KYBERNETIKA — VOLUME 40 (2004), NUMBER 3, PAGES 275 - 292

EFFICIENCY-CONSCIOUS PROPOSITIONALIZATION FOR RELATIONAL LEARNING

FILIP ŽELEZNÝ

Systems aiming at discovering interesting knowledge in data, now commonly called data mining systems, are typically employed in finding patterns in a single relational table. Most of mainstream data mining tools are not applicable in the more challenging task of finding knowledge in structured data represented by a multi-relational database. Although a family of methods known as inductive logic programming have been developed to tackle that challenge by immediate means, the idea of adapting structured data into a simpler form digestible by the wealth of AVL systems has been always tempting to data miners. To this end, we present a method based on constructing first-order logic features that conducts this kind of conversion, also known as propositionalization. It incorporates some basic principles suggested in previous research and provides significant enhancements that lead to remarkable improvements in efficiency of the feature-construction process.

We begin by motivating the propositionalization task with an illustrative example, review some previous approaches to propositionalization, and formalize the concept of a first-order feature elaborating mainly the points that influence the efficiency of the designed feature-construction algorithm.

Keywords: machine learning, inductive logic programming, propositionalization

AMS Subject Classification: 68T30

1. INTRODUCTION

A family of computer programs now collectively termed *data mining systems* aim to discover interesting knowledge in observational data. Their underlying algorithms are usually based on principles of inductive learning and most commonly they seek for dependencies among attributes (columns) in a single relational table, that hold in a sufficient number of instances (rows) in that table. Very often such dependencies are expressed in languages reminding propositional logic, such as *decision trees* in the C4.5 algorithm [21], *classification rules* as in CN2 [4], or *association rules* produced by the APRIORI system [1], or the algorithm of GUHA [8]. Such systems are generally called *attribute value learners* (AVL) owing to their way of forming hypotheses out of propositions which assign a constant value to a specified attribute. For instance, a rule such as

```
size = large, luxury = high → affordable = no
```

would be a typical example of the kind of representation of knowledge discovered with an AVL system applied to a relational table with attributes containing those appearing on the left-hand side of the equalities. To date, there is a vast quantity of mature and perpetually augmented AVL systems accompanied with auxiliary methods such as for pre-learning attribute selection [17], pre-processing [24], etc.

It has been however widely recognized [6, 7] that AVL systems cannot stand well to the challenge of discovering knowledge from highly structured or multi-relational data, which manifests itself in important problems such as predicting mutagenicity of chemical compounds [23], pharmacophore discovery and others.

A toy problem will illustrate the difficulty inherent to such domains: consider the set of trains depicted in Figure 1. Here the task is to find a rule discriminating between east-bound and west-bound trains, each having a variable number of different cars with a variable number of wheels and different loads.

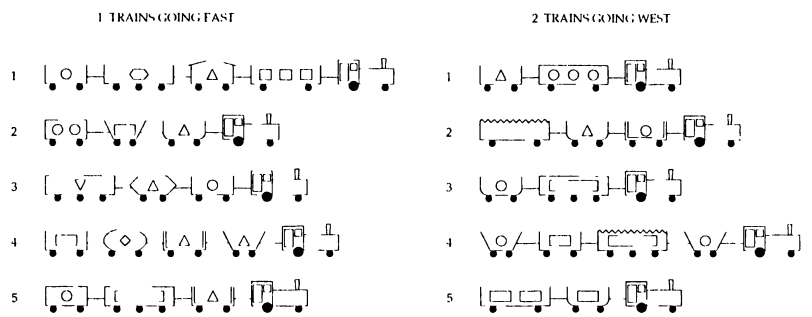

Fig. 1. The 10-train Michalski's East-West challenge.

To represent the data in a relational database, one would resort to introducing a table with each row corresponding to a train (containing an attribute indicating the train's direction), another table for cars (with an attribute linking the given car to a train), yet another table for loads (linked to cars), etc. Unfortunately, to apply an AVL learner, a data miner will have to join these tables into a single one, such that each row thereof has to bear the whole structural information about a train. In order to achieve this by means of a database query, it will be necessary[1] to unify the sizes of train descriptions by formally adding null ('dummy') cars to each train up to the number of cars occurring in the longest train, similarly null loads to cars with a smaller-than-maximum number of loads, etc. The number of attributes in such a joined table will clearly be excessive, with many null fields e. g. in rows corresponding to short trains or those mostly consisting of 'underloaded' cars. Another aspect of this approach is even more painful: as an AVL discriminates instances by assigning a value to a specific attribute, it will not be able to arrive to rules with existential quantification, such as *a train is east-bound if one of its cars has three wheels.*

[1]We thank the reviewer for pointing out this issue to us.

In the last 15 years, a considerable effort has been devoted within the field of *inductive logic programming* (ILP), to devise learning techniques able to cope with such problems [6] by representing both the input data and the resulting hypotheses in the language of Prolog. Here, a learning example describing the first east-bound train t_1 (thus being a *positive* example of the east-bound *concept*) would read[2]

$$\texttt{east(t1)} \leftarrow \texttt{hasCar(t1,c1),has2Whells(c1),hasLoad(c1,l1),box(l1)}\ldots \text{(etc.)} \tag{1}$$

Note that commonly in ILP, the first literal is considered the example, while the rest of literals belong to the *background knowledge* – a database containing the descriptions of all trains. An ILP system might then generalize a set of positive and negative examples towards a rule[3] proposing that a train goes east if it has a roofed car with a circle load:

$$\texttt{east(T)} \leftarrow \texttt{hasCar(T,C),hasLoad(C,L),circle(L),hasRoof(C).} \tag{2}$$

In our example, this rule would represent a hypothesis that is correct (in that its antecedent does not hold for any negative example, i.e. a west-bound train) but incomplete (in that it does not hold for all east-bound trains). The final hypothesis would thus have to be attained by adding further rules, while `east(T)` would be concluded for a given `T` if any of the rules' antecedents was satisfied for `T`.

Despite the relative maturity of ILP research, the selection of available AVL systems suitable for various kinds of data mining tasks is overwhelmingly larger than that offered by ILP. One thus wonders whether structured / multi-relational data could be adapted into a form allowing to be processed by an AVL system while avoiding the obstacles imposed by the table-joining approach, as demonstrated above on the example of trains. Recently, the term *propositionalization* has been accepted [11] to describe this goal.

Our system EFFEDRIN[4] implements a procedure for propositionalization, which is briefly as follows. First, generate a finite set of Prolog queries ('features'), each expressing a property of an object ('individual') under investigation (e. g. a train). An example of a feature may be `hasCar(T,C),long(C)`, where the *key* variable T binds to an individual. Then produce a single relational table, where each row corresponds to an individual, each column to a feature, and each field to the truth value of the corresponding feature w.r.t. the corresponding individual. Such a table is provided to an AVL system whose result (e. g. a set of rules) is interpreted by plugging the feature definitions in place of the corresponding attribute-identifiers occuring therein. This approach can be viewed as a middle-ground between attribute-value learning and 'full-power' inductive logic programming. Obviously, not all relational learning problems can be solved via propositionalization, for example those where the target concept includes recursion, such as many benchmarks of automatic logic program synthesis [14]. However, by sacrificing part of generality[5], we gain the advantage of

[2] Lower case arguments denote constants.

[3] Upper cases stand for variables.

[4] Efficiency-minded First-order FEature DeRivation for INductive learning

[5] Corresponding to reducing the method scope to so called *individual-centered* domains [14].

a strong language bias. The language of features we shall define will be considerably more constrained and easier to handle than unrestricted Prolog.

The current part of the paper is organized followingly. Next we review the existing previous work on propositionalization and address a previous incarnation of the presently described system. Afterwards we formalize the fundamentals of the implemented feature construction method paying specific attention to the points that influence the efficiency of the designed algorithm.

2. RELATED WORK

The idea of converting a relational learning problem into a propositional one was first materialized in the pioneering system LINUS [15]. Its principle assumed that all target rules were *constrained*, that is, all variables found in the succedent (head) of a rule occur also in its antecedent (body). LINUS thus can learn rules such as

$$\texttt{triangle(A,B,C)} \leftarrow \texttt{link(A,B), link(A,C), link(B,C).}$$

To learn such rules, LINUS considers the set of all background knowledge predicates with all possible placements of the head variables as the predicate arguments. Then a distinct binary attribute is assigned to each element of the set, holding its truth value for variable instantiations determined by the head of each learning example. With this attribute representation, a set of rules such as the above is then learned via an AVL. By imposing a fixed maximum arity of background predicates together with the assumption of constrained rules, LINUS guaranteed an efficient representation of a relational problem by propositional means. Unfortunately, the latter assumption is clearly intolerably strong in common domains. Later improvements implemented in the system DINUS [5] alleviated the assumption by allowing to learn *determinate* rules, where each body variable not found in the head had to be uniquely determined by the values of those occurring in the head. Still, even the simple rule (2) is neither constrained nor determinate.

A systematically different approach to propositionalization, stemming from the ideas of [26], later implemented in systems such as PROPAL [2], selects one example, such as (1) above, as a *seed* and variabilizes it by assigning a distinct variable to each constant found in the example and replacing all constants occurring therein with the corresponding variables (i. e. multiple occurrences of a constant symbol are replaced by the same variable). Each body literal l in the seed then corresponds to a newly established binary attribute a whose value is determined for each pair of a learning example e and each possible substitution σ of variables in the variabilized seed with constants in e, in such a way that a is true if and only if $l\sigma$ is present in e. Two shortcomings of this approach are obvious. (a) One relational example is presented in general by more than one attribute-value tuples, each corresponding to a different possible substitution, and the relational learning problem thus converts into what is known as a *multiple-instance* learning problem only few AVL systems can tackle. (b) The combinatorial curse here manifests itself in the number of possible substitutions between the seed and each example. This issue has been tackled e. g. by stochastic techniques [22], or using so called 'lazy propositionalization' [2].

Yet another way of propositionalization is represented by systems RELAGGS (RELational AGGregation) [13] or POLKA[6] [10] which abandon the first-order logic principles. Their basic principle is that of extending each attribute tuple in the table of main individuals with aggregate values (such as statistics of numeric values) computed from related records in the rest of the tables. We shall not deal with the details of this branch of propositionalization.

Recently, approaches to logic-based propositionalization, based on the creation of new attributes holding the truth values of first-order logic queries (patterns) related to the original data, have flourished. Besides techniques utilizing some form of the general frequent-pattern search strategy, such as in the RAP system[7][3], several algorithms have been concurrently implemented based on the more constrained understanding of a first-order feature proposed by [14]. These include the systems 1BC (a first-order Bayesian classifier), SINUS [12] (a successor of LINUS), RSD [16] (a relational subgroup discovery system implemented by the first author and available at `http://labe.felk.cvut.cz/~zelezny/rsd`) as well as in the henceforth presented system EFFEDRIN, the successor of RSD's propositionalization component. The contribution of RSD to the propositionalization approach stems from the way RSD controls the complexity of the transformation process by language-bias declaration techniques inspired by state-of-art ILP systems (e.g. Progol [19]). RSD has been shown to provide feasible means of a tackling real-world relational data mining problems including those in telecommunications [16] and mutagenesis prediction [27]. A recent study [12] compared three propositionalization systems (RSD, SINUS and RELAGGS) on six benchmarks of predictive relational learning, feeding feature-sets generated by the respective systems into the J48 decision tree learner (a reimplementation of the well-known C4.5 [21] algorithm within the WEKA wrapper [25]). On two of the six domains, RSD provided the feature set leading to the best predictive accuracy of the induced model. On the problem of predicting mutagenesis of chemical structures – one of the most widely recognized benchmark of relational learning – it provided a feature set leading to the highest predictive accuracy we are aware of ever reported for this domain.

From RSD, EFFEDRIN inherits the understanding of the concept of a first-order feature, along with the technique of language-bias declaration. However, the systems differ in one some principal procedural aspects. While RSD separates the phase of purely syntactical construction of features from the phase of their evaluation on the user's data, EFFEDRIN clones both operations into a common procedure. This allows for early rejections of irrelevant features and, most importantly, for pruning of whole subspaces of the feature search space based on both data-related and syntactical considerations. Furthermore, an operator for ordering of literals in a feature has been redesigned to enable a faster exploration of the feature search space.

[6]The name is meant to be indicative of the two main procedural steps of the algorithm detailed in [10], analogous to the two steps taken in the Czech dance Polka.

[7]RAP searches for *maximal* patterns, i.e. the longest literal conjunctions subject to exceeding a minimum coverage on data. Clearly, this approach goes against the Occam's dogma, which we follow in our work, dictating to bias feature selection towards simple, rather than complex features. It would surely be interesting to compare the two 'antagonist' approaches in a systematic empirical study.

In this necessarily incomplete review we have not mentioned a few other interesting approaches to propositionalization, such as those based on extracting features from theories constructed by a relational learner on the original relational form of the data, various stochastic strategies for feature extraction (see e. g. the recent work-in-progress paper [20]), etc. The source [11] provides a well-elaborated, though neither complete review.

3. ADMISSIBLE FEATURES

In all that follows we assume that three mutually disjoint countable sets of symbols are given: P ("predicate symbols"), T ("variable types") and V ("variable symbols"). We assume there is an irrefexive total order (e. g. alphabetical) $\prec_V$ on V and we denote $w = \max_{\prec_V}(W)$ the maximum of a finite subset $W \subset V$ with respect to $\prec_V$ (i. e. $w \in W$ and $w' \prec_V w$ for all $w' \in W$), similarly we introduce the minimum of W, and finally $v' = s_{\prec_V}(v)$ $(v, v' \in V)$ the successor of v in V (i. e. $v \prec_V v'$ and there is no v'' such that $v \prec_V v'' \prec_V v'$). Further, Ex will be a finite domain whose elements are called "examples". By N we denote the set of natural numbers, $\{\}$ is the empty set and $|S|$ stands for the number of elements in a finite set S. Every expression $p(v_1, v_2, \ldots, v_a)$ where $a \in N$, $p \in P$, $v_1 \ldots v_a \in V$ is called an *atom*.

We shall now describe how to arrive to what we perceive as an admissible set of features. A language used for the feature notation has to respect a grammar, and we first expose the way of its specification. There is a four-stage trajectory from the grammar to an admissible feature-set. We subsequently define (1) which atoms form admissible literals, (2) when a sequence of admissible literals forms a feature candidate, i. e. when such a sequence may be extended to potentially form a feature and will thus be considered as a node in the feature search space, (3) when a feature candidate is an admissible feature and finally (4) which set of admissible features is an admissible feature-set. The main role of the feature grammar is the introduction of variable *moding* and *typing*, which has been recognized as a convenient way of constraining language bias in ILP systems (e. g. [19]).

Definition 1. (Feature grammar.) A *feature grammar* is a pair $G = (\kappa, \Delta)$ where $\kappa \in T$ and $\Delta = \{\delta_1; \delta_2; \ldots; \delta_n\}$, $n \in N$ such that

1. for each $1 \leq i \leq n$

$$\delta_i = [r_i, p_i(m_{i_1} t_{i_1}, \ldots, m_{i_{arity(i)}} t_{i_{arity(i)}})] \tag{3}$$

   where $r_i, arity(i) \in N$, $p_i \in P$ and for all $1 \leq j \leq arity(i)$: $m_{i_j} \in \{+, -\}$, $t_{i_j} \in T$,

2. for each $1 \leq i \leq n$ there is some j such that $m_{i_j} = +$,

3. there is some i and j such that $m_{i_j} t_{i_j} = +\kappa$,

4. for each $1 \leq i, j \leq n$ if $p_i = p_j$ and $arity(i) = arity(j)$ then $i = j$.

We call κ the "key", δ_i the "declaration for the predicate $p_i/arity(i)$"[8] and r_i its "recall value", denoted $\mathbf{recall}_\Delta(p_i/arity(i))$. The rest of symbols in δ_i are said to declare successively the "mode" and "type" of each argument of the predicate. Lastly, the set of all feature grammars is denoted Γ.

The definition thus requires that the recall value (whose meaning will be clarified later in the text), along with the modes and types of all variables are specified for a predicate in a non-ambiguous way (item 4). The + (−) mode will denote an "input" ("output") variable of the declared predicate, such that each declared predicate has to have at least one input (item 2) and at least one predicate declared by a feature grammar takes the key (the type corresponding to the main individual, such as a train in the ongoing example) as an input (item 3). For example, the `hasCar/2` predicate here may have a declaration[9] `[5,hasCar(+car, -train)]`. From the technical point of view, the grammar is specified by the user of the system using the same system interface as in RSD (see e.g. [16]).

The next definition qualifies atoms which are 'correct' with respect to a given declaration.

Definition 2. (Admissible literal.) Let $G = (\kappa, \Delta)$ be a feature grammar. Let further $l = p(v_1 \ldots v_a)$ be an atom. We say that l is a *literal*[10] *admissible by* G if $\delta = [r, p(m_1 t_1, \ldots, m_a t_a)] \in \Delta$ and for all $1 \leq i, j \leq a$, $i \neq j$

1. (type respecting) if $v_i = v_j$ then $t_i = t_j$
2. (distinguished outputs) If $m_i = m_j = -$, $i < j$ and there is no k, $i < k < j$ such that $m_k = -$ then $v_j = s_{\prec_V}(v_i)$.

Further, v_i is said to be of *type* t_i and to be an *input* variable of l if $m_i = +$, otherwise it is an *output* variable thereof. The type of v_i is denoted $\mathbf{type}_\Delta(l, v_i)$ and the set of input and output (respectively) variables of l under Δ is denoted $\mathbf{invars}_\Delta(l)$ and $\mathbf{outvars}_\Delta(l)$. Finally we set $\mathbf{vars}(l) = \{v_1 \ldots v_a\}$ and $\mathbf{recall}_\Delta(l) = \mathbf{recall}_\Delta(p/a)$.

Note that type respecting here merely dictates that a variable cannot appear simultaneously at two argument places declared with different types. However, typing does not delimit the set of values the variable can acquire. While it is completely up to the user to specify argument types, the intended role of types is to discriminate between 'incompatible' quantities (thereby ultimately constrain the number of constructible features that miss an intuitive rationale). On one hand, a type here does not need to be as specific as to coincide with the notion of a *relational attribute* – for example, if two of the properties (attributes) of cars were their height

[8] As usual in the practice of logic programming, two different predicates may share the predicate symbol, being distinguished only by different arities. We thus do not assign arity directly to a predicate symbol.

[9] For the moment, we are not interested in the recall value 5, here chosen arbitrarily.

[10] Note that for simplicity we do not consider a literal being the negation of the atom $p(v_1 \ldots v_a)$. This does not constrain generality–the semantics of the predicate p/a can be inverted e.g. within its definition in the background knowledge database.

and width, we may want to approve for a feature comparing these two quantities (e.g. `hasCar(T,C)`, `height(C,H)`, `width(C,W)`, `H > W`) and we would thus assign a type such as `linear_size` to both the quantities (assuming there is some declaration for the inequality predicate with this type for both arguments). On the other hand, an argument type will not necessarily be as general as a *data type*. For instance, although the number of wheels and the number of loads in a car are both integer values, we will most likely want to ascribe a distinct type (e.g. `wheelnum`, `loadnum`) to each of them for it is hardly reasonable to compare these quantities, or represent them by a common variable.

We are now heading to define the concept of a feature candidate, which is a correctly built sequence of admissible literals. It is a basic element of the search space traversed by the algorithm when seeking for features: it may not itself constitute a feature but it may be refined towards a feature by adding further admissible literals. A feature candidate has to comply with (a) the given feature grammar, (b) constraints on its syntactical complexity and additionally, (c) its literals have to be ordered in such a way that no other ordering of the same literals is a feature candidate. The efficiency-related motivation for condition (c) is obvious by interpreting a node as a conjunction of logic goals: we do not want to explore two nodes such that one is a permutation of the other's literals.

Let us digress into a small discussion. Alternatively to introducing a required literal order, we might rather proceed to see a feature as a *set* of literals. Our basic requirements on a feature may be formulated as order-independent, e.g. for each literal with some non-key input variable V we would check if another literal, where V is an output, is in the set. An obvious additional workload in this approach would lie in preventing input-output loops in features. Loops are eliminated automatically in the order-employing approach. But more importantly, designing a unique total order and viewing features as series of literals under that order allows for a straightforward recursive implementation (described later in the text) of the feature search.

Note that the ordering itself will be constrained by the feature grammar. Namely, we will stipulate that a literal with a non-key input variable can only appear after a literal where the same variable is an output. The feature grammar thus already imposes a partial order with whom the selected unique total order has to be compatible. Due to this constraint, we split the definition of a feature candidate into parts. First we define a 'search node' satisfying (a), then define what we mean by 'literal ordering' and then at last we define a feature candidate satisfying all of (a), (b) and (c).

Definition 3. (Search node.) Let $G = (\kappa, \Delta)$ be a feature grammar and for some $n \in N$ let $f = (l_1, l_2, \ldots, l_n)$ where each $l_i = p_i(v_{i_1}, \ldots, v_{i_{\mathrm{arity}(i)}})$ is a literal admissible by G. We denote $\mathbf{vars}(f) = \cup_{i=1}^{n} \mathbf{vars}(l)$ and say that f is a *search node* (for G) if

1. (type respecting) for all $v \in \mathbf{vars}(f)$ and $1 \leq i, j \leq n$: $\mathbf{type}_\Delta(l_i, v) = \mathbf{type}_\Delta(l_j, v) \equiv \mathbf{type}_\Delta(f, v)$,
2. (key) there is exactly one $v \in \mathbf{vars}(f) \setminus \cup_{i=1}^{n} \mathbf{outvars}_\Delta(l_i)$, denoted $v = \mathbf{key}_\Delta(f)$, and it holds $\mathbf{type}_\Delta(f, v) = \kappa$,

3. (variable production) for each $v \in \mathbf{vars}(f)$, $v \neq \mathbf{key}_\Delta(f)$: if $\exists j$: $v \in \mathbf{invars}_\Delta(l_j)$ then $\exists\, i, j > i$: $v \in \mathbf{outvars}_\Delta(l_i)$,
4. (new outputs) for each $1 \leq i \leq n$: $\cup_{j=1}^{i-1} \mathbf{vars}(l_j) \cap \mathbf{outvars}(l_i) = \{\}$.

Let $N(G)$ be the set of all search nodes for G. If $f \in N(G)$ then we denote the set of all literals in f as $\mathbf{lits}(f) = \{l_1, l_2, \ldots, l_n\}$. We also say that two literals l_j, l_k in f are *in the same call*, denoted[11] $l_j \sim_f l_k$ if $p_j = p_k$, $arity(j) = arity(k)$, and $v_{j_q} = v_{k_q}$ for each input variable v_{j_q} of l_j. Further we set $Recalls(f) = \{I \mid I \subseteq \{1, 2, \ldots, n\},\ l_j \sim_f l_k\ \forall j, k \in I\}$. Finally we denote $\mathbf{recall}_\Delta(f, l_i) = |I|$ such that $i \in I \in Recalls(f)$ for a literal l_i in f.

We interchange 'search node' with 'node' when there is no risk of confusion. An informally expressed meaning of items 1 to 3 is: "A literal is a part of a search node only if each of its input arguments is of a type equal to the type of some output argument of a preceding literal, or is the key variable. The key variable is the only one 'entering' the feature, i. e. not produced within it."

For example `hasCar(T,C1),long(C1),short(C2)` does not qualify to be a search node.

Assumption 4 (in conjunction with item 2 of Definition 2) guarantees that a new variable will be used in place of each output argument of a literal in a search node. This gives the user the freedom of choosing the types of variables considered for equality checking. For example[12] `NumberOfWheels(C1,N)`, `NumberOfWheels(C2,N)` cannot in principle be a substring of a search node, however, `NumberOfWheels(C1,N1)`, `NumberOfWheels(C2,N2)`, `N1=N2` is a legal substring if the user elects to declare the equality predicate (with two inputs) for the type describing the number of wheels (e. g. integer).

Finally, two literals are said to be in the same call if they are of the same predicate and they share the same variable in each of their inputs. The relation which we denote as $\sim_f$ is clearly an equivalence relation (reflexive, symmetric and transitive) and so the set $Recalls(f)$ consists of equivalence classes disassembling the set $\{1, 2, \ldots, n\}$. Thus for a given $i \in \{1, 2, \ldots, n\}$ there is exactly one I such that $i \in I \in Recalls(f)$ and consequently the value $\mathbf{recall}_\Delta(f, l_i)$ is determined uniquely. We will use this quantity later in the text as a means to introduce a natural bound on the complexity of a feature.

Before we turn attention to defining a desired literal order, let us expose three auxiliary values called *variable depth*, *literal depth* and *node depth*. The node depth will be used as a complexity-constraining parameter, delimiting how 'deep' a feature can go in the structure of an individual.

Definition 4. (Depth.) Let $G = (\kappa, \Delta)$ be a feature grammar and $f = (l_1, l_2 \ldots, l_n) \in N(G)$. The variable $v = \mathbf{key}_\Delta(f)$ is said to be *in depth* 0, denoted $\mathbf{depth}_\Delta(f, v) = 0$. For other variables $v' \in \mathbf{vars}(f)$ we define

$$\mathbf{depth}_\Delta(f, v') = 1 + \max_{w \in \mathbf{invars}_\Delta(l_i)} \mathbf{depth}_\Delta(f, w) \tag{4}$$

[11] For clarity, the dependence of the relation on Δ is implicit in the notation.

[12] We omit the obvious declaration of the exemplified predicate.

where l_i satisfies $v' \in \mathbf{outvars}_\Delta(l_i)$, and for a literal l_j, $1 \leq j \leq n$ we define

$$\mathbf{depth}_\Delta(f, l_j) = \max_{w \in \mathbf{invars}_\Delta(l_i)} \mathbf{depth}_\Delta(f, w) \tag{5}$$

and lastly we define

$$\mathbf{depth}_\Delta(f) = \max_{w \in \mathbf{vars}(f)} \mathbf{depth}_\Delta(f, w). \tag{6}$$

Considering items 3 in Definition 3 and the fact $v' \neq \mathbf{key}_\Delta(f)$, there must be an i such that $v' \in \mathbf{outvars}_\Delta(l_i)$. Further, from item 4 in the same definition it follows that a variable does not appear as an output in more than one literal in f. Therefore l_i is determined uniquely and consequently the values of $\mathbf{depth}_\Delta(f, v')$, $\mathbf{depth}_\Delta(f, l_j)$ and $\mathbf{depth}_\Delta(f)$ are determined uniquely for any v', l_j and f, respectively. As an example, consider the search node `hasCar(T,C), hasShape(C,S1), hasLoad(C,L1), hasShape(L1,S2), similar(S1,S2), hasLoad(C,L2)`. Here, the variables `T, C, S1, L1, S2` and `L2` have successively depths 0, 1, 2, 2, 3 and 2, the literals have (in order of their appearance) depths 0, 1, 1, 2, 3, 1, and the node has depth 3.

We will now formally prove a 'monotonicity' lemma about depths and recalls, which is rather obvious intuitively, but quite important for later theorems influencing the procedural design of the feature construction.

Lemma 1. Let $G = (\kappa, \Delta)$ be a feature grammar, $f = (l_1, l_2 \ldots, l_n) \in N(G)$ and $f' = (l_1, l_2 \ldots, l_m) \in N(G)$, $m < n$. Then it holds $\mathbf{depth}_\Delta(f) \geq \mathbf{depth}_\Delta(f')$ and for each literal l_i in f' it holds $\mathbf{recall}_\Delta(f, l_i) \geq \mathbf{recall}_\Delta(f', l_i)$.

Proof. It clearly suffices to prove the inductive step, i.e. to show that both assertions hold if $f' = (l_1, l_2 \ldots, l_{n-1})$. We will treat both assertions successively.

Regarding the depth inequality, clearly $\mathbf{vars}(f') \subseteq \mathbf{vars}(f)$. Considering equation (6), it suffices to see that all variables in f' have the same depth in f and f'. Due to condition 4 in Definition 3 and the fact that f' is a search node, for every variable v' occurring in f' there is exactly one literal l_i in $l_1, l_2 \ldots, l_{n-1}$ such that $v' \in \mathbf{outvars}_\Delta(l_i)$. If $l_1, l_2 \ldots, l_n$ is also a search node and $v' \in \mathbf{outvars}_\Delta(l_i)$ $(i < n)$ then $v' \notin \mathbf{outvars}_\Delta(l_n)$. It holds (equation (4)) that $\mathbf{depth}_\Delta(f, v') = 1 + \max_{w \in \mathbf{invars}_\Delta(l_i)} \mathbf{depth}_\Delta(f, w)$ where $v' \in \mathbf{outvars}_\Delta(l_i)$. Since $l_i \neq l_n$, the depth of v' does not depend on l_n. Since we have shown that the depth of an arbitrary variable v' in f' does not depend on l_n, the depth of v' is thus equal in both f' and f. Consequently, due to equation (6), the depth of f is equal or larger than that of f'.

The recall inequality is rather simple to show. Note that $\sim_f$ is an equivalence relation on $\{1, 2, \ldots, n\}$ (see Definition 3 and the attached comment). Thus clearly for each $1 \leq i \leq n-1$ if $l_i \sim_f l_n$ then $\mathbf{recall}_\Delta(f', l_i) < \mathbf{recall}_\Delta(f, l_i)$, otherwise $\mathbf{recall}_\Delta(f', l_i) = \mathbf{recall}_\Delta(f, l_i)$. □

Let us now postulate what we require from a literal-ordering operator O, which will help to prevent a repetitive inspection of search nodes with the same logic

meaning during the search space exploration. Rather than viewing the order as a relation on literals, it will be more convenient to define it as a function on nodes. Informally, the operation of O will be as follows: a search node n will be explored (and refined) only if $O[n] = n$.

Definition 5. (Literal order.) Let $G = (\kappa, \Delta)$ be a feature grammar. A *literal order* for G is a function $O : N(G) \to N(G)$ such that for all $f, f' \in N(G)$, $f = (l_1, l_2, \ldots, l_n)$, $f' = (l'_1, l'_2, \ldots, l'_n)$

1. (permutation) if $O[f] = o$ then $\mathbf{lits}(f) = \mathbf{lits}(o)$,
2. (monotonicity) if $O[f] = (o_1, o_2, \ldots, o_n)$ and $(l_1, l_2, \ldots, l_{n-1}) \in N(G)$ then $O[(l_1, l_2, \ldots, l_{n-1})] = (o_1, o_2, \ldots, o_{n-1})$,
3. (unification) if $\mathbf{lits}(f) = \mathbf{lits}(f')$ then $O[f] = O[f']$.

It is important to realize that O is a function, i. e. it must order any search node. The first item in the definition merely guarantees that O does not remove or add any literal from/to the ordered search node. The importance of the second item will be illuminated in the context of a further exposed theorem. The last condition's role is obvious.

It is straightforward from the respective definitions that the depth of a variable and the recall value of a literal in a search node do not depend on the order of literals in the search node. We thus have an immediate corollary of Lemma 1.

Corollary 1. Let $G = (\kappa, \Delta)$ be a feature grammar and $f, f' \in N(G)$, such that $\mathbf{lits}(f') \subseteq \mathbf{lits}(f)$. Then it holds $\mathbf{depth}_\Delta(f) \geq \mathbf{depth}_\Delta(f')$ and for each literal l_i in f' it holds $\mathbf{recall}_\Delta(f, l_i) \geq \mathbf{recall}_\Delta(f', l_i)$.

Unfortunately, the space alloted does not allow us to analyze formally our design of the specific literal-ordering operator as implemented in EFFEDRIN. However, a few technical remarks are in order. First of all, note that the ordering operator has to comply with the partial order induced by Definition 3 (see the discussion preceding that definition). This means that some permutations of a search node do not produce a search node, therefore applying a simple total (e. g. alphabetical) order on literals may cause $O[f] \notin N(G)$. For example, while `hasCar(T,C), awesome(C)` is a search node, `awesome(C), hasCar(T,C)` is not. A principal proviso to make O comply with the assumptions is that $O[f] = o_1, o_2, \ldots, o_n$ for any $f \in N(G)$ must satisfy the implication

$$o_i \prec_+ o_j \to i < j \tag{7}$$

where $\prec_+$ is the transitive closure of $\prec$ such that $o_i \prec o_j$ whenever $\mathbf{invars}_\Delta(f, o_j) \cap \mathbf{outvars}_\Delta(f, o_i) \neq \{\}$. However, in general there are clearly still many possible literal-orderings O satisfying this condition. To systemize them, each search node

$f = l_1, l_2, \ldots l_n$ is viewed as an undirected graph[13] γ_f having a node for each l_i and a link between l_i and l_j whenever

$$l_j = \max_{<_A}\{l_k \mid l_k \prec l_i,\ \mathbf{depth}(f, l_k) = \mathbf{depth}(f, l_i) - 1\} \tag{8}$$

where $\max_{<_A}$ is the maximum with respect to an arbitrary fixed total order (e. g. alphabetical) $<_A$ on atoms. It can be shown that γ_f is connected and acyclic and therefore is a tree. Further, its nodes can be enumerated by a standard systematic tree exploration procedure (such as breadth-first or depth-first search) starting in the node $l_1 = o_1$ (chosen as the 'root' as there is clearly no l_i s.t. $l_i \prec_+ l_1$). As long as children of any node are explored in the order dictated by $<_A$, equation (8) guarantees that the node order resulting from such an enumeration satisfies condition 7 above and thus $O[f] \in N(G)$. It is trivial to check that conditions 1 and 3 of Definition 5 hold as well for this order. It is a little more technical to show that 2 is also valid, the proof (which we skip for lack of space) namely considers that due to condition 4 in Definition 3, no output variable of l_n occurs in $l_1, \ldots, l_{n-1}$.

Let us provide an illustrative example. A breadth-first exploration of γ_f for a node f may yield the node

```
hasCar(T,C), short(C), hasLoad(C, L1), hasLoad(C,L2), small(L1),
                    round(L2), notSame(L1,L2)
```

whereas a depth-first exploration for the same node would yield[14]

```
hasCar(T,C), short(C), hasLoad(C, L1), small(L1), hasLoad(C,L2),
                    round(L2), notSame(L1,L2).
```

The breadth-first node ordering basically corresponds to the 'layer' structure of *bottom clauses* generated in some state-of-art ILP systems[15] (such as Progol [19] or Aleph). However, we shall see later in this paper that the depth-first version of ordering will be more convenient for our purposes.

Having both the notions of order and depth at hand, we can proceed to define a feature candidate.

Definition 6. (Feature candidate.) Let $G = (\kappa, \Delta)$ be a feature grammar, O a literal order for G. Let further $D, L \in N$ and $f = (l_1, l_2, \ldots, l_n) \in N(G)$. Then f is said to be a *feature candidate* admissible by (G, O, D) and L if

1. (order respecting) $O[f] = f$,
2. (recall respecting) $\mathbf{recall}_\Delta(f, l_i) \leq \mathbf{recall}_\Delta(l_i)$ for each literal l_i in f,

[13] This graph should not be confused with the graph induced later by a *refinement* operator which determines links *between* search nodes.

[14] It should be clear that `notSame(L1,L2)` is connected in γ_f by an edge only with `hasLoad(C,L2)`, not e. g. with `hasLoad(C, L1)`. The two former literals lie both in depth 1, and `hasLoad(C, L1)` $<_A$ `hasLoad(C, L2)`.

[15] And in fact it alone can be implemented with less computational expenditures than the more general approach described hereby.

3. (length respecting) $n \leq L$,
4. (depth respecting) $\mathbf{depth}_{\Delta}(f) \leq D$.

We denote $N_{O,D,L}(G)$ the set of all such nodes.

The first assumption installs the unique-order requirement, whose underlying ideas we described in the discussion preceding Definition 3 and another one connected to Definition 5. Assumption 2 is based on the declared recall value of a literal $\mathbf{recall}_{\Delta}(l_i)$ (see Definition 2) and the computed recall value $\mathbf{recall}_{\Delta}(f, l_i)$ (see Definition 3). This assumption[16] allows the user to incorporate a rather natural bound on the number of immediate substructures of one structure that can be addressed within a feature. For example, with declarations containing `[2,hasCar(+train,-car)]` and `[1,hasLoad(+car,-load)]`, the string

```
hasCar(T,C), hasLoad(C,L1), hasLoad(C,L2), ...
```

could not appear in a search node since it refers to two loads of one car (whether or not `L1 = L2`), although the following could: `hasCar(T,C1), hasLoad(C1,L1), hasCar(T,C2), hasLoad(C2,L2), ...` The last two assumptions in Definition 6 implement straightforward constraints on the complexity of a feature. As follows from the comment preceding Definition 4, the D parameter sets the maximum depth in an object structural description that is addressable by a feature candidate (and therefore any admissible feature). For example, if $D = 2$, then a feature can regard a car of a train, a load of a car, but not any substructures of a load, were there any. The L parameter, simply constraining the feature description length, is crucial. This will follow from an algorithmic complexity discussion provided later in the paper.

The following theorem will clarify the 'procedural' utility of a feature candidate.

Theorem 1. Let G be a feature grammar, O an order for G and $D, L \in N$. If $f = (l_1, l_2, \ldots, l_n) \in N_{O,D,L}(G)$ and $n > 1$ then $(l_1, l_2, \ldots, l_m) \in N_{O,D,L}(G)$ for any $1 \leq m < n$.

Proof. Assuming $f \in N_{O,D,L}(G)$ for $n > 1$, to prove the theorem we need to prove the inductive step, i.e. that $(l_1, l_2, \ldots, l_n) \in N_{O,D,L}(G)$ implies $f' = (l_1, l_2, \ldots, l_{n-1}) \in N_{O,D,L}(G)$. This will be proved by contradiction. Consider $f' \notin N_{O,D,L}(G)$. Then either $f' \in N(G)$ or $f' \notin N(G)$. We treat the two cases separately.

If $f' \in N(G)$, then necessarily one or more of the conditions 1–4 of Definition 6 does not hold for f'. But condition 1 holds by the assumption 2 of Definition 5, conditions 2 and 4 hold due to Lemma 1 and condition 3 holds trivially since $n - 1 < n$. Since all conditions 1–4 hold for f' while $f' \notin N_{O,D,L}(G)$, it must be that $f' \notin N(G)$, i.e. we have a contradiction.

Now consider the case when $f' \notin N(G)$. Since all literals in f are admissible then so must be all literals in f'. Thus one or more conditions 1–4 of Definition 3 must be invalid. It is straightforward to check that conditions 1, 3 and 4 hold for f' provided they hold for f, and we leave it to the reader. It remains to check if condition 2 is

[16] inspired by a similar constraint used in the ILP systems Progol [19] and Aleph

satisfied for f'. Given that this condition holds for f, it can be invalidated when removing l_n only if l_n is the only literal containing the variable $\mathbf{key}_\Delta(f)$, since clearly $\mathbf{vars}(f') \subseteq \mathbf{vars}(f)$. But this variable is also contained in l_1: if it were not, l_1 would either have no input variable, thus harming condition 2 of Definition 1 (and failing to be an admissible literal) or at least one input variable, thus violating condition 3 od Definition 3. Since we assumed $n > 1$, it holds $l_n \neq l_1$ and therefore l_n cannot be the only literal containing $\mathbf{key}_\Delta(f)$ and condition 2 is valid for f'. Since all conditions 1 4 of Definition 3 are then valid, it holds $f' \in N(G)$ which contradicts with the assumption of this paragraph. □

We have thus seen an important property of a feature candidate: any prefix thereof is a feature candidate. In other words, adding literals to any literal sequence that is not a feature candidate will never produce a feature candidate. Since a feature, as we shall define in a moment, will itself have to be a feature candidate and we will construct features by successively adding literals, we will abandon any 'non-candidate' literal sequences in this process. Let us now formalize the step of adding literals.

Definition 7. (Refinement.) Let G be a feature grammar, O a literal order for G, and $D, L \in N$. Let $N_{\prec_V}$ be a subset of $N_{O,D,L}(G)$ such that for each $f = (l_1, l_2, \ldots, l_n) \in N_{\prec_V}$ and each i, $1 \leq i \leq n$ it holds

$$\min_{\prec_V}[\mathbf{outvars}(l_i)] = s_{\prec_V}(v) \tag{9}$$

where

$$v = \max_{\prec_V}[\cup_{j=1}^{i-1} \mathbf{vars}(l_j) \cup \mathbf{invars}_\Delta(l_i)]. \tag{10}$$

The *refinement* for (G, O, D, L) is the function $\mathbf{ref}_{(G,O,D,L)} : \{Empty\} \cup N_{\prec_V} \to 2^{N_{\prec_V}}$ such that[17] $\mathbf{ref}[Empty] = \{f \in N_{\prec_V}; |\mathbf{lits}(f)| = 1\}$ and $\mathbf{ref}[(l_1, l_2, \ldots, l_n)] = \{f \in N_{\prec_V}; f = (l_1, l_2, \ldots, l_n, l_{n+1})\}$. A *refinement closure* is a function $\mathbf{ref}^+ : \{Empty\} \cup N_{\prec_V} \to 2^{N_{\prec_V}}$ such that it holds $\mathbf{ref}^+[Empty] = N_{\prec_V}$ and $\mathbf{ref}^+[(l_1, l_2, \ldots, l_n)] = \{f \in N_{\prec_V}; n < m \leq L, f = (l_1, l_2, \ldots, l_m)\}$.

The refinement provides all feature candidates that are one-literal extensions of a feature candidate, so that the first newly introduced variable (which must clearly be an output of the added literal) is the successor in the variable order $\prec_V$ of the 'maximum variable' (with respect to this order) found in the refined feature candidate. This condition (along with condition 2 of Definition 2) prevents obtaining multiple feature candidates differring only in variable naming. Also note that $\mathbf{ref}^+(f) = \cup_n F_n$ $(1 \leq n \leq L - m)$ where $F_1 = \mathbf{ref}(f)$ and $F_{n+1} = \{f' \in \mathbf{ref}(f''); f'' \in F_n\}$, for any $f = (l_1, l_2, \ldots, l_m) \in N_{O,D,L}(G)$.

Some of the conditions a feature will have to comply with are determined by the quantity of instances for which the feature holds. Whether a feature holds for an

[17] Henceforth we shall dismiss the parameters (G, O, D, L) in the subscript when their instantiation is obvious from the context.

instance means that, in the first-order logic interpretation, it is more general than the instance description (such as the antecedent in Example 1). In this paper we abstract from technicalities inherent to verifying this, rather we merely reflect three properties of a function yielding the subset of instances ('examples') for which a feature holds.

Definition 8. (Coverage.) A *coverage* is a function $\mathbf{cov} : A \times 2^{Ex} \to 2^{Ex}$ (where A is the set of all finite sequences of atoms) if for any $G \in \Gamma$, $E \subseteq Ex$, $D, L \in N$, any literal order O for G and all $f, f', f'' \in N_{O,D,L}(G)$ it holds:

1. (contraction) $\mathbf{cov}(f, E) \subseteq E$,
2. (disconnected conjunction) if $\mathbf{lits}(f'') = \mathbf{lits}(f') \cup \mathbf{lits}(f)$ and $\mathbf{vars}(f') \cap \mathbf{vars}(f) \subseteq \{\mathbf{key}_{\Delta}(f)\}$ then $\mathbf{cov}(f'', E) = \mathbf{cov}(f', E) \cap \mathbf{cov}(f, E)$,
3. (subsumption) if $f' \in \mathbf{ref}(f)$ then $\mathbf{cov}(f', E) \subseteq \mathbf{cov}(f, E)$,

where **ref** is the refinement for (G, O, D, L).

As we are nevertheless concerned with the efficiency of the explained method, we should address the question of whether the above function can be computed efficiently. The issue of efficiently proving whether a Prolog (Datalog[18]) clause entails a fact with respect to a background knowledge has been under investigation in both the field of logic programming and ILP (e. g. [9]). Not imposing constraints on the background knowledge database, this problem is in general undecidable, but can be solved approximately. A naive, but rather tolerable technique guaranteeing an efficient approximation of logic entailment is a step-bounded resolution procedure [19] (know also as *h-easy* resolution), conducted in several state-of-art ILP systems. Among more sophisticated methods falls 'stochastic matching' [22] exploiting the correspondence between first-order logic subsumption (in this case constrained to the frame of Datalog) and the generally NP-complete constraint-satisfiability problem, and existing fast probabilistic algorithms applicable on the latter form. Also, based on the same problem correspondence and research of the phase transition phenomenon in constraint satisfiability, deterministic methods have been suggested to speed up subsumption check [18]. Any of such proving techniques could in principle be incorporated into the feature construction method. However, in our previous experimentation the dimensionality of the *feature space* has consistently been a dominant tractability factor, diminishing the efficiency aspects of the inherent *proving*. We nonetheless do not exclude a future augmentation of the currently implemented proving procedure e. g. for problems characterized by a large number of data instances.

Let us now proceed to define the concept of a feature.

[18] Prolog without functions of arity greater than 0. All example features in this paper are proper Datalog clauses.

Definition 9. (Admissible feature.) Let C be a tuple $C = (G, O, D, M, L)$ where $G = (\kappa, \Delta)$ is a feature grammar with a literal order O and $D, M, L \in N$. Then C is called a *feature constraint.* Let **cov** be a coverage and let $E \subseteq Ex$. Finally, let $f = (l_1, l_2, \ldots, l_n) \in N_{O,D,L}(G)$. We say that f is a *feature admissible with* **cov** *on* E *by* C if

1. (variable consumption) if $\exists i$: $v \in \mathbf{outvars}_\Delta(l_i)$ then $\exists j$, $v \in \mathbf{invars}_\Delta(l_j)$,
2. (non-triviality) $\mathbf{cov}(f, E) \subset E$.
3. (relevance) $|\mathbf{cov}(f, E)| \geq M$,
4. (undecomposability) there are no non-empty $I, J \subseteq \{1, 2, \ldots, n\}$ such that $I = \{1, 2, \ldots, n\} \setminus J$ and $[\cup_{i \in I} \mathbf{vars}(l_i)] \cap [\cup_{j \in J} \mathbf{vars}(l_j)] \subseteq \{\mathbf{key}_\Delta(f)\}$.

Condition 1 implements a principle suggested by [14] that all output variables should be 'consumed' within the feature. Note that if f is a feature, this condition automatically implies $j > i$, otherwise f would violate one of the conditions 3, 4 in Definition 3, thereby not being a search node and contradicting the assumption of the definition above. Assumptions 2 and 3 reflect a standard constraint used in general feature-selection approaches to avoid features that are either unusable for data discrimination, or too special (their scope is too limited). In the latter case, such features are avoided in the effort of preventing the effect of data overfitting. Assumption 4 is motivated clearly: if it did not hold, it would be possible to express the feature as a conjunction of two or more features, which we will formally prove in prepared continuation of the paper. We do not care for such decomposable features, since AVL systems are themselves typically able to construct conjunctive formulas from feature identifiers. A decomposable feature is thus redundant. Besides, we will present several theorems enabling for an often massive search space pruning accompanied with a dramatic speed-up of the feature construction. Some of them largely exploit the notion of undecomposability by pruning search subspaces surely containing only decomposable features.

ACKNOWLEDGEMENT

The authors are grateful to Jaroslav Pokorný (MFF UK, Prague) for his unreserved criticism of a previous version of this paper, which led to the discovery of several flaws thereof. The author is supported by the US Department of Defense through the DARPA grant F30602-01-2-0571 and by the Ministry of Education, Youth and Sports of the Czech Republic through the grant MSM 212300013. Part of the method described in the paper was developed while Filip Železný was on leave at the Institute Josef Stefan in Ljubljana, working with Nada Lavrač.

(Received November 21, 2003.)

REFERENCES

[1] R. Agrawal and R. Srikant: Fast algorithms for mining association rules. In: Proc. 20th Internat. Conference Very Large Data Bases, VLDB, Morgan Kaufmann, San Francisco, CA 1994 pp. 487–499.

[2] E. Alphonse and C. Rouveirol: Lazy propositionalization for relational learning. In: Proc. 14th European Conference on Artificial Intelligence (ECAI'2000) (W. Horn, ed.), IOS Press 2000, pp. 256–260.

[3] J. Blatăk and L. Popelínský: Feature construction with RAP. In: Proc. of the Work-in-Progress Track at the 13th Internat. Conference on Inductive Logic Programming, University of Szeged 2003.

[4] P. Clark and T. Niblett: The cn2 induction algorithm. Mach. Learning *3* (1989), 261–283.

[5] S. Džeroski: Numerical Constraints and Learnability in Inductive Logic Programming. Ph.D. Thesis. Faculty of Electrical Engineering and Computer Science, University of Ljubljana 1995.

[6] S. Džeroski and N. Lavrač, eds.: Relational Data Mining. Springer–Verlag, Berlin 2001.

[7] W. Emde and D. Wettschereck: Relational instance based learning. In: Machine Learning – Proc. 13th Internat. Conference on Machine Learning, Morgan Kaufmann, San Francisco, CA 1996, pp. 122–130.

[8] P. Hájek: Mechanizing Hypothesis Formation. Springer–Verlag, Berlin 1966.

[9] J. U. Kietz: Some lower bounds for the computational complexity of inductive logic programming. In: Machine Learning: ECML-93, Proceedings of the European Conference on Machine Learning, volume 667, Springer–Verlag, Berlin 1993, pp. 115–123.

[10] A. J. Knobbe, M. de Haas, and A. Siebes: Propositionalisation and aggregates. In: Proc. Fifth European Conference on Principles of Data Mining and Knowledge Discovery (PKDD), Springer–Verlag, Berlin 2001.

[11] S. Kramer, N. Lavrač, and P. A. Flach: Propositionalization Approaches to relational data mining. In: Relational Data Mining (N. Lavrač and S. Džeroski, eds.), Springer–Verlag, Berlin 2001.

[12] M. A. Krogel, S. Rawles, F. Železný, P. A. Flach, N. Lavrač, and S. Wrobel: Comparative evaluation of approaches to propositionalization. In: Proc. 13th Internat. Conference on Inductive Logic Programming, Springer–Verlag, Berlin 2003.

[13] M. A. Krogel and S. Wrobel: Transformation-based learning using multirelational aggregation. In: Proc. 11th Internat. Conference on Inductive Logic Programming (ILP), Springer–Verlag, Berlin 2001, pp. 142–155.

[14] N. Lavrač and P. A. Flach: An extended transformation approach to inductive logic programming. ACM Trans. Comput. Logic *2* (2001), 4, 458–494.

[15] N. Lavrač and S. Džeroski: Inductive Logic Programming: Techniques and Applications. Ellis Horwood, 1993.

[16] N. Lavrač, F. Železný, and P. A. Flach: RSD: Relational subgroup discovery through first-order feature construction. In: Proc. 12th Internat. Conference on Inductive Logic Programming (ILP), Springer–Verlag, Berlin 2002.

[17] H. Liu and H. Motoda: Feature Selection for Knowledge Discovery and Data Mining. Kluwer, Dordrecht 1998.

[18] J. Maloberti and M. Sebag: Theta-subsumption in a constraint satisfaction perspective. In: Proc. 11th Internat. Conference on Inductive Logic Programming (ILP) (Lectures Notes in Artificial Intelligence 2157), Springer–Verlag, Berlin 2001, pp. 164–178.

[19] S. Muggleton: Inverse entailment and Progol. New Generation Computing, Special Issue on Inductive Logic Programming *13* (1995), 3–4, 245–286.

[20] B. Pfahringer and G. Holmes: Propositionalization through stochastic discrimination. In: Proc. of the Work-in-Progress Track at the 13th Internat. Conference on Inductive Logic Programming, University of Szeged 2003.
[21] J. Ross Quinlan: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Francisco, CA 1992.
[22] M. Sebag and C. Rouveirol: Tractable induction and classification in first-order logic via stochastic matching. In: Proc. 15th Internat. Joint Conference on Artificial Intelligence, Morgan Kaufmann, San Francisco, CA 1997, pp. 888–893.
[23] A. Srinivasan, S. H. Muggleton, M. J. E. Sternberg, and R. D. King: Theories for mutagenicity: a study in first-order and feature-based induction. Artificial Intelligence *85* (1996), 1, 2, 277 299.
[24] O. Štěpánková, P. Aubrecht, Z. Kouba, and P. Mikšovský: Preprocessing for Data Mining and Decision Support Data Mining and Decision Support: Integration and Collaboration. Kluwer, Dordrecht 2003.
[25] I. H. Witten, E. Frank, L. Trigg, M. Hall, G. Holmes, and Sally Jo Cunningham: Weka: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann, San Francisco, CA 1999.
[26] J. D. Zucker and J. G. Ganascia: Representation changes for efficient learning in structural domains. In: Internat. Conference on Machine Learning 1996, pp. 543 551.
[27] F. Železný, N. Lavrač, and S. Džeroski: Constraint-based relational subgroup discovery. In: Proc. Multi-Relational Data Mining Workshop at KDD 2003, Washington 2003.

Filip Železný, Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, Technická 2, 166 27 Praha 6, Czech Republic, and University of Wisconsin Medical School, Department of Biostatistics, Medical Science Center, 1300 University Avenue, Madison, WI 53706, U.S.A.
e-mails: zelezny@biostat.wisc.edu, zelezny@fel.cvut.cz