translation to a SAT problem and running an off-the-shelf SAT solver. Here we explore to what extent an FO(·)$^{\texttt{IDP3}}$ formalization can compete with this competition winner.

[11] The whole method can be implemented as an IDP3 procedure; however, the scripts had been implemented before the Lua interface was available.
[12] Using an Intel^{R} Core^{TM} i7 CPU 870 at 2.93 GHz with 8 GB of RAM running Ubuntu; default settings for IDP3.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

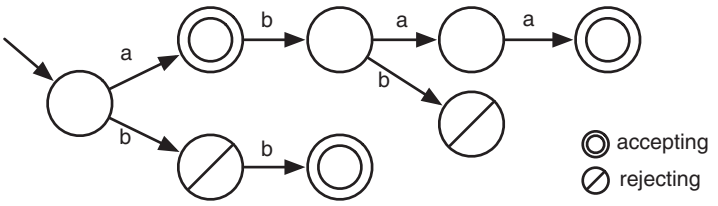

Fig. 7. An augmented prefix tree acceptor (APTA) for $S = (S^+ = \{a, abaa, bb\}, S^- = \{abb, b\})$. The start state (annotated with incoming arrow) is the root of the APTA.

5.1 The problem

A DFA is a directed graph comprising a set of *states* Q (nodes) and labeled *transitions* T (directed edges). The root is the start state and any state is either an *accepting* or a *rejecting* state. In each state, there is exactly one transition for each symbol. A DFA defines a language, the set of strings it accepts. It can be used to *generate* or *verify* sequences of symbols (strings) using a process called *DFA computation*. When verifying strings, the symbols of the input string determine a path through the graph. When the final state is an accepting state, the string is accepted, otherwise it is rejected.

Given a pair of finite sets of positive example strings S^+ and negative example strings S^- (the *input sample*), the goal of *DFA identification* (or *learning*) is to find a (non-unique) *smallest* DFA A that is *consistent* with $S = \{S^+, S^-\}$, i.e., every string in S^+ is accepted, and every string in S^- is rejected by A. Typically, the size of a DFA is measured by $|Q|$, the number of states it contains.

Most DFA learning algorithms are based on the method of state-merging. This method first constructs a tree-shaped automaton called the *augmented prefix tree acceptor* (APTA). As can be seen in Figure 7, the APTA accepts the positive examples and rejects the negative ones. Other strings either end up in a non-final state or cannot be processed due to a missing transition. The APTA automaton can be *completed* to obtain a DFA with the same number of states by (arbitrarily) labeling the non-final states and adding the missing transitions. When all non-final states are labeled as reject and all extra transitions target a reject state with no path to an accepting state, this DFA accepts only the positive examples.

A smaller DFA, accepting more strings, can be constructed by state-merging on the APTA. Merging states under the constraints that the automaton remains deterministic (at most one transition/label in each state) and accepting and rejecting states cannot be merged preserves consistency with the input sample. State-merging increases the number of strings accepted by the automaton, and hence generalizes the language accepted by the DFA that completes the automaton.

States of the final automaton are thus equivalence classes of states of the APTA. Calling the states of the final automaton colors, the problem becomes that of finding a coloring of the states of the APTA that is consistent with the input sample. Following Coste and Nicolas (1997), Heule and Verwer (2010) take this approach. They formulate constraints expressing which pairs of states are incompatible, and abstract the problem as a graph. The nodes of this graph are the states of the

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

APTA and the edges are the incompatible pairs. The decision problem, whether there exists an automaton with k states, becomes a graph coloring problem for k colors. They use a clever SAT encoding to solve this decision problem and embed it in a workflow to solve the minimization problem. For really large problems, the SAT formulation becomes too big (hundreds of colors, resulting in over 100 million clauses) to be handled by a SAT solver (Heule and Verwer 2010). To reduce the problem size, they used a greedy heuristic procedural method based on state-merging. Every merge performed by this method reduces the size of the APTA and therefore also the size of the encoding. In addition, this preprocessing identifies a clique of pairwise incompatible states in the APTA. For states in this clique, the colors can be fixed in advance. The effect is to break the symmetries between these colors and thus to further reduce the size of the problem. The preprocessing also deduces that certain state/color combinations cannot result in a solution. A preprocessed problem instance is then extended with a set of SAT clauses and is the input for the SAT solver. The SAT clauses express the constraints of the problem and are generated from the instance.

5.2 *An* FO(·)$^{\text{IDP3}}$ *solution*

Our goal is not to set up the complete workflow described above, but to compare the performance of the native SAT encoding of Heule and Verwer (2010) and Heule and Verwer (2012) with the performance of an FO(·)$^{\text{IDP3}}$ model on the same problem instances as obtained after the preprocessing. Our FO(·)$^{\text{IDP3}}$ model for solving a single instance is shown in Listing 11.

Listing 11. Modeling DFA in FO(·)$^{\text{IDP3}}$.

```
vocabulary dfaVoc {
  type state // states used in APTA
  type label // symbols triggering transitions
  type color // available states for resulting automaton
  partial trans(state,label): state // transitions of APTA
  acc(state) // accepting states of APTA
  rej(state) // rejecting states of APTA
  colorOf(state): color // fixed in input for colors in clique
  // the resulting automaton:
  partial colorTrans(color,label): color // transitions of DFA
  accColor(color) // accepting states
}
theory dfaTheory : dfaVoc {
  ! x : acc(x) => accColor(colorOf(x)).
  ! x : rej(x) => ~accColor(colorOf(x)).
  // trans induces colorTrans:
  ! x l z: trans(x,l)=z =>
       ! i j : colorOf(x)=i & colorOf(z)=j => colorTrans(i,l)=j.
}
procedure main() {
  print(modelexpand(dfaTheory,instance)[1])
}
```

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

The types `state`, `label`, the function `trans` and the predicates `acc` and `rej` describe the given input samples (and hence the APTA). Note that `trans` is partial as it is only defined for the transitions present in the input sample. The states of the resulting automaton are the elements of the type `color`. Its transitions are described by the function `colorTrans`. This function is also declared as a partial function. To obtain a complete DFA, the function `colorTrans` has to be extended with the missing transitions. Which one is assigned does not matter, as it does not affect the processing of the strings in the input sample (although it has an effect on the language that is accepted). The function `colorOf` maps the states of the APTA on the states (colors) of the DFA. The predicate `accColor` describes the accepting states of the resulting automaton.

The theory expresses two constraints on `accColor`: accepting states of the APTA must and rejecting states cannot be mapped to an accepting state of the DFA. The third constraint states that each transition in the APTA induces a transition (between colors) in the DFA.

The input structure, which is omitted, not only completely defines the types `color`, `state` and `label` but also the APTA. That implies that the IDP3 grounder has complete knowledge about the relations `accept` and `reject` and the function `trans`. Hence, for example, the grounder only grounds the formula colorTrans(colorOf(x),l)=colorOf(z) for tuples `(x,l,z)` for which trans(x,l)=z is true in the input structure. Further, the input structure also contains the partial information about `colorOf` and `colorTrans` that has been derived by preprocessing.

The `main` procedure assumes that the input structure is named `instance`; it calls the solver to search for a model, and prints it.

The above model is a very natural formulation of the problem and corresponds quite closely to a "decompilation" of the SAT clauses expressing the constraints of the problem (Table 1 in both Heule and Verwer (2010) and Heule and Verwer (2012)). The most noticeable difference is in a redundant constraint, which can be decompiled into:

Listing 12. Redundant constraint

```
!v l w: trans(w,l)=v =>
            !j: colorTrans(colorOf(w),l)=j => colorOf(v)=j.
```

This formula can be derived from the last formula in our theory together with the fact that `colorOf` is a total function.

In FO(·)$^{\text{IDP3}}$, it is very straightforward to extend the model with a term counting the number of colors used and to minimize that number. This makes our FO(·)$^{\text{IDP3}}$ method very similar to the optimization method used by Heule and Verwer (2010, 2012). IDP, however, has several advantages over an encoding constructed by hand. For instance, variants of the model such as minimizing the number of transitions in the DFA instead of the number of states are very straightforward to obtain, but would require a major re-engineering of the SAT encoding. This has practical value as different application domains of DFA prefer different optimization criteria. Furthermore, it is much easier to introduce bugs in handmade encodings than in the few lines of IDP code. In fact, by analyzing and comparing the results of IDP

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

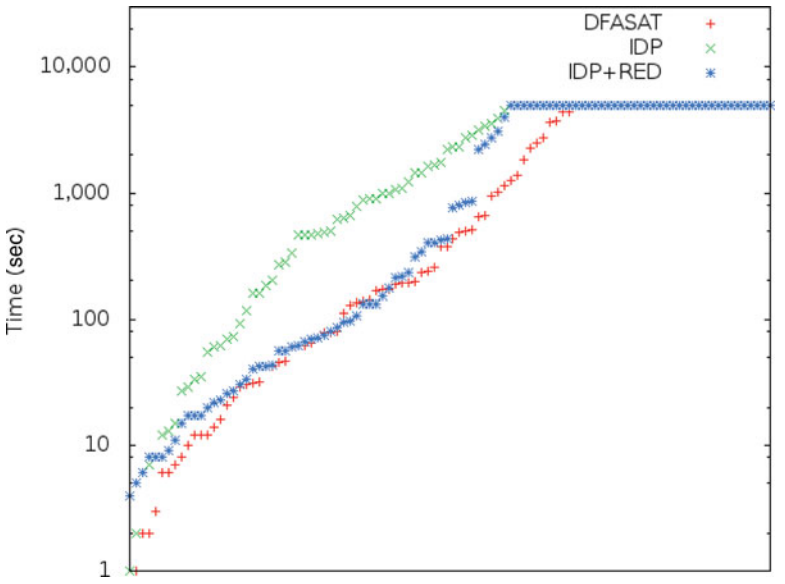

Fig. 8. (Colour online) Solving time for the SAT encoding (DFASAT), the FO(·)$^{\text{IDP3}}$ model of Listing 11 (IDP) and the model extended with the redundant constraint of Listing 12 (IDP+RED). Time in each system is monotonically increasing, so the order of problem instances is different for each system. Timeout is set at 5,000 sec. Sixty-nine problems are solved by DFASAT, 59 by IDP and IDP+RED.

and the handmade encoding, we discovered some subtle bugs in the handmade translation that caused an incorrect answer in rare occasions.

5.3 Experiments

We compared the performance of our model with that of the SAT encoding, denoted DFASAT, for 100 tough problems (the DFA is restricted to have only five states on top of those in the initial clique) from the 2010 Stamina DFA learning competition (Stamina 2010).[13]

Figure 8 compares the solving time of DFASAT with that of two FO(·)$^{\text{IDP3}}$ models. The first one is as shown in Listing 11 (IDP); the second one extends the model with the redundant constraint of Listing 12 (IDP+RED). One can observe that the redundant constraint improves the performance of the IDP3 system and that the performance comes quite close to that of DFASAT. Still, DFASAT can solve more problems than IDP+RED (69 versus 59). We also have to add that the dedicated preprocessing that generates the SAT instances requires on average 5 sec, while the grounding takes substantially more time. For the IDP version, it is on average 124 sec; for ID+RED, the average is 168 sec. The results reported here are substantially better than those reported in Blockeel *et al.* (2012). By analyzing

[13] Using an Intel^{R} Core^{TM} i5-2500 CPU at 3.30 GHz with 7.7 GB of RAM Running Ubuntu. Memory use was limited to 4 GB, time to 5000 sec. IDP3 was ran with standard options.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

these earlier results, we unraveled that the large performance gap was due to our grounding being three times the size of the SAT encoding. This was caused by the introduction of unneeded auxiliary predicates (so-called Tseitins) during the grounding. The problem was repaired in a new version of the grounder. As mentioned before, our detailed analysis also revealed subtle bugs in the dedicated preprocessing which generates the SAT encodings for DFASAT. Thus, using an IDP implementation of a hard problem such as DFA learning and comparing it with a fast competition winning SAT translation was not only useful for improving IDP but also for improving the competition winner.

It is very encouraging to observe that the performance of a tiny *and* comprehensible predicate logic model comes very close to that of an ingeniously tuned SAT encoding that is a key component of a competition winner.

6 Conclusions

In this paper, we presented the IDP3 system from a user's perspective. We introduced the various components of an $\mathrm{FO}(\cdot)^{\mathrm{IDP3}}$ model and illustrated their use in a model for the shortest path problem. We also showed models for some problems encountered by researchers in data mining and machine learning. In the first problem from stemmatology, $\mathrm{FO}(\cdot)^{\mathrm{IDP3}}$ models proved to be of invaluable help for researchers trying to cope with stemma that go beyond tree structures (Andrews *et al.* 2012). We obtained a model that not only correctly handles arbitrary directed acyclic graphs but also achieved better performance than the original (incomplete) procedural code. In the second problem, about phylogenetic trees, $\mathrm{FO}(\cdot)^{\mathrm{IDP3}}$ models helped researchers to explore approximate solutions for an NP-hard problem (Labarre and Verwer to appear). The third problem that we modeled is the classical problem of learning a DFA. We compared an $\mathrm{FO}(\cdot)^{\mathrm{IDP3}}$ model with the state-of-the-art SAT encoding of the problem. Here we found that the performance of an IDP3 solution comes pretty close to that of a highly tuned SAT encoding. These applications illustrate that $\mathrm{FO}(\cdot)^{\mathrm{IDP3}}$ models are a valuable alternative for dedicated procedural code when novel data needs to be analyzed and explored. Interestingly, in both problems, where we compared with an existing solution (stemmatology and DFA learning), we uncovered some bugs in those solutions. It is fair to add that we also uncovered some cases where the grounder of the IDP3 system performed a suboptimal job. In the minimum common supergraph application we found that it deals poorly with multiple occurrences of the same term in a formula; in the DFA application we found that it introduced unneeded auxiliary symbols. While the latter problem has already been solved, the former is, at the time of writing, still on the to-do list of the implementation team.

Our work is a further indication that the IDP3 system is coming of age. It was already known from the ASP-competitions that it compares pretty well with ASP systems in terms of performance (Denecker *et al.* 2009; Calimeri *et al.* 2011). In contrast to ASP, which relies on the stable semantics (Gelfond and Lifschitz 1988), it is based on the first-order logic. The informal semantics of FO's connectives and the novel language constructs is clear and easy to understand. This probably makes

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

it easier for newcomers to start modeling. For example, the authors of the minimum common supergraph problem (Labarre and Verwer to appear) were neither familiar with Prolog nor with FO(·)$^{\text{IDP3}}$ and hardly needed any help from the IDP team. The core of an FO(·)$^{\text{IDP3}}$ model consists on the one hand of formulas in the first-order logic, which act as constraints, and on the other hand of definitions, which are close to the rules of traditional logic programs. Given interpretations for open predicates (the predicates that are not defined in the theory), the definitions determine a unique model through the well-founded semantics (Van Gelder *et al.* 1991). The search results in an interpretation of the open predicates and hence a model of the theory that is consistent with the constraints. What distinguishes FO(·) from traditional logic programming is the use of non-Herbrand interpretations, and correspondingly the lack of constructor functions. This often leads to a simpler data representation and gives rise to elegant model formulations. On the other hand, there are cases where rich data structures that arise in the Herbrand interpretations (compound terms, lists, trees, ...) are useful too and these currently cannot easily be modeled in IDP3. Another distinction is that the IDP framework offers other forms of inference, most notably model expansion and model minimization. A feature of the IDP3 system is the integration of procedures in FO(·)$^{\text{IDP3}}$ models (De Pooter *et al.* 2011) and the clean separation between declarative and procedural components. As we illustrated in the stemmatology application, this allows a user to develop a whole workflow in an FO(·)$^{\text{IDP3}}$ model.

The logic of FO(·)$^{\text{IDP3}}$ extends predicate logic with inductive definitions, types, arithmetic, aggregates and partial functions. Of these, inductive definition is the most fundamental one. The basis of the language is predicate logic. In fact, in many applications, the extensions only serve for making models more readable. For example, the aggregates (in the form of quantifications $? < 2$) in the shortest path problem are directly translatable to FO and the (non-recursive) definition in the minimum common supergraph problem is equivalent with its completion. Hence, three out of the four problems that we describe in this text are in fact solved with pure predicate logic models.

Our work on applications taught us also a few things about good models. In all problems that we solved in this paper, a class of objects is separated in equivalence classes. (In the shortest path problem there is the class of edges participating in the path, and the class of other edges.) It is tempting to represent these equivalence classes by the transitive closure of some relation. However, the transitive closure of a binary relation is expensive. It gives rise to large groundings and this, together with the cost of checking for unfounded sets, results in poor performance. Binary transitive closures arise naturally during modeling, but they are better avoided in the IDP3 system. In the shortest path problem our first solution had a binary transitive `reaches` relation. It required some creative tinkering and awareness that binary transitive closures are harmful to make the switch to the solution we presented. Replacing it with the unary `reachable` relation had a major impact on efficiency. Also, our first solution to the stemmatology problem had a transitive closure. Here transitive closure could be avoided altogether. It was a major step forward in efficiency to replace it by a coloring function for the nodes in the stemma graph.

https://doi.org/10.1017/S147106841400009X Published online by Cambridge University Press

The other two problems also use functions (`label` and `colorOf` respectively) whose range defines membership in an equivalence class.

The preference of a unary transitive relation over a binary one is an illustration of another general principle: less variables is better in rules and constraints. One should try to break up complex rules and constraints in simpler ones requiring less variables and explore whether one can do with predicates and functions having less arguments. Another important point is that one should not be satisfied with the first correct model. Often, major improvements are possible, as we illustrated in several of our applications.

The IDP3 system is an evolving research system and further improvements are on the way. A lot of ongoing work aims at making the performance less dependent on clever modeling. One recent feature is symmetry breaking. Predicate-level symmetry detection and dynamic symmetry breaking (during search) automatically exploit symmetries present in the problem (Devriendt *et al.* 2012) (symmetry is present in the DFA problem: permuting the colors gives another solution; however it was broken in an *ad hoc* way in the SAT encoding and hence also in the input structure of our instances). One recent feature is to avoid complete proposionalization during grounding, on one hand by keeping function terms in the grounding (De Cat *et al.* 2013), and, on the other hand through lazy, demand-driven grounding during search (De Cat *et al.* 2012, 2014). Another feature is the detection of functional dependencies and their use to reduce the arity of predicates (De Cat and Bruynooghe 2013).

Acknowledgements

Caroline Macé and Tara Andrews introduced some of the authors to stemmatology and provided the data sets; Tara also explained the working of the procedural code. This work was supported by Research Foundation – Flanders (FWO-Vlaanderen) and the Research Council of KU Leuven (GOA/08/008 and GOA 13/010).

References

Andrews, T., Blockeel, H., Bogaerts, B., Bruynooghe, M., Denecker, M., De Pooter, S., Macé, C. and Ramon, J. 2012. Analyzing manuscript traditions using constraint-based data mining. In *COmbining COnstraint solving with MIning and LEarning (CoCoMile)*. Montpellier, France, 27 August 2012, Proceedings First Workshop on Combining Constraint Solving with Mining and Learning (CoCoMile). (ECAI 2012 Workshop), 15–20.

Andrews, T. and Macé, C. 2013. Beyond the tree of texts: Building an empirical model of scribal variation through graph analysis of texts and stemmata. *Literary and Linguistic Computing 28,* 4, 504–521.

Baret, P., Macé, C., Robinson, P., Peersman, C., Mazza, R., Noret, J., Wattel, E., Van Mulken, M., Robinson, P., Lantin, A-C., Canettieri, P., Loreto, V., Windram, H., Spencer, M., Howe, C., Albu, M. and Dress, A. 2006. Testing methods on an artificially created textual tradition. In *The Evolution of Texts: Confronting Stemmatological and Genetical Methods*, *Proceedings of the International Workshop*, Louvain-la-Neuve. Istituti editoriali e poligrafici internazionali, Pisa, Italy, 255–283.

Biermann, A. W. and Feldman, J. A. 1972, June. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers 21,* 6, 592–597.

Blockeel, H., Bogaerts, B., Bruynooghe, M., De Cat, B., De Pooter, S., Denecker, M., Labarre, A., Ramon, J. and Verwer, S. 2012. Modeling machine learning and data mining problems with FO(·). In *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, Budapest, Hungary, September 4–8, 2012, A. Dovier and V. S. Costa, Eds., LIPIcs, vol. 17. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Wadern, Germany, 14–25.

Brewka, G., Eiter, T. and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M. C., Sirianni, M., Terracina, G. and Veltri, P. 2011. The third answer set programming system competition: Preliminary report of the system competition track. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. Springer, Berlin, Germany, 388–403.

Costa Florêncio, C. and Verwer, S. 2012. Regular inference as vertex coloring. In *Algorithmic Learning Theory*, N. Bshouty, G. Stoltz, N. Vayatis and T. Zeugmann, Eds., Lecture Notes in Computer Science, vol. 7568. Springer, Berlin, Germany, 81–95.

Coste, F. and Nicolas, J. 1997. Regular inference as a graph coloring problem. In *ICML Workshop on Grammatical Inference, Automata Induction, and Language Acquisition.* Workshop on Automata Induction, Grammatical Inference, and Language Acquisition at The Fourteenth International Conference on Machine Learning (ICML-97) July 12, 1997, Nashville, Tennessee, 6 pages.

De Cat, B., Bogaerts, B., Bruynooghe, M. and Denecker, M. 2014. Predicate logic as a modelling language: The IDP system. CoRR abs/1401.6312.

De Cat, B., Bogaerts, B., Denecker, M. and Devriendt, J. 2013. Model expansion in the presence of function symbols using constraint programming. In *25th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2013)*, Washinton, WA, November 4–6, 2013. 1068–1075.

De Cat, B. and Bruynooghe, M. 2013. Detection and exploitation of functional dependencies for model generation. *Theory and Practice of Logic Programming 13,* 4–5, 471–485.

De Cat, B., Denecker, M. and Stuckey, P. 2012. Lazy model expansion by incremental grounding. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, Budapest, Hungary, September 4–8, 2012, A. Dovier and V. S. Costa, Eds., LIPIcs, vol. 17. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Wadern, Germany, 201–211.

De Cat, B., Denecker, M., Stuckey, P. J. and Bruynooghe, M. 2014. Lazy model expansion: Interleaving grounding with search. *CoRR abs/1402.6889*.

de la Higuera, C. 2005. A bibliographical study of grammatical inference. *Pattern Recognition 38,* 9, 1332–1348.

Denecker, M., Lierler, Y., Truszczynski, M. and Vennekens, J. 2012. A Tarskian informal semantics for Answer Set Programming. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, Budapest, Hungary, September 4–8, 2012, A. Dovier and V. S. Costa, Eds. LIPIcs, vol. 17. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Wadern, Germany, 277–289.

Denecker, M. and Ternovska, E. 2008, April. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL) 9,* 2, 14:1–14:52.

Denecker, M. and Vennekens, J. To appear. The well-founded semantics is the principle of inductive definition, revisited. In *14th International Conference on Principles of Knowledge Representation and Reasoning*, Vienna, Austria, July 20–24, 2014, AAAI press.

Denecker, M., Vennekens, J., Bond, S., Gebser, M. and Truszczyński, M. 2009. The second Answer Set Programming competition. In *10th International Conference on Logic*

Programming and Non-Monotonic Reasoning (LPNMR), E. Erdem, F. Lin and T. Schaub, Eds. LNCS, vol. 5753. Springer, Berlin, Germany, 637–654.

De Pooter, S., Wittocx, J. and Denecker, M. 2011. A prototype of a knowledge-based programming environment. In *International Conference on Applications of Declarative Programming and Knowledge Management*. Lecture Notes in Computer Science, vol. 7773. Springer, Berlin, Germany, 279–286.

Devriendt, J., Bogaerts, B., Mears, C., Cat, B. D. and Denecker, M. 2012. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *Proceedings of the 24th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'12)*, Athens, Greece, IEEE Press, 49–56.

Dovier, A. and Costa, V. S., Eds. 2012. *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, Budapest, Hungary, September 4–8, 2012. LIPIcs, vol. 17. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Wadern, Germany.

Eén, N. and Sörensson, N. 2003. An extensible SAT-solver. In *International Conference, SAT*, E. Giunchiglia and A. Tacchella, Eds. LNCS, vol. 2919. Springer, Berlin, Germany, 502–518.

Erdem, E. 2011. Applications of answer set programming in phylogenetic systematics. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning – Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, Lecture Notes in Computer Science, vol. 6565, Springer, 415–431.

Felsenstein, J. 2004. *Inferring Phylogenies*. Sinauer, Sunderland, MA.

Frisch, A. M., Harvey, W., Jefferson, C., Hernández, B. M. and Miguel, I. 2008. Essence: A constraint language for specifying combinatorial problems. *Constraints 13,* 3, 268–306.

Gambette, P. 2010. Who is who in phylogenetic networks: Articles, authors and programs. Published electronically. Accessed 2011. URL: `http://www.atgc-montpellier.fr/phylnet`.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007. *clasp*: A conflict-driven answer set solver. In *LPNMR*, C. Baral, G. Brewka and J. S. Schlipf, Eds. LNCS, vol. 4483. Springer, Berlin, Germany, 260–265.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *ICLP/SLP*, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Cambridge, MA, 1070–1080.

Gold, E. M. 1978. Complexity of automaton identification from given data. *Information and Control 37,* 3, 302–320.

Grinchtein, O., Leucker, M. and Piterman, N. 2006. Inferring network invariants automatically. In *Automated Reasoning*, U. Furbach and N. Shankar, Eds. Lecture Notes in Computer Science, vol. 4130. Springer, Berlin, Germany, 483–497.

Guns, T., Nijssen, S. and Raedt, L. D. 2011. Itemset mining: A constraint programming perspective. *Artificial Intelligence 175,* 12–13, 1951–1983.

Heule, M. and Verwer, S. 2010. Exact DFA identification using SAT solvers. In *Grammatical Inference: Theoretical Results and Applications (ICGI 2010)*, 66–79.

Heule, M. J. H. and Verwer, S. 2012. Software model synthesis using satisfiability solvers. *Empirical Software Engineering*, August, 1–32. doi: http://dx.doi.org/10.1007/s10664-012-9222-z.

Huson, D. H., Rupp, R. and Scornavacca, C. 2010. *Phylogenetic Networks: Concepts, Algorithms and Applications*. Cambridge University Press, Cambridge, UK.

Ierusalimschy, R., de Figueiredo, L. H. and Celes, W. 1996. Lua – an extensible extension language. *Software: Practice and Experience 26,* 6, 635–652.

Jansen, J., Jorissen, A. and Janssens, G. 2013. Compiling input* FO(·) inductive definitions into tabled Prolog rules for IDP3. *Theory and Practice of Logic Programming 13,* 4–5, 691–704.

Kowalski, R. A. 1974. Predicate logic as programming language. In *IFIP Congress*, J. L. Rosenfeld, Ed. Information Processing 74, Proceedings of IFIP Congress 74, Stockholm, Sweden, August 5–10, 1974. North-Holland, 1974, 569–574.

Labarre, A. and Verwer, S. To appear. Merging partially labelled trees: Hardness and an efficient practical solution. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2002. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7*, 499–562.

Mariën, M., Wittocx, J., Denecker, M. and Bruynooghe, M. 2008. SAT(ID): Satisfiability of propositional logic extended with inductive definitions. In *SAT*, H. Kleine Büning and X. Zhao, Eds. LNCS, vol. 4996. Springer, Berlin, Germany, 211–224.

Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P. J., Garcia de la Banda, M. and Wallace, M. 2008. The design of the Zinc modelling language. *Constraints 13,* 3, 229–267.

Milner, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17,* 3, 348–375.

Mitchell, D. G. and Ternovska, E. 2005. A framework for representing and solving NP search problems. In *Twentieth AAAI National Conference on Artificial Intelligence (AAAI-05)*, M. M. Veloso and S. Kambhampati, Eds. MIT Press, Cambridge, MA, 430–435.

Nieuwenhuis, R., Oliveras, A. and Tinelli, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM 53,* 6, 937–977.

Pelov, N., Denecker, M. and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming (TPLP) 7,* 3, 301–353.

Roos, T. and Heikkilä, T. 2009. Evaluating methods for computer-assisted stemmatology using artificial benchmark data sets. *Literary and Linguistic Computing 24,* 4, 417–433.

Schulte, C. and Stuckey, P. J. 2008. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems 31,* 1.

Stamina 2010. The StaMinA competition, learning regular languages with large alphabets. Accessed 2012. URL: `http://stamina.chefbe.net/`.

Swift, T. and Warren, D. S. 2012. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming 12,* 1–2, 157–187.

Syrjänen, T. and Niemelä, I. 2001. The smodels system. In *LPNMR*, T. Eiter, W. Faber and M. Truszczyński, Eds. LNCS, vol. 2173. Springer, Berlin, Germany, 434–438.

Timpanaro, S. 2005. *The Genesis of Lachmann's Method*, G. W. Most, Trans. University of Chicago Press, Chicago, IL.

Van Gelder, A., Ross, K. A. and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM 38,* 3, 620–650.

Wittocx, J., Denecker, M. and Bruynooghe, M. 2013. Constraint propagation for first-order logic and inductive definitions. *ACM Transactions on Computational Logic 14,* 3.

Wittocx, J., Mariën, M. and Denecker, M. 2008. The IDP system: A model expansion system for an extension of classical logic. In *The 2nd International Workshop on Logic and Search (LaSh 2008)*, M. Denecker, Ed. November 6–7, 2008, Leuven, Belgium, 153–165.

Wittocx, J., Mariën, M. and Denecker, M. 2010. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research 38*, 223–269.