Modeling Machine Learning and Data Mining Problems with FO(·)*

Hendrik Blockeel, Bart Bogaerts, Maurice Bruynooghe, Broes De Cat, Stef De Pooter, Marc Denecker, Anthony Labarre, Jan Ramon,[1] and Sicco Verwer[2]

1 **Department of Computer Science, KU Leuven**
`firstname.secondname@cs.kuleuven.be`
2 **Radboud Universiteit Nijmegen, Institute for Computing and Information Sciences**
`siccoverwer@gmail.com`

Abstract

This paper reports on the use of the FO(·) language and the IDP framework for modeling and solving some machine learning and data mining tasks. The core component of a model in the IDP framework is an FO(·) theory consisting of formulas in first order logic and definitions; the latter are basically logic programs where clause bodies can have arbitrary first order formulas. Hence, it is a small step for a well-versed computer scientist to start modeling. We describe some models resulting from the collaboration between IDP experts and domain experts solving machine learning and data mining tasks. A first task is in the domain of stemmatology, a domain of philology concerned with the relationship between surviving variant versions of text. A second task is about a somewhat similar problem within biology where phylogenetic trees are used to represent the evolution of species. A third and final task is about learning a minimal automaton consistent with a given set of strings. For each task, we introduce the problem, present the IDP code and report on some experiments.

1998 ACM Subject Classification D.1.6 [Logic Programming], F.4.1 [Mathematical Logic]: Computational logic, I.2.4 [Knowledge Representation Formalisms and Methods]

Keywords and phrases Knowledge representation and reasoning, declarative modeling, logic programming, knowledge base systems, FO(·), IDP framework, stemmatology, phylogenetic tree, deterministic finite state automaton.

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.14

1 Introduction

Researchers in machine learning and data mining are often confronted with problems for which no standard algorithms are applicable. Here we explore a few of these problems. They can be abstracted as graph problems and are NP-complete. This means that algorithms inherently involve search and that heuristics are needed to guide the search towards solutions. Doing this in a procedural language is complex and cumbersome; this is the kind of application for which high level modeling languages can be very useful. Under such a paradigm, a model specifies the format of the data, the function to be optimized and a set of constraints to be satisfied. The model together with a given problem instance is handed over to a solver which

* This work was supported by BOF project GOA/08/008 and by FWO Vlaanderen.

produces a solution. Several modeling languages exist in the field of Constraint Programming; the Zinc language [16] is a good example. The Answer Set Programming (ASP) paradigm can also be considered a modeling language; many solvers exist, examples are the systems described in [15, 9, 21]. Another such modeling language is FO(·) [6]. Problem solving with such modeling languages makes use of powerful solvers that perform propagation to squeeze the search space each time a choice is made. They relieve the programmer from encoding such propagation in procedural code. By default, these solvers use heuristics which are not problem-specific, but even so they often outperform procedural solutions.

This paper explores the use of the FO(·) language and its incarnation in the IDP framework for solving some machine learning and data mining problems. The core component of a model in the IDP framework is an FO(·) theory consisting of formulas in first order logic and definitions; the latter are basically logic programming clauses with arbitrary first order formulas in the body. The necessary background on FO(·) is given in Section 2.

Section 3 solves a task in the domain of a stemmatology, a part of philology that studies the relationship between surviving variant versions of a text. Section 4 discusses a problem about phylogenetic trees as used in biology. Whereas the first two tasks are new, in Section 5, it is investigated how well FO(·) performs on a standard machine learning task, namely the learning of a minimal deterministic finite state automaton (DFA) that is consistent with a given set of accepted and rejected strings.

In all of these problems, model expansion [17] —expanding a partially given structure into a complete structure that is a model of a theory— is the core computational task. Sometimes, a model that is minimal according to some criterion is required.

2 FO(·) and the IDP framework

2.1 FO(·)

The term FO(·) is used to denote a family of extensions of first order logic (FO). In this text, the focus lies on FO(·)$^{\text{IDP}}$, the instances supported by the IDP framework. FO(·)$^{\text{IDP}}$ extends FO with (among others) *types*, *arithmetic*, *aggregates*, *partial functions* and *inductive definitions*. This section recalls the aspects of FO(·) that are necessary for a good understanding of the rest of the paper; more information can be found in [23] and [3].

A specification in FO(·)$^{\text{IDP}}$ consists of a number of logical components, namely *vocabularies, structures, terms*, and *theories*. A vocabulary declares the symbols to be used (contrary to Prolog, the first character of a symbol has no bearing on its kind); a structure is a database with input knowledge; a term declared as a separate component represents a value to be optimized; a theory consists of FO formulas and inductive definitions. An *inductive definition* is a set of *rules* of the form $\forall \bar{x} : P(\bar{x}) \leftarrow \varphi(\bar{x})$. where φ is an FO(·)$^{\text{IDP}}$ formula. As argued in [6], the intended meaning of all common forms of definitions is captured by the well-founded semantics [22] which extends the least model semantics of Prolog's definite clauses to rule sets with negation. An FO(·)$^{\text{IDP}}$ *formula* differs from FO formulas in several ways. Firstly, FO(·)$^{\text{IDP}}$ is a many-sorted logic: every variable has an associated *type* and every type an associated domain. Moreover, it is order-sorted: types can be subtypes of others. Secondly, besides the standard terms of FO, FO(·)$^{\text{IDP}}$ also has aggregate terms: functions over a set of domain elements and associated numeric values which map to the sum, product, cardinality, maximum or minimum value of the set.

We write $\mathcal{M} \models T$ to denote that structure $\mathcal{M}$ satisfies theory T. With $x^{\mathcal{M}}$, we denote the interpretation of x under $\mathcal{M}$, where x can be a formula or a term.

2.2 The IDP framework

The IDP framework [5] combines a declarative specification, in FO(·)$^{\text{IDP}}$, with imperative manipulation of the specification via the Lua [13] scripting language. Such an interaction makes it a *Knowledge Base System* (KBS), as it allows one to reuse the same declarative knowledge for a range of inference tasks such as *model expansion*, *optimization*, *verification*, *symmetry breaking*, *grounding*, etc. For an in-depth treatment of the framework and the supported inferences, we refer to [3].

In this paper, we focus on the inference tasks *model expansion* and *model minimization*. 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 an interpretation $\mathcal{M}$ 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 the model. Such a task is represented as $\langle V, T, S\rangle$.

The task of model minimization, represented as $\langle V, T, S, t\rangle$ with V, T and S as above and t a term, is to find a model $\mathcal{M}$ of T that expands S such that for all other models $\mathcal{M}$' expanding S, $t^{\mathcal{M}} \leq t^{\mathcal{M}'}$.

The IDP framework allows users to specify FO(·)$^{\text{IDP}}$ problem descriptions. Such a problem description consists of logical and procedural components. The basic overall structure of the various logical components is as in the following schema.

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

The first component defines a vocabulary V. The other components define respectively a theory T, a term t and a structure S. They all refer to the vocabulary V for the symbols they use. In general, several vocabularies can be defined, eventually, one vocabulary extending another.

We use IDP syntax in the examples throughout the paper. Each IDP operator has an associated logical operator, the main (non-obvious) operators being: `&`($\wedge$), `|`($\vee$), $\sim$($\neg$), `!`($\forall$), `?`($\exists$), `<=>`($\equiv$), $\sim$`=`($\neq$).

The procedural component consists of procedures, coded in Lua, that provide the interface between the user and the logical components. Examples will be shown in the next sections.

3 Stemmatology

3.1 The task

The Oxford English Dictionary defines stemmatics, or stemmatology, 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 set of manuscripts that indicates which manuscripts have been copied from which other manuscripts, and which manuscript 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. Hence, a stemma is in general a connected directed acyclic graph with a single root [1]; we use the term CRDAG (connected rooted DAG) for it.

While constructing a stemma has some similarities with constructing a phylogenetic tree in biology, the algorithms of that domain do not fit the stemmatological context well and specific algorithms are developed [2].

The problem studied here assumes that a CRDAG representing a stemma is given, as well as feature data about (some of) the manuscripts. More specifically, for each location where variation is observed in the manuscripts, the data includes a feature that indicates which variant a particular manuscript has. Note that, in practice, it is highly unlikely that exactly the same variant originated multiple times independently; when a variant occurs in multiple manuscripts, it is reasonable to assume there was one ancestor, common to all of these, where the variant occurred for the first time (the "source" of the variant)[1]. 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. Using colors to denote the value of a variant, this property is captured by the following definition.

▶ **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.

3.2 An IDP solution

A pair of researchers in stemmatology attempted to develop a search free algorithm. They wrote 370 lines of perl and used a graph library in the background. While it worked for their benchmarks, they were worried about the completeness of their approach. After abstracting the problem as the color-connected problem, we proved that it was NP-complete (hence requires search) and constructed a solvable example for which their algorithm claimed no solution exists. We also worked on an IDP solution. After several iterations, we arrived at the following simple solution which turned out to be faster than the (incomplete) procedural algorithm on the benchmark set. It is shown in Listing 1.

The vocabulary part introduces two types (`manuscript` and `color`), two functions and one predicate. The function `colorOf` maps a manuscripts to its color and the function `sourceOf` maps a color to the manuscript that is the source of the feature. The predicate `copiedBy` is used to represent the CRDAG of the stemma in the input structure.

The theory part compactly represents the color-connectedness property by a single constraint: when the source of the color of a manuscript (`x`) is not equal to the manuscript itself then there must exist a manuscript (`y`) with the same color that has been copied by `x`.

The Lua code of the procedure `process` (omitted, 60 lines) processes the stemma data and builds the input structure for `copiedBy`. It then iterates over the features, partially builds the structure for the function `colorOf` and calls the procedure `check`, passing all structures in the variable `feature`. The latter procedure calls the model expansion and returns the result to `process` which reports them to the user.

Our largest benchmark so far is the Heinrichi data set [18]. This stemma about old Finnish texts includes 48 manuscripts, 51 `copiedBy` tuples and information about 1042 features. Processing all features takes 12 seconds with the IDP system while it took 25 seconds with the original procedural code. Our solution is integrated in the toolset of [20].

[1] For some features, e.g., the spelling of a particular word, this does not hold.

Listing 1 Description of the connected-coloring problem using IDP.

```
vocabulary V {
  type manuscript
  type color
  copiedBy(manuscript,manuscript)
  colorOf(manuscript): color
  sourceOf(color): manuscript
}
theory T : V {
  ! x : x ~= sourceOf(colorOf(x))
     => ? y : copiedBy(y,x) & colorOf(x) = colorOf(y).
}
procedure check(feature) {
  return sat(T,feature) // checks existence of a model
}
procedure process(stemmafilename,samplefilename) {
read the stemma data and build a structure for copiedBy
for each feature {
   read the given colors and build a partial structure for colorOf
   call check(feature)
   report the results }
}
```

4 Minimum common supergraphs of partially labelled trees

Phylogenetic trees, extensively surveyed by [7], 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 hybridisation events (i.e., the combination of genetic material from several species) are known to occur. A second reason is that even when evolution is indeed tree-like, there are cases in which a relatively large number of tree topologies might be "equally good" according to some chosen criteria, and 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, on which we focus here, 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 a given set of evolutionary trees. For more information about those *phylogenetic networks*, see the recent book by [12] and the online, up-to-date annotated bibliography maintained by [8].

4.1 The problem

The studied problem is about the evolution of a fixed set of m given species. The input is a set of phylogenetic trees, each tree showing a plausible relationship between the 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 into a full labeling. The completely labeled trees then induce pairs

of labels, whose union yields a graph over the set of n names. The task is to find a network with a minimum number of edges. Here, 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 S of n names and a set of graphs $\{G_1, G_2, \ldots, G_t\}$ where each graph $G_i = (V, E_i, \mathcal{L}_i)$ has n vertices and is partially labeled with an injective function $\mathcal{L}_i : V \to S$. A graph (S, ES) is a *common supergraph* of $\{G_1, G_2, \ldots, G_t\}$ if there exists, for each i, a bijection $\mathcal{L}'_i : V \to S$ that extends $\mathcal{L}_i$ and such that, for each edge $\{v, w\}$ of E_i: $\{\mathcal{L}'_i(v), \mathcal{L}'_i(w)\} \in ES$.

A *minimum* common supergraph (S, ES) is a common supergraph such that $|ES'| \geq |ES|$ for all common supergraphs (S, ES').

Note that every labeling function $\mathcal{L}'_i$ induces an injection $E_i \to ES$, hence the name common supergraph. Figure 1 shows two partially labeled 7-graphs, along with two of their common supergraphs. 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. G_2 is not a minimum common supergraph since it has more edges than G_1.

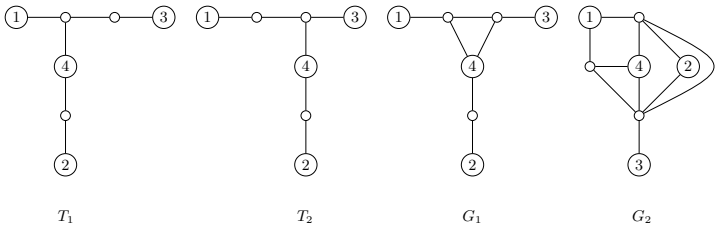

Figure 1 Two 7-graphs T_1 and T_2, a minimum common supergraph G_1, and a common supergraph G_2 that is not minimum.

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 common supergraph with at most k edges? It is proven in [14] that this problem is NP-hard, even if the n-graphs are trees with all leaves labeled.

4.2 The IDP solution

Listing 2 shows a simple model inspired by [14]. The labeling is declared as a function from nodes to the names (it is partly specified in the input structure). The only constraint of the theory forces the function to be bijective. The common supergraph over the names induced by the labeling is given by the arc atoms. As the minimization is on the number of such atoms, some care is required. Either one should make arc a symmetric relation or one should pay attention to the direction, e.g., by ensuring $x < y$ in arc(x,y) (every type is ordered in FO$(\cdot)^{\texttt{IDP}}$ and provided of a $<$ predicate). The latter is done here as the former gives a somewhat larger grounding.

A feature of the shown solution is that the terms label(t,x) and label(t,y) each have two occurrences in the rules defining arc. The current grounder associates a distinct variable with each occurrence. One can avoid this by replacing the head of the definition by arc(lx,ly) and by adding lx=label(t,x) and ly=label(t,y) to the body. This has a dramatic effect on the size of the grounding and on the solving time; e.g., the grounding is reduced from 620798 to 6024 lines and the solving time from 144s to 8 s on a problem with 5 trees of 8 vertices (4 leaves).

Listing 2 Modelling CS-PLT in the IDP format.

```
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(label(t,x),label(t,y)) <- edge(t,x,y) &
                                  label(t,x) < label(t,y).
    arc(label(t,x),label(t,y)) <- edge(t,y,x) &
                                  label(t,x) < label(t,y).
  }
  ! t c : ?1 n : label(t,n) = c. // label function is bijective
}
term SizeOfSupergraph: CsPltVoc { #{ x y : arc(x,y) } }
procedure main() {
  print(minimize(CsPltTheory,CsPltStructure,SizeOfSupergraph)[1])
}
```

The solving time is exponential in the number of nodes and 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 the smallest MCS (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 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.

Steps 1 and 3 of this simple procedure are performed by IDP using a model very similar to that of Listing 2 (see [14] for the actual model). 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.

5 Learning deterministic finite state automata

A third task is about learning a *deterministic finite state automaton* (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 [4], has many application areas, and is known to be NP-complete [10]. Recently [11] won the 2010 Stamina DFA learning competition [19] by reducing the DFA learning problem to a SAT problem and running an off-the-shelf SAT solver. Here we explore whether an FO(·)$^{\text{IDP}}$ formalization can compete with this competition winner.

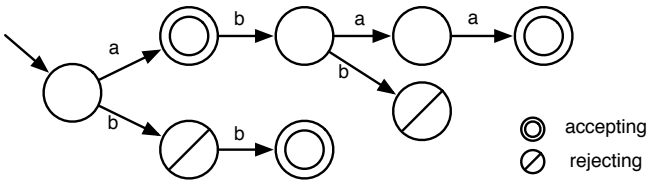

Figure 2 An augmented prefix tree acceptor (APTA) for $S = (S^+ = \{a, abaa, bb\}, S^- = \{abb, b\})$. The start state is the root of the APTA.

5.1 The problem

A *deterministic finite state automaton* (DFA) is a directed graph consisting of a set of *states* Q (nodes) and labeled *transitions* T (directed edges). The root is the start state and any state can be an *accepting state*. In each state, there is exactly one transition for each symbol. A DFA can be used to *generate* or *accept* sequences of symbols (strings) using a process called *DFA computation*. When accepting 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.

5.2 The solution

Most DFA learning algorithms use a form of state-merging. First, a a tree-shaped automaton called the *augmented prefix tree acceptor* (APTA), is constructed. As can be seen in Figure 2, the APTA accepts the positive examples and rejects the negative ones. State-merging merges states under the constraint that the automaton remains deterministic (at most one transition/label in each state) and that accepting and rejecting states cannot be merged.

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. This is also the approach taken by [11]; they formulate constraints expressing which pairs of states are incompatible, and abstract the problem as a graph, with as states the states of the APTA and as links the incompatible pairs. The problem is now a conventional graph coloring problem and they use a clever SAT encoding to solve it. Here we construct a direct model in FO$(\cdot)^{\text{IDP}}$. But before doing so, we have to consider one more aspect. For really large problems, the SAT formulation was too big (hundreds of colors, resulting in over 100.000.000 clauses) [11]. To get around such problems, they used a greedy heuristic procedural method to identify a clique of pairwise incompatible states in the APTA. For states in such a clique, the colors can be fixed in advance. The effect is to break some symmetries and to reduce the size of the problem. We assume here that the states of the clique are already colored in the input structure.

The FO$(\cdot)^{\text{IDP}}$ DFA learning theory is depicted in Listing 3. 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 the input samples do not define all transitions.

Listing 3 Modelling DFA in the IDP format.

```
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 defining 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
  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 => colorTrans(colorOf(x),l)=colorOf(z).
}
term nbColorsUsed: dfaVoc { #{ x : (? y : ColorOf(y) = x ) } }
procedure main() {
  stdoptions.symmetry = 1 //detect and break symmetries
  print(minimize(dfaTheory, simple, nbColorsUsed)[1])
}
```

The states of the resulting automaton are elements of the type `color`. Its transitions are described by the function `colorTrans`. This function is also declared as a partial function. To construct a complete DFA from the result, `colorTrans` has to be made total by mapping the missing transitions to a hidden "sink" state. The function `colorOf` maps the states of the APTA on the states (colors) of the final automaton. Finally, 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 final automaton. The third constraint states that each transition on the APTA induces a transition between colors. The term nbColorsUsed counts the number of states (colors) of the resulting automaton and is used for minimization. Instead of minimizing the number of states, one could as well minimize other properties such as the number of transitions, depth of the model, the size of loops, etc. They are also easy to formalize in $\text{FO}(\cdot)^{\text{IDP}}$. This makes the resulting DFA learning tool very suitable for application in different problem domains such as software engineering or bioinformatics where other optimization criteria are preferred.

In order to test the performance of the IDP translation, we ran it on the benchmark set of [11]. We compare IDP with two versions of the encoding in [11]: an unoptimized plain encoding (but with the symmetry breaking clique), and an optimized version (with extra symmetry breaking, unit literal propagation, but without redundant clauses). The experiment is not on the minimization problem but on the problem of constructing a DFA with a fixed set of states.

IDP, with the symmetry breaking option on, is significantly faster than the plain SAT encoding (not for the easy problems where the IDP time is dominated by the approximately one second grounding time, a time not needed when the problem is directly encoded in SAT). For example the maximum runtime of an instance in IDP is approximately 1400 seconds while one instance takes over 70000 seconds to solve in the plain encoding. The IDP

translation is however outperformed by the optimized version of the direct SAT translation. In the optimized encoding, the longest recorded runtime is slightly above 100 seconds. In [11] an even better time is obtained by including extra redundant clauses. It is an interesting question whether the performance gap can be closed by adding redundant constraints or by parameter tuning of the SAT solver.

6 Conclusion

We have described three NP-hard problems together with their solution with FO(·) and the IDP framework. The first problem is in the domain of stemmatology. We developed an IDP solution that outperformed the dedicated procedural code of a researcher in the field. We proved the problem is NP-complete and constructed problem instances on which the original code errs. The resulting program is a useful tool for the researchers and is integrated in [20]. In a trivial extension we made the `colorOf` function partial; then only those manuscripts are colored as necessary for making the coloring consistent. This gives useful insight to the philologist. Another planned variation does not enforce a unique source for each color, but minimizes the number of sources. This can provide additional insight when the data are in disagreement with the hypothesized stemma.

The second problem addressed the construction of a minimal common supergraph out of a given set of phylogenetic trees. The use of $\text{FO}(\cdot)^{\texttt{IDP}}$ allowed the authors of [14] to quickly explore various approaches and to arrive at an approximate method that gives good results.

These two applications illustrate the versatility of FO(·) for solving a new problem. The third application compares an FO(·) formalization with a state of the art solution for the NP-complete problem of learning a DFA. While we observe a performance gap with a highly tuned competition winner, our solution performs better than the initial encoding of [11]. On the other hand, the FO(·) formalization took much less effort to develop and offers a lot more flexibility, e.g., to change the optimization criterion. The application is also a good benchmark for further improving the IDP system.

We hope these applications inspire others to try out the IDP framework. It is a small step for computer scientists knowledgeable about logic and Prolog. While our solutions look deceivingly simple, a word of caution is in place. A first solution is hardly ever the best solution; be convinced that it can be done simpler. Simpler not only means a more concise and elegant model but also, almost always, a better performance. Try to break up complex constraints in simpler ones, requiring less variables.

A common beginners misconception we observed, is to use one function (or relation) for information partially given in the input structure and to use another function that extends the partial function into a total one while that same function can serve by declaring it total (the default for functions) and stating that the input structure is partial.

We also observed a very useful programming pattern. In each of our applications, some equivalence class over some given elements is to be constructed. Representing this relationships as a function from the elements to the set of equivalence classes is an excellent choice (the function `colorOf` in stemmatology and in DFA learning, the function `label` in the phylogenetic trees).

Acknowledgements Caroline Macé and Tara Andrews brought some of the authors in touch with stemmatology and Tara explained them the working of the procedural code.

References

1 T. Andrews and C. Macé. Beyond the tree of texts: Graph methods for stemmatic analysis. *In preparation*, 2012.

2 P. Baret, C. Macé, P. Robinson, C. Peersman, R. Mazza, J. Noret, E. Wattel, Van Mulken M., Robinson P., A. Lantin, P. Canettieri, V. Loreto, H. Windram, M. Spencer, C. Howe, M. Albu, and A. Dress. Testing methods on an artificially created textual tradition. In *The evolution of texts: Confronting stemmatological and genetical methods*, pages 255–283. Istituti editoriali e poligrafici internazionali, Pisa, 2006.

3 Bart Bogaerts, Broes De Cat, Stef De Pooter, and Marc Denecker. The IDP framework reference manual. `http://dtai.cs.kuleuven.be/krr/software/idp3/documentation`.

4 Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

5 Stef De Pooter, Johan Wittocx, and Marc Denecker. A prototype of a knowledge-based programming environment. In *International Conference on Applications of Declarative Programming and Knowledge Management*, 2011.

6 Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Transactions on Computational Logic (TOCL)*, 9(2):Article 14, 2008.

7 Joseph Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Sunderland, MA, 2004.

8 Philippe Gambette. Who is who in phylogenetic networks: Articles, authors and programs. Published electronically at `http://www.atgc-montpellier.fr/phylnet`, 2010.

9 Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp*: A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *LPNMR*, volume 4483 of *LNCS*, pages 260–265. Springer, 2007.

10 E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

11 Marijn Heule and Sicco Verwer. Exact DFA identification using SAT solvers. In *Grammatical Inference: Theoretical Results and Applications, ICGI 2010*, pages 66–79, 2010.

12 Daniel H. Huson, Regula Rupp, and Celine Scornavacca. *Phylogenetic Networks: Concepts, Algorithms and Applications*. Cambridge University Press, November 2010.

13 Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. Lua – an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

14 Anthony Labarre and Sicco Verwer. Merging partially labelled trees: hardness and an efficient practical solution. *In preparation*, 2012.

15 Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2002.

16 Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

17 David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 430–435. AAAI Press / The MIT Press, 2005.

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

19 The StaMinA competition, Learning regular languages with large alphabets. `http://stamina.chefbe.net/`, 2010.

20 Stemmaweb, a collection of tools for analysis of collated texts. `http://byzantini.st/stemmaweb/`, 2012.

21 Tommi Syrjänen and Ilkka Niemelä. The smodels system. In Thomas Eiter, Wolfgang Faber, and Mirosław Truszczyński, editors, *LPNMR*, volume 2173 of *LNCS*, pages 434–438. Springer, 2001.

22 Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

23 Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In Marc Denecker, editor, *LaSh*, pages 153–165, 2008.