Analyzing manuscript traditions using constraint-based data mining

Tara Andrews[1] and **Hendrik Blockeel**[2] and **Bart Bogaerts**[2] and **Maurice Bruynooghe**[2] and **Marc Denecker**[2] and **Stef De Pooter**[2] and **Caroline Macé**[1] and **Jan Ramon**[2]

Abstract. Data mining tasks and algorithms are often categorized as belonging to one of a few specific types: clustering, association rule discovery, probabilistic modeling, etc. For some time now, it has been recognized that concrete tasks do not always fit nicely in this categorization. The concepts of constraint-based data mining and inductive querying have been proposed to alleviate this problem; they offer more flexibility with respect to specifying the task. In this paper, we illustrate an approach that goes one step further: we show how a general-purpose declarative modeling language can be used to specify and solve data mining tasks in the area of philology. These tasks have the following properties: they are easily described in words; they are of real interest to philologists; they cannot be performed using standard querying or data mining systems; manually programming a solution for them is challenging, time-consuming and error-prone. We show that a prototype declarative programming framework, IDP, allows for easy modeling and efficient solving of these tasks. We conclude from this case study that the declarative modeling approach to data mining has a large potential and deserves further investigation.

1 Introduction

In data mining, many standard types of techniques exist, such as association rule discovery, decision tree induction, probabilistic modeling, clustering, etc. Once the data have been preprocessed into a format ready for analysis, these standard techniques can be run by simply "pushing the button", possibly after setting a few parameters. However, these techniques do not always produce exactly the type of results that the user wants. The term "constraint-based data mining" is often used to refer to data mining approaches where the user can impose constraints on the patterns he is looking for. For instance, association rule discovery algorithms may return only rules that fulfill certain syntactical constraints; clustering algorithms may accept must-link and cannot-link constraints that impose certain conditions on the returned clustering, etc. By allowing the user to specify such constraints, the data mining techniques become much more flexible: the user can tune them towards his own interests.

Instead of simply restricting the patterns returned by a standard data mining system, constraints may more generally be used to express background knowledge about the domain, or define the task one wants to solve. Nijssen and Guns [9], for instance, have shown that the classical task of frequent pattern discovery, possibly with constraints imposed on these patterns, can be defined completely as a constraint solving problem, and solved efficiently using a general-purpose solver. This shows that redefining classical data mining tasks in terms of constraints, and using a standard solver, can add a lot of versatility to the data mining process (users can define more precisely what they want), at a low efficiency cost.

Taking this one step further, one may try to find a single declarative language in which practically any data mining task could be defined, and solved efficiently, removing the need for many specialized systems. Even data mining tasks that do not belong to one of the predefined categories can then be solved quite easily. This approach is very similar to what is sometimes called "inductive querying" or "query-based data mining", but is more flexible in the sense that some data mining task descriptions may be too complicated to easily fit into a query, and more suitable for the more modular type of description that programming languages offer.

In this paper, we describe an approach where a declarative modeling language, $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$, is used to describe the data, the background knowledge, and the data mining task. The application of interest is situated in the area of stemmatology, a sub-field of philology in which the history of manuscript traditions is studied. The mining tasks are motivated by real questions from philologists. The framework we use is called IDP. It provides seamless integration of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ with a procedural language that can be used for reading files, preprocessing data, formatting the output, and calling the solver.

The remainder of this paper is structured as follows. Section 2 describes the application domain, stemmatology, in some more detail. Section 3 describes the IDP framework. In Section 4 we discuss several data mining tasks that are relevant for stemmatologists, and show how they can be solved using IDP. Section 5 presents conclusions.

2 Stemmatology

Before the invention of the printing press, texts were copied manually by scribes. This copying process was not perfect: scribes often modified texts, accidentally or intentionally. As a result, for many old texts the surviving copies 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 dis-

[1] KU Leuven, Faculty of Arts
[2] KU Leuven, Department of Computer Science

appeared, 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 analysing 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. It indicates which manuscripts have been copied from which other manuscripts ("parents"), and which manuscript is the original source. It may include both extant (currently existing and available) and non-extant ("lost" or conjectured) manuscripts. Although stemmata are often assumed to be tree-shaped, they need not be. Sometimes, a manuscript has been copied partially from one manuscript, and partially from another, so it has multiple parents. In general, a stemma is a connected directed acyclic graph (DAG) [1].

The 19th century philologist Karl Lachmann was among the first to apply a principled method for reconstructing stemmata from sets of manuscripts [11]. 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.[3] 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 on new algorithms, better suited for the stemmatological context, continues [2].

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 have received even less attention, and they can be very diverse. The data mining tasks we address in this paper belong to this category. Multiple tasks will be addressed, but before discussing them in more detail, we first introduce the IDP framework.

3 IDP: an FO(·)$^{\text{IDP}}$ knowledge-based programming environment

IDP [5] is a logic-based programming environment that extends the Lua [7] scripting language with support for constructing, manipulating, and performing inference with logical objects such as vocabularies, theories (in the FO(·)$^{\text{IDP}}$ language), structures, and terms.

3.1 FO(·)$^{\text{IDP}}$

The term FO(·) is used to denote the family of extensions of first-order logic (FO); for instance, FO(ID) is the extension of FO with inductive definitions. In this text, the focus lies on FO(·)$^{\text{IDP}}$, the FO(·) language supported by the IDP framework. FO(·)$^{\text{IDP}}$ extends FO with (among others) *types*, *arithmetic*, *aggregates*, *partial functions* and *inductive definitions*. This section focuses on what is needed for this paper; more information on FO(·)$^{\text{IDP}}$ can be found in [13] and [4].

An FO(·)$^{\text{IDP}}$ theory is a set of typed FO sentences and inductive definitions. We explain its syntax using a theory for solving a shortest path problem, presented in Figure 1. First, one declares a vocabulary, `sp_voc` in the example. It contains declarations of types (e.g., `node`), typed constants (e.g., `from`) and predicates (e.g., `edge(node,node)`). Next, one defines a theory over some vocabulary. It contains (well-typed) FO sentences constructed from symbols of the vocabulary and connectors and quantifiers, including `&`($\wedge$), `|`($\vee$), $\sim$($\neg$), `!`($\forall$), `?`($\exists$) and $\sim$`=`($\neq$). Finally, it also contains inductive definitions, sets of rules of the form `!` x_1 ...x_n `:` `(P(`t_1`,...,`t_n`)` `<-` `B)` where `B` is an FO formula. The semantics of inductive definitions is the well-founded semantics [12], as this semantics formalizes the intended meaning of all common forms of definitions [6].

The theory in Figure 1 expresses that the predicate `edgeOnPath` is the set of edges of a path from the node `from` to the node `to`. The following conditions must hold for such paths. (1) `edgeOnPath` is a subset of `edge`. (2) `from` and `to` are the first, respectively last node and hence, have no entering, respectively exiting edge in the path. (3) Each node on the path has at most one entering and at most one exiting edge in the path. Here, `?<2 y : edgeOnPath(y,x)` means that there are strictly less than 2 y's that have an edge to x in the path. This can be expressed also using an aggregate `#{ y : edgeOnPath(y,x)} < 2` or using a sentence `! y1 y2 : edgeOnPath(y1,x) & edgeOnPath(y2,x) => y1=y2`. (4) `from` can reach `to` using edges of the path and (5) `from` can reach each node on the path (e.g., `{ (from,to), (c,c)}` is not a path). The predicate `reaches(x,y)` is defined inductively.

The model semantics of FO and FO(·)$^{\text{IDP}}$ is based on the notion of *structures*. A structure is an assignment of values to symbols: sets to type symbols, domain elements to constants, relations to predicate symbols and functions to function symbols. A *model* of a theory is an assignment that satisfies all expressions of the theory. In many problems, structures are useful to represent data. The structure defined in Figure 1 interprets the symbols `node`, `edge`, `from` and `to` from vocabulary `sp_voc`. Note that no value is specified for `edgeOnPath`, which means that this is a partial structure of `sp_voc`.

3.2 The IDP programming environment

Apart from vocabularies, theories and structures, the programming environment also contains *procedures* and *terms*. As said above, procedures are written in the language Lua and may call a range of predefined methods operating on the logical objects. The most relevant are the following:

- `sat(<theory>,<structure>)` is a boolean function that returns true iff the theory is satisfied by the structure.
- `modelexpand(<theory>,<structure>)` takes as input a theory over vocabulary Σ and a partial structure assigning values to some symbols in Σ, and returns a list of models of the theory that expand the partial structure.
- `minimize(<theory>,<structure>,<term>)` returns a list of models of the theory expanding the input structure in which the numerical term is minimal.

[3] 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.

```
vocabulary sp_voc {
  type node
  from, to: node
  edge(node,node)
  edgeOnPath(node,node)
  reaches(node,node)
}
theory sp_theory: sp_voc {
  ! x y : edgeOnPath(x,y) => edge(x,y).
  ~(? x : edgeOnPath(x,from)) & ~(? x : edgeOnPath(to,x)).
  !x: (?<2 y: edgeOnPath(y,x)) & (?<2 y: edgeOnPath(x,y)).
  { reaches(x,y) <- edgeOnPath(x,y).
    reaches(x,y) <- reaches(x,z) & reaches(z,y). }
  reaches(from,to).
  ! x y : edgeOnPath(x,y) => reaches(from,y).
}
structure sp_struct: sp_voc {
  node = {A..D} // shorthand for A,B,C,D
  edge = {A,B; B,C; C,D; A,D}
  from = A
  to = D
}
term lengthOfPath: sp_voc {
  #{ x y : edgeOnPath(x,y) }
}
procedure main() {
  sols = minimize(sp_theory,sp_struct,lengthOfPath)
  if sols
  then print(sols[1])
  else print("No models exist.\n")
  end
}
```

Figure 1. `main()` finds the shortest path for the given data.

These methods are implemented with state-of-the-art technologies, such as the grounder GIDL [14] and solver MINISAT(ID) [8]. The solver is an extended SAT solver with support for aggregate expressions, inductive definitions and branch-and-bound optimization. MINISAT(ID) also has support for finite domain constraints, using the propagation techniques described in [10] or, alternatively, interfacing with the GECODE Constraint Programming engine.

In Figure 1, `main()` performs the minimization inference on the path theory `sp_theory` and prints out the first model in the list if it exists. The term to be minimized here is called `lengthOfPath` and is declared as the number of pairs in `edgeOnPath`. A minimal model of this term is indeed a shortest path from A to D.

An IDP-program is a collection of declarations of vocabularies, theories, structures, terms and procedures. For an in-depth treatment of the framework, see [4].

3.3 Illustration: frequent itemset mining

Figure 2 shows how the task of frequent itemset mining can be described in IDP. We include it only as an example of how a classical data mining task can be defined in IDP; an in-depth study of the use of general-purpose solvers for frequent itemset mining is given by Nijssen and Guns [9].

The vocabulary declares two types `Transaction` and `Item`, the threshold `Freq`, the predicate `Includes(t,i)` which expresses that transaction `t` includes item `i`, and finally, the unary predicate `FrequentItemset`. The theory simply expresses that the number of transactions that include all items

```
vocabulary FrequentItemsetMiningVoc {
  type Transaction
  type Item
  Freq: int
  Includes(Transaction,Item)
  FrequentItemset(Item)
}
theory FrequentItemsetMiningTh: FrequentItemsetMiningVoc {
  #{t: !i: FrequentItemset(i) => Includes(t,i) } >= Freq.
}
structure Input : FrequentItemsetMiningVoc {
  Freq = 7 // threshold for frequent itemsets
  Transaction = { t1; ... ; tn } // n transactions
  Item = {i1 ; ... ;  im }       // m items
  Includes = {t1,i2; t1,i7; ...} // items of transactions
}
```

Figure 2. An IDP description of frequent itemset mining.

in `FrequentItemset` is at least `Freq`. The structure describes the data: it specifies the threshold, all transactions and items, and the `Includes` relation.

In any structure of this vocabulary that extends `Input` and satisfies the theory, `FrequentItemset` represents a frequent itemset. As a consequence, the task of computing all frequent itemsets is solved by letting IDP generate all such models.

4 Data mining tasks

4.1 Formalization of the context

A *tradition* is a set of manuscripts that are related in a particular way (specifically, they can be considered variants of the same text, the variation having been introduced through imperfect manual copying). A dataset represents one tradition. Each manuscript is described by a fixed set of attributes $A_1, \ldots, A_n$, each of which has a nominal domain $Dom(A_i)$. Typically, one attribute represents a particular location in the text, and the different elements of its domain represent the different variant readings at that location.

A *stemma* is a connected DAG $G(V, E)$, where V contains one element for each manuscript, and $(v, w) \in E$ if and only if manuscript w was (wholly or partially) copied from manuscript v.

Given an attribute A_i, we can label nodes in G with their value for A_i (observed or predicted). The whole labeling is then a partial or complete function $\lambda : V \to Dom(A_i)$. A labeling λ *extends* another labeling λ' if and only if, whenever $\lambda'(v)$ is defined, $\lambda(v)$ is also defined and $\lambda(v) = \lambda'(v)$.

4.2 The datasets

We will evaluate the feasibility of our declarative modeling approach on five datasets (`http://byzantini.st/stemmaweb/`). Three are artificial traditions, constructed with the purpose of testing stemmatological methods; for these, the correct stemma is known. They may be found at `http://www.cs.helsinki.fi/u/ttonteri/casc/data.html`. The other two (Sermon 158 and Florilegium) are real traditions, with stemmata that have been constructed according to current philological best practice. Table 1 gives an overview of the number of manuscripts and attributes for each tradition, as well as the

number of edges in the DAG. Note that a tree with n nodes always has $n-1$ edges. Thus, three of the five stemmata are tree-shaped; two are "almost" tree-shaped, in the sense that they have few additional edges.

Table 1. The five traditions used in this work.

Name	nodes	edges	attributes
Notre Besoin	13	13	44
Parzival	21	20	122
Florilegium	22	21	547
Sermon 158	34	33	270
Heinrichi	48	51	1042

4.3 Task 1: Consistency checking

New variants are introduced when a scribe, intentionally or accidentally, changes a text. Often, a variant reading at one particular location is complicated enough to consider it unlikely that exactly the same reading has been introduced multiple times independently. We therefore introduce the following terminology. A manuscript is a *source* for an attribute if none of its parents have the same value for that attribute. An attribute is *consistent* with a stemma if the values observed for the attribute can be explained using only one source per value. Formally, given a stemma $G(V, E)$, an attribute A, and a (partial) labeling λ indicating which nodes in V have which value for A, A is consistent with G if and only if a complete labeling exists that extends λ and has one source for each value. Figure 3 illustrates these concepts.

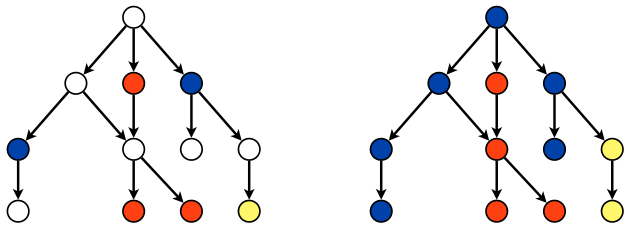

Figure 3. Left: a partial labeling showing for a given attribute which manuscripts have which value (indicated by colors). Right: a complete extension of that labeling with one source per label. Because such an extension exists, the attribute is consistent with the stemma.

Now consider the task of checking whether a given attribute is consistent with a given DAG. This requires searching for a complete extension of the given labeling that has one source per label. This problem is NP-hard [3] and does not reduce to any problem solved by standard data mining techniques.

The problem is easily modeled in IDP, however. Figure 4 shows an IDP program that determines for each attribute in each dataset whether it is consistent with the stemma. The problem of checking consistency of a single attribute with a stemma is defined under the header "Knowledge base". This definition is very simple: besides introducing the vocabulary (there are manuscripts; there are variant readings; manuscripts may be copied by other manuscripts; with

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

/* ---------- Knowledge base ------------------------- */
vocabulary V {
  type Manuscript
  type Variant
  CopiedBy(Manuscript,Manuscript)
  VariantIn(Manuscript): Variant
}
vocabulary Vsrc {
  extern vocabulary V
  SourceOf(Variant): Manuscript
}
theory Tsrc : Vsrc {
  ! x : (x ~= SourceOf(VariantIn(x))) =>
        ? y: CopiedBy(y,x) & VariantIn(y) = VariantIn(x).
}

/* --------- Check whether sample fits stemma -------- */
procedure check(sample) {
  idpintern.setvocabulary(sample,Vsrc)
  return sat(Tsrc,sample)
}

/* ---------- Procedures for processing -------------- */
procedure process(name) {
  io.write("Processing ",name,".\n")
  local path = "data/"
  local stemmafilename = path..name..".dot"
  local samplefilename = path..name..".json"
  processFiles(stemmafilename,samplefilename)
}
procedure processFiles(stemmafilename,samplefilename) {
  local stemma,nbnodes,nbedges = readStemma(stemmafilename)
  io.write("Stemma has ",nbnodes," nodes and ",nbedges, " edges.\n")
  local nbp,nbs,time = processSamples(stemma,samplefilename)
  io.write("Found ",nbp," positive out of ",nbs," groupings ")
  io.write("in ",time," sec.\n")
}
procedure readStemma(stemmafilename) {
  ... // 19 lines
}
procedure processSamples(stemma,samplefilename) {
  ... // 23 lines
}
```

Figure 4. The IDP code for checking the consistency of all the attributes in a dataset with a hypothesized stemma for that dataset. The `.dot` and `.json` files contain the stemma and attribute-value-table, respectively.

each manuscripts is associated a variant; each variant has one source, which is a manuscript), it only states that if a manuscript is not the source of a variant, it must have a parent with that same variant.

Besides this declarative specification, the IDP program contains procedural code that loads the data files, builds for each dataset and attribute a structure that represents a partially labeled DAG, calls a solver to check the satisfiability of the theory for this structure (`sat(Tsrc, sample)`), and produces readable output. We include some of the procedural code to illustrate the seamless integration of declarative and procedural knowledge in IDP.

The IDP program determines consistency for all attributes and datasets in a matter of seconds:

```
> 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 5 sec.
Processing sermon158.
Stemma has 34 nodes and 33 edges.
Found 64 positive out of 270 groupings in 4 sec.
Processing heinrichi.
Stemma has 48 nodes and 51 edges.
Found 1 positive out of 1042 groupings in 28 sec.
>
```

It is interesting to compare these results with earlier results obtained using a procedural implementation of the consistency check by one of the authors. This procedural implementation contained 370 lines of Perl code, using a graph library as working horse, and could not be shown correct. This was our main motivation for trying a declarative approach. Our declarative specification is more easily verified, is solved faster than with the Perl version, and eventually allowed us to show that the original procedural implementation was not correct [3]. This demonstrates the usefulness of the IDP framework for non-traditional types of data analysis.

4.4 Task 2: Determining the minimal number of independent sources

A remarkable result of the previous analysis was that for some of the artificial traditions, very few attributes were consistent with the stemma (for Heinrichi, 1 out of 1042). This indicates that either the artificial traditions are not representative for real traditions, or the assumption that each variant originates only once is not realistic.

Given that inconsistent attributes occur so often, one may wonder how often multiple introductions of the same variant must have occurred. In other words: what is the smallest number of sources needed to explain the observations?

This question is again easily expressed in IDP, now as a minimization problem. Figure 5 shows this. In the vocabulary, the function `SourceOf` (which allows only one source per label) is replaced by a predicate `IsSource`, which indicates whether a node x is a source or not. The theory simply defines `IsSource` as such; nothing else is needed. Further, a term `NbOfSources` is introduced that counts the number of sources, and a procedure is introduced that returns a model in which the number of sources is minimal. The procedure `minimize` performs model minimization, as explained before (in this case, it finds a complete and consistent labeling with a minimal number of sources). Apart from changing a call of `check(sample)` into `minSources(sample)`, and some output formatting, no procedural code needs to be changed.

Figure 6 shows part of the output for the Notre Besoin dataset. All datasets were processed in a few seconds, except for Heinrichi, which took about 5 minutes. Adding more constraints may further reduce processing time, but this was not investigated here.

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

/* --- Find model with minimal number of sources --- */

procedure minSources(sample) {
  idpintern.setvocabulary(sample,Vms)
  return minimize(Tms, sample, NbOfSources)[1]
}
```

Figure 5. IDP code for minimizing the number of sources required to explain the data. Function `SourceOf` is replaced by predicate `IsSource`; the term `NbOfSources`, which counts the number of sources, is introduced; and a procedure is introduced that returns a model with minimal `NbOfSources`.

```
Processing besoin.
Stemma has 13 nodes and 13 edges.
IsSource = { T2; U }
IsSource = { C; T2 }
IsSource = { D; J; L; M; T2; U; V }
... (40 output lines omitted)
IsSource = { B; F; J; T2 }
Minimized for 44 groupings in 0 sec.
```

Figure 6. IDP output indicating a minimal set of sources for each attribute in *Notre Besoin*.

4.5 Task 3: Sources versus reversions

Up till now, we said a manuscript introduces a variant if none of its parents have that variant. However, stemmatologists distinguish the case where a manuscript reverts to an older variant (which did not occur in the parents but occurs somewhere among the ancestors) from the case where it introduces a completely new variant. It may be interesting to minimize a cost function where sources have a higher cost than reversions.

This is again easily modeled in IDP. An inductive definition is provided for the predicate `IndirectAncestor`. Further, there are now three types of nodes: sources, reversions, and copies. This is modeled by introducing a function `ClassOf` that classifies nodes as one of `Source`, `Copy` or `Revert`. The theory and term-to-be-optimized are shown in Figure 7. In this case, some more changes to the procedural code (not shown) are required, for technical reasons beyond the scope of this paper. A part of the output for Notre Besoin is shown in Figure 8. Finding the model with minimal cost takes significantly longer than for the previous tasks: seconds to minutes for the first four datasets, and about 18 hours for Heinrichi. We currently do not know why it takes so much longer for

```
vocabulary Vcls {
  extern vocabulary V
  type Cost isa nat
  type Class
  Copy: Class
  Revert: Class
  Source: Class
  ClassOf(Manuscript): Class
  IndirectAncestor(Manuscript,Manuscript)
}
theory Tcls : Vcls {
  !x: (ClassOf(x)=Copy) <=>
      ?y: CopiedBy(y,x) & VariantIn(y) = VariantIn(x).
  !x: (ClassOf(x)=Revert) <=>
      ClassOf(x) ~= Copy &
      ?y: IndirectAncestor(y,x) &
          VariantIn(y) = VariantIn(x).
  {!x y: IndirectAncestor(x,y) <-
         ?z: CopiedBy(x,z) & IndirectAncestor(z,y).
   !x y: IndirectAncestor(x,y) <-
         ?z: CopiedBy(x,z) & CopiedBy(z,y).}
  NbOfSources = #{x: ClassOf(x)=Source}.
  NbOfReverts = #{x: ClassOf(x)=Revert}.
}
term TotalCost : Vcls {
  3 * NbOfSources + NbOfReverts
}
```

Figure 7. IDP code for minimizing the cost of a labeling, where each source has a cost of 3 and each reversion a cost of 1. The function `ClassOf` indicates which of three classes a node belong to: `Source`, `Revert` or `Copy`. The theory inductively defines the `IndirectAncestor` predicate, and defines the conditions under which a node has class `Copy` or `Revert`.

```
Processing besoin.
Stemma has 13 nodes and 13 edges.
ClassOf = {T2->s; U->s}
ClassOf = {C->s; T2->s}
ClassOf = {A->s; D->r; J->r; L->s; M->r; T2->s; U->r; V->r}
... (40 output lines omitted)
ClassOf = {A->s; B->s; F->r; J->r; T2->s}
Minimized for 44 groupings in 3 sec.
```

Figure 8. IDP output (edited for conciseness) showing sources (s) and reversions (r) for minimal-cost models in *Notre Besoin*.

Heinrichi. It may be possible to reduce runtime by adding more constraints to better guide the solver. Alternatively, approximate methods for optimization could be explored.

Tasks 2 and 3 demonstrate the ease with which new data mining tasks can be defined and solved, once the procedural code for preprocessing etc. is in place.

5 Conclusions

While many data mining tasks can be solved using off-the-shelf tools, some tasks deviate significantly from the standard ones and cannot be performed using any of the standard methods or query languages. Inductive query languages may provide a solution when the data mining task can be formulated as a relatively simple query. Here, we have explored an alternative approach that consists of defining the task using a declarative modeling language, then performing inference using advanced, built-in, constraint solving and optimization techniques. More concretely, the IDP framework has been used for addressing data mining tasks in stemmatology. As it turns out, IDP has the power and versatility to define these data mining tasks with relative ease, and solve them efficiently and provably correctly. Important elements of IDP that contribute to this are the ability to formulate constraints in full first-order logic, to include inductive definitions, to define aggregate functions, and to solve satisfiability and optimization problems. One opportunity for improvement is the minimization procedure, which might benefit from approximate methods. We conclude that declarative modeling frameworks such as IDP have a large potential for data mining, and this type of approaches deserves further investigation.

Acknowledgements

Research supported by Research Foundation - Flanders (FWO-Vlaanderen), KU Leuven CREA/10/004, and ERC Starting Researcher Grant 240186.

REFERENCES

[1] T. Andrews and C. Macé, 'Beyond the tree of texts: Building an empirical model of scribal variation through graph analysis of texts and stemmata', *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*, 255–283, Istituti editoriali e poligrafici internazionali, Pisa (2006).

[3] H. Blockeel, B. Bogaerts, M. Bruynooghe, B. De Cat, S. De Pooter, M. Denecker, A. Labarre, J. Ramon, and S. Verwer, 'Modeling machine learning and data mining problems with FO(.)', in *Proc. 28th ICLP*, Leibniz International Proceedings in Informatics (2012). To appear.

[4] B. Bogaerts, B. De Cat, S. De Pooter, and M. Denecker. The IDP framework reference manual. `http://dtai.cs.kuleuven.be/krr/software/idp3/documentation`.

[5] S. De Pooter, J. Wittocx, and M. Denecker, 'A prototype of a knowledge-based programming environment', in *International Conference on Applications of Declarative Programming and Knowledge Management* (2011).

[6] M. Denecker and E. Ternovska, 'A logic of nonmonotone inductive definitions', *ACM Transactions on Computational Logic (TOCL)*, **9**(2), Article 14 (2008).

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

[8] M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe, 'SAT(ID): Satisfiability of propositional logic extended with inductive definitions', in *Proc. SAT 2008*, volume 4996 of *LNCS*, pp. 211–224. Springer (2008).

[9] S. Nijssen and T. Guns, 'Integrating constraint programming and itemset mining', in *ECML/PKDD (2)*, volume 6322 of *Lect. Notes in Comp. Sc.*, pp. 467–482. Springer (2010).

[10] C. Schulte and P.J. Stuckey, 'Efficient constraint propagation engines', *ACM Transactions on Programming Languages and Systems*, **31**(1) (2008).

[11] S. Timpanaro and G.W. Most (translator), *The Genesis of Lachmann's Method*, University of Chicago Press, 2005.

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

[13] J. Wittocx, M. Mariën, and M. Denecker, 'The IDP system: a model expansion system for an extension of classical logic', in *LaSh*, pp. 153–165 (2008).

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