Extension of the Top-Down Data-Driven Strategy to ILP

Erick Alphonse and Céline Rouveirol

LIPN-CNRS UMR 7030, Université Paris 13, France
{alphonse,rouveirol}@lipn.univ-paris13.fr

Abstract. Several upgrades of Attribute-Value learning to Inductive Logic Programming have been proposed and used successfully. However, the Top-Down Data-Driven strategy, popularised by the AQ family, has not yet been transferred to ILP: if the idea of reducing the hypothesis space by covering a seed example is utilised with systems like PROGOL, Aleph or MIO, these systems do not benefit from the associated data-driven specialisation operator. This operator is given an incorrect hypothesis h and a covered negative example e and outputs a set of hypotheses more specific than h and correct wrt e. This refinement operator is very valuable considering heuristic search problems ILP systems may encounter when crossing plateaus in relational search spaces. In this paper, we present the data-driven strategy of AQ, in terms of a *lgg*-based change of representation of negative examples given a positive *seed* example, and show how it can be extended to ILP. We evaluate a basic implementation of AQ in the system Propal on a number of benchmark ILP datasets.

1 Introduction

In Inductive Logic Programming (ILP), various learning strategies from Attribute-Value (AV) learning have been adapted: to name a few, top-down induction of decision trees in the TILDE system [4], top-down induction of rules in the systems FOIL [28], PROGOL [24], Aleph [35] and MIO [26]. Bottom-up data-driven algorithms, based on the *least-general-generalisation* (*lgg*) operator (also known as *most-specific generalisation*) have also been implemented (see [17] for the main results). However, the Top-down Data-Driven (TDD) strategy has very few incarnations and is not used in ILP. Its emblem is the family of AV systems AQ [22]. The search is top-down in the space of hypotheses more general than or equal to a particular example which is named in this context a *seed example*. If the idea of reducing the hypothesis space by covering a seed example is utilised with systems like PROGOL, Aleph or MIO, these systems do not benefit from the associated TDD operator. They address the learning problem within the generate-and-test paradigm (computing refinements based on the structure of the search space only) : they have to deal with many refinements, for a given hypothesis, that are not relevant with respect to the discrimination task. The TDD operator is the dual of the *lgg* operator in the sense that, given an incorrect hypothesis h and a covered negative example e, it outputs a set of hypotheses more specific than h and correct with respect to e. This refinement operator is described in [22] as a set of *extension-against* rules for computing refinements of boolean attributes, numerical attributes, nominal as well as hierarchical ones. For example, a rule for using a boolean attribute *att* for refining an incorrect hypothesis is:

If $att = val$ in the seed and $att \neq val$ in a covered negative example then $att = val$ is a valid refinement

Relying on the training set allows a TDD strategy to have a branching factor which is necessarily smaller than or equal to the branching factor of a generate-and-test strategy searching in the same hypothesis space. This makes this strategy very appealing for ILP which is known to be prone to important plateau phenomena in heuristic search (see e.g. [13,2]). As a special case and as advocated by Winston [39] (see also [34]), a TDD learning algorithm can take advantage of negative examples that differ from positive examples by only one attribute, the so-called *near-misses*, to reduce the branching factor to 1 during the heuristic search. Ultimately, a TDD algorithm learning from a dataset provided with all near-misses of the target concept would converge to the concept without search, generating only one refinement each step.

In the rest of the paper, we present the TDD strategy of the AQ system in terms of a *lgg*-based change of representation of negative examples given a positive *seed* example. After applying this representation change, the instance space and the hypothesis space are merged into a simpler hypothesis space, and learning can rely on an algebraic formalisation of AQ's extension against rules. The second contribution of the paper concerns the implementation of the TDD in relational languages as complex as Datalog with negation and constrained variables, which is complete for OI-subsumption [1]. We propose a formalisation of the problem of computing the set of "nearest-miss" *lggs* between two relational examples, which is at the core of the TDD strategy, as a Weighted Constraint Satisfaction Problem [8]. In the last part of the paper, we present an implementation of the basic AQ strategy as given by Clark and Niblett [5] (often referred as AQR) in PROPAL and we evaluate it on a number of ILP benchmark datasets. Although this version of PROPAL does not include any noise-handling mechanism, its performance is quite competitive with respect to state of the art ILP generate-and-test systems.

2 Change of Representation of Learning Data in the TDD Strategy

2.1 Top-Down Data Driven Strategy

The AQ system [22] is a top-down covering learning algorithm. AQ's outer loop is a classical covering algorithm that iterates while some positive examples are still uncovered. Its inner-loop randomly selects an uncovered positive example, the *seed example*, denoted as s in the rest of the paper. AQ then performs a top-down data driven beam search to build a set of maximal and correct generalisations of s, given the set of negative examples. For a given negative example e^-, if a candidate hypothesis h in the beam covers e^-, h is minimally specialised in order to reject it, while still covering s. Throughout the paper, E^+ and E^- denote the set of positive and negative examples of the learning problem, $\mathcal{L}_h$ denotes the hypothesis space, $\succeq$ the coverage relation between a hypothesis of $\mathcal{L}_h$ and an example, $\geq_h$ the partial order between hypotheses of $\mathcal{L}_h$ (generality relationship). $\mathcal{L}_s \subseteq \mathcal{L}_h$ is the space of generalisations of the seed s. The TDD operator can be formally defined as follows.

Definition 1 (TDD operator). *Let* $s \in E^+, h \in \mathcal{L}_s, e^- \in E^-, \rho_s(h, e^-) = \{h' \in \mathcal{L}_s \mid h \geq_h h' \text{ and } h' \not\succeq e^-\}$

This operator can be seen as the dual of the *lgg* operator [27]: given an incorrect hypothesis h and a covered negative example e^-, it outputs a set of maximally general hypotheses more specific than h and correct with respect to e^-, with the additional constraint that each of these specialisations of h should still cover s.

The fact that each minimal specialisation of h should cover a seed example amounts to map the initial search space of the learning algorithm onto the space of generalisations of the seed example. Looking for a hypothesis of $\mathcal{L}_h$ that both covers s and rejects e^- can be equivalently performed by looking for a generalisation of s that rejects $lgg(s, e^-)$. By definition of the *lgg* [27], we have $h \succeq s \wedge h \succeq e^- \Leftrightarrow h \geq_h lgg(s, e^-)$. Equivalently, by contraposition, we have $h \not\succeq s \vee h \not\succeq e^- \Leftrightarrow h \not\geq_h lgg(s, e^-)$. As the TDD strategy is biased towards generating hypotheses that cover s, $h \not\succeq e^- \Leftrightarrow h \not\geq_h lgg(s, e^-)$.

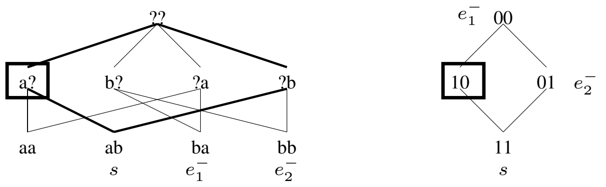

Fig. 1. Bias of $\mathcal{L}_h$ towards the covering of a positive example

This *lgg*-based representation change transforms the initial learning problem $(E, \succeq, \geq_h, \mathcal{L}_h)$ into a new learning problem $(E_s, \geq_h, \geq_h, \mathcal{L}_s)$, E_s being the new set of examples where each example $e \in E$ is reformulated into $lgg(s, e)$. In this new problem, the instance space and the hypothesis space are merged, as illustrated in figure 1 for a simple AV learning problem. The leftmost part of figure 1 shows three training instances, described in terms of two AV attributes. In the initial search space, each of the involved attributes has domain $\{a, b, ?\}$, where $'?'$ denotes any value in the domain, meaning this attribute should be dropped from the hypothesis. The three initial examples s, e_1^- and e_2^- are each mapped in the new search space $\mathcal{L}_s$ (right part of figure 1), here the power-set of the seed example. By definition, s is mapped to the lower bound of $\mathcal{L}_s$, e_1^- is mapped to the top node of $\mathcal{L}_s$ $'00'$ ($lgg(s, e_1^-) =' ??' \geq_h\ 'ab'$) and e_2^- is mapped to $'01'$ ($lgg(s, e_2^-) =' ?b' \geq_h\ 'ab'$).

This reformulation is interesting, because it shows that the so-called "extension-against" rules correspond to an algebraic resolution of the learning problem in a boolean lattice[1] and have broader applications than attribute-value learning as long as the generalisation space of the seed is isomorphic to a boolean lattice as shown in figure 1. In the rest of the paper, we will refer to the TDD refinement operator instead of the "extension-against" rules to point that we take into account the *lggs* of the negative examples directly in the generalisation space of the seed. It is therefore possible to reformulate the specialisation step of AQ as shown in the algorithm of figure 2.

```
FindBestRule($s$,$E^-$,$E^+$)
  $G := \{\top\}$ % top element of $\mathcal{L}_h$
  $BestRule := \emptyset$
  While $G \neq \emptyset$
    $G' := \emptyset$
    For each $g \in G$
      $G := G \setminus g$
      If $g$ is correct and $score(g, E^+) \geq score(BestRule, E^+)$ Then
        $BestRule := g$
      Else
        % computation of the nearest-miss
        $NM := g$ % by definition $g \geq_h NM$
        For each $e^- \in E^-$
          If $lgg(s, e^-)$ nearer-miss than $NM$ Then $NM := lgg(s, e^-)$
        $G' := G' \cup \rho_s(g, NM)$ % specialisation using the TDD refinement operator
    $G := k$ best hypotheses from $G'$ % beam search
  Return $BestRule$
```

Fig. 2. AQ's specialisation loop algorithm for a given seed s

In order to make the specialisation step efficient, the algorithm makes the most of the partial ordering of negative examples to handle most informative negative examples only. First of all, only most specific negative examples in E_s are useful: in the toy example of figure 1, e_2^- is more specific than e_1^- so rejecting e_2^- also rejects e_1^-. In this simple example, there is only one candidate solution obtained by applying once the TDD operator: hypothesis $'10'$ in the boolean space, corresponding to the hypothesis $'a?'$ in the initial search space. If several most specific negative examples are available at that step, which are incomparable by definition, we give preference to the one that is closer to s than

[1] The "extension-against" rules are actually more general and can consider distributive lattices as the product of a boolean lattice with interval lattices and chains whenever numerical and hierarchical attributes are involved [1]. In the present work, we use the product of boolean and interval lattices but in this section, we only discuss the logical part.

any other negative examples as it yields the smallest branching factor for ρ_s (see e.g. [34,5]). We name it the *nearest-miss* and we define it as a most specific element with respect to a total pre-order named *nearer-miss*. It is reflexive, transitive, total, but not antisymmetric.

Definition 2 (nearer-miss). *Let $s \in \mathcal{L}_h$ be the seed example, $x, y \in \mathcal{L}_s$, the distance $d(s, y)$ be the number of attributes' values that differ between s and y. x is nearer-miss than y iff $(x \leq_h y) \vee (y \not\leq_h x \wedge d(s,x) \leq d(s,y))$. It is a total pre-order on the elements of $\mathcal{L}_s$. A least element (most specific) with respect to this total pre-order is a nearest-miss.*

The nearer-miss pre-order induces an equivalence relation between elements of $\mathcal{L}_s$ which are incomparable under $\geq_h$ and at the same distance from the seed. Note that we define the nearer-miss relation as a linear extension of the partial order $\geq_h$ to deal with redundancy in E_s, such that the nearest-miss is necessarily a most specific element of E_s wrt $\geq_h$. This is necessary when we deal with numerical attributes. For example, if we have only one numerical attribute a and a seed $a = 1$ and two negative examples $a = 2$ and $a = 3$ reformulated as $a \in [1, 2]$ and $a \in [1, 3]$, although both negative examples are at a distance of 1 from the seed, $a \in [1, 2]$ is nearer-miss than $a \in [1, 3]$ as it is more specific and its rejection will reject the other negative example.

On figure 1, it can be seen that the maximal branching factor of a top-down generate-and-test operator is 4 without the seed bias, and 2 when only considering specialisations covering s. The branching factor of the TDD operator is 1, as the *lgg* of e_2^- with s is actually a Winston's near-miss. Note that even in the worst case (only far-misses are provided, i.e. negative examples that maximally differ from the seed example), the branching factor of the TDD operator cannot exceed the one of the top-down generate-and-test operator biased to cover a seed example. We now will go on to discuss the extension of the TDD strategy to ILP.

3 Extension of the TDD Strategy to ILP

The TDD strategy, which is biased towards covering a seed example, relies on the reformulation of each negative example e^- as its *lgg* with the seed. The strength of the strategy is that the instance space is merged into the hypothesis space which forms a simple boolean lattice (or a product of a boolean lattice and interval lattices in the case of numerical learning). In this lattice, the TDD refinement operator can efficiently discriminate the negative examples as explained in the previous section. Our approach to upgrade this TDD strategy to ILP consists in working in a seed generalisation space with the same algebraic structure and then computing *lggs* between the negative examples and the seed in such a space. Such algebraic structures can be obtained in relational languages, although with the cost of increased complexity, and sometimes incompleteness.

In this work, we target languages as expressive as non-recursive Datalog clauses with negation [21]. In order to deal with numerical data, we add constraint variables to the language (see e.g. [32]) and classically set their generalisation language to the lattice of convex intervals for numerical variables [22].

A substantial number of works have been done on computing *lggs* in restrictions of first-order logic under several partial orders [27,14,17,12]. One particularly interesting partial order is the *Object Identity* (OI) subsumption. It is stronger than the well-known θ-subsumption, because matching substitutions are limited to be injective, that is, each variable has to be bound to a different object. It has been shown in [38,10] that a Datalog space lower-bounded by a null element (the seed here) under OI-subsumption is isomorphic to a boolean lattice: the set of generalisations of a clause is its power set (up to a variable renaming) and the complete generalisation operator is the dropping-literal rule. An important corollary is that the TDD strategy is complete under OI-subsumption. However, as noted for example in [14,18], the generalisation of two examples is not unique, as opposed to AV learning, and computing their least general generalisation will yield several *lggs*, as shown in figure 3.

$$
\begin{array}{r|l}
s: & west(T) \leftarrow car(T, V_1), rectangular(V_1), \\
 & \qquad\qquad car(T, V_2), \#wheels(V_2, 2), \\
 & \qquad\qquad car(T, V_3), \neg roof(V_3), short(V_3), circular(V_3). \\
e^-: & west(T') \leftarrow car(T', V_1'), \neg roof(V_1'), rectangular(V_1'), \\
 & \qquad\qquad car(T', V_2'), \neg roof(V_2'), \#wheels(V_2', 3), circular(V_2'), \\
 & \qquad\qquad car(T', V_3'), triangular(V_3'), short(V_3').
\end{array}
$$

Fig. 3. A train-like problem with a positive example s and a negative example e^-

This figure describes a toy relational learning problem inspired by the Michalski's trains. The semantic is classical: s is a train having three cars, one is rectangular, the other has two wheels and the last one does not have a roof, is short and circular. Using s as a seed example, it is equivalent to consider the new learning problem where all *lggs* between the seed and the negative examples have to be rejected in the generalisation space of s. This new problem is given in figure 4 as well as its propositional encoding, given the OI-subsumption order.

In fact, any partial order can be used within the TDD strategy, like θ-subsumption, as soon as the generalisation space of the seed example is limited to a boolean lattice for the logical part (the logical part of a clause excludes literals with numerical variables). This is at the expense of completeness, as it is known that the space of generalisation under θ-subsumption is infinite even in Datalog [27] and that no ideal refinement exists for this partial order [37]. Various restrictions in generate-and-test approaches have been proposed to define operational restrictions of θ-subsumption, the most usual one consisting in restricting the generalisation space of the clause to its power-set (see e.g. [24,35]), which exactly corresponds to the applicability condition of the TDD strategy. Let us now provide an example that illustrates this *lgg-based* representation change as well as a sketch of the algorithm to compute these *lggs*.

e_1^- :	$west(T) \leftarrow car(T, V_1),$ $car(T, V_2), \#wheels(V_2, [2, 3]),$ $car(T, V_3), \neg roof(V_3).$
e_2^- :	$west(T) \leftarrow car(T, V_1), rectangular(V_1),$ $car(T, V_2),$ $car(T, V_3), \neg roof(V_3), circular(V_3).$
e_3^- :	$west(T) \leftarrow car(T, V_1), rectangular(V_1),$ $car(T, V_2), \#wheels(V_2, [2, 3])$ $car(T, V_3), short(V_3).$

s	$car(T, V_1)$	$rec(V_1)$	$car(T, V_2)$	$\#w(V_2, N)$	N	$car(T, V_3)$	$\neg roof(V_3)$	$short(V_3)$	$cir(V_3)$
e_1^-	1	0	1	1	$[2, 3]$	1	1	0	0
e_2^-	1	1	1	0	-	1	1	0	1
e_3^-	1	1	1	1	$[2, 3]$	1	0	1	0

Fig. 4. A train-like problem and its reformulation with s as seed example, with OI-subsumption as partial ordering on the relational search space

3.1 Example

In order to exemplify the approach, let us solve the learning problem presented in figure 3. This problem is reformulated by replacing e^- by the three clauses resulting from the computation of $lgg(s, e^-)$ as shown in figure 4. For instance, e_2^- in figure 4 represents the lgg obtained with the matching substitution $\{V_1/V_1', V_2/V_3', V_3/V_2'\}$ between s and e^- of figure 3. The learning algorithm is that of figure 2. In this example, we instantiate the beam size k to 2. The candidate literals to refine the top clause produced by the rejection of the first negative example e_1^- are:

$$\{rectangular(V1); N \in (-\infty, 3); short(V_3); circular(V_3)\}$$

Those produced by the second negative example e_2^- are:

$$\{\#wheels(V_2, N); short(V_3)\}$$

Note that the examples e_1^-, e_2^- and e_3^- are incomparable with respect to $\geq_h$, but according to algorithm of figure 2, we chose to reject e_2^- first, as e_2^- is nearer-miss than both e_1^- and e_3^-. Specialising G against e_2^- produces only two refinements, which corresponds to the size of the beam. Selecting nearest-miss examples has the advantage that the algorithm relies as little as possible on the evaluation function to select the best refinements of the current hypothesis. G specialises into two hypotheses with the addition of the literal $\#wheels(V_2, N)$ and the literal $short(V_3)$. The most general specialisation of G is produced by adding all the literals necessary to get linked hypotheses.[2] The following new bound is obtained:

[2] The discriminant literal selected by the TDD strategy do not necessarily produce a connected clause. We assume in this example and in this work that adding the literals to produce linked clauses, such as $car(T, V_2)$ and $car(T, V_3)$, is simple.

$$\begin{aligned} G = \{\ & west(T) \leftarrow car(T,V_2), \#wheels(V_2,N);\\ & west(T) \leftarrow car(T,V_3), short(V_3)\} \end{aligned}$$

We now take each hypothesis in G and check for their correctness. None of them are correct and both cover the nearest-miss e_3^-. After another specialisation step, we obtain the new G bound:

$$\begin{aligned} G = \{\ & \mathbf{west(T) \leftarrow car(T,V_2), \#wheels(V_2,N), N \in (-\infty,3)};\\ & west(T) \leftarrow car(T,V_2), \#wheels(V_2,N), car(T,V_3), circ(V_3);\\ & \mathbf{west(T) \leftarrow car(T,V_2), \#wheels(V_2,N), car(T,V_3), \neg roof(V_3)};\\ & west(T) \leftarrow car(T,V_3), short(V_3),\\ & \qquad car(T,V_2), \#wheels(V_2,N), N \in (-\infty,3);\\ & west(T) \leftarrow car(T,V_3), short(V_3), circular(V_3);\\ & west(T) \leftarrow car(T,V_3), short(V_3), \neg roof(V_3)\} \end{aligned}$$

Let us now assume that the evaluation function selects the two hypotheses in boldface in the previous list: the first hypothesis is correct and is a candidate solution. The second one is incorrect (it covers e_1^-) and will in turn be specialised. Finally, after specialising and pruning G, we obtain at the end of this refinement step:

$$\begin{aligned} G = \{\ & west(T) \leftarrow car(T,V_2), \#wheels(V_2,N), N \in (-\infty,3);\\ & west(T) \leftarrow car(T,V1), rectangular(V1), car(T,V_2), \#wheels(V_2,N),\\ & \qquad car(T,V_3), \neg roof(V_3)\} \end{aligned}$$

After three refinement steps, we have a subset of all correct hypotheses with respect to the initial relational example e^-. We can notice that the second hypothesis has six literals which would have required six refinement steps with a generate-and-test approach *a la* FOIL or PROGOL.

3.2 Computation of a Nearest-Miss of the Seed from a Negative Example

At the core of algorithm of figure 2 is the computation of nearest-miss examples among *lggs* between the seed and the negative examples. In this section, we show that their computation is equivalent to the resolution of Weighted Constraint Satisfaction Problems. After recalling the main results on computation of *lggs* under OI, we extend them to handle constraint variables and give an example of encoding for the learning example given above.

A complete algorithm to compute all *lggs* under OI-subsumption has been proposed by [18,12]. This algorithm is based on the observation that these *lggs* are maximally incomparable substructures embedded into Plotkin's *lgg*. Both works propose a graph encoding of the problem such that computing *lggs* under OI-subsumption amounts to extract all incomparable maximal cliques in the graph. We build upon their result but provide some simplifications and an extension of the algorithm to handle constraint variables. First, let us note that not all *lggs* are needed to solve the problem as shown in the algorithm of figure 2:

i) only those more specific than the hypotheses in the current G bound are necessary; ii) only the nearest-miss *lgg* is used for the current specialisation step (see section 2). The problem is then to compute the maximum-clique, that is the largest maximal one, in the corresponding graph. Second, it can be seen that their graph formulation is the consistency graph of a Constraint Satisfaction Problem (CSP). This equivalence between the CSP and the clique problem on the CSP consistency graph is well-known [15,30]. The CSP formulation is more natural as it is equivalent to the one used for computing the covering test in ILP [9]. Therefore, finding the nearest-miss *lgg* between a seed and a negative example corresponds to finding the largest subset of variables in s which admits a consistent variable assignment.

This formulation needs to be adapted for handling constraint variables. To take that information into account, we need to add a valuation structure to the CSP which is known in the literature as a Weighted CSP. Weighted CSP (WCSP) extends the CSP framework by associating costs to tuples. These costs give preferences among partial assignments. The usual task is to find a complete consistent assignment with minimum cost, which is NP-hard. Informally, to compute a nearest-miss, we define a cost as the number of literals and constraint variables' values of the seed example that are not matched onto the negative example. Concerning the numerical literal $\#wheels(V_2, 2)$ in the seed, there are three options: either there is an exactly matching literal in the negative example, the associated cost is then 0. If there is a literal of the form $\#wheels(V_2, N)$ with $N \neq 2$ in the negative example, the cost is 1[3]. Finally, it may also be the case that the literal is unmatched, in that case the cost is 2.

Due to lack of space, we refer to [8,7] for a detailed description of WCSPs and the associated algorithms. Here, we briefly give the definition of a Weighted CSP and illustrate the encoding of the problem of nearest-miss computation of figure 3.

Definition 3 (Weighted CSP). *A binary WCSP is a tuple* (k, X, D, C)*. X and D are the variables and domains as in classical CSP. C is a set of cost functions. A binary constraint C_{ij} assigns costs to assignments of variables i and j, ranging from 0 to k. A unary constraint C_i assigns costs to assignments of variable i, ranging from 0 to k. The cost of a tuple t, noted $cost(t)$, is the sum of all its associated costs. When a constraint C assigns a cost greater than or equal to k to a tuple t ($cost(t) \geq k$), it means that C forbids t, otherwise t is allowed by C, with the corresponding cost. A tuple is consistent if $cost(t) < k$.*

For computing the nearest-miss of the seed example s from the example e^- (see figure 3), we have the corresponding WCSP, omitting the head literal for convenience:

[3] This way of handling cost does not take into account partial ordering between numerical values in the negative examples and this has to be handled through post-processing.

Variables	Domains
$car(T, V_1)$	$1 : nm$ $\mathbf{0 : car(T', V_1')}$ $0 : car(T', V_2')$ $0 : car(T', V_3')$
$rectangular(V_1)$	$1 : nm$ $\mathbf{0 : rectangular(V_1')}$
$car(T, V_2)$	$1 : nm$ $0 : car(T', V_1')$ $0 : car(T', V_2')$ $\mathbf{0 : car(T', V_3')}$
$\#wheels(V_2, 2)$	$\mathbf{2 : nm}$ $1 : \#wheels(V_2', 3)$
$car(T, V_3)$	$1 : nm$ $0 : car(T', V_1')$ $\mathbf{0 : car(T', V_2')}$ $0 : car(T', V_3')$
$\neg roof(V_3)$	$1 : nm$ $0 : \neg roof(V_1')$ $\mathbf{0 : \neg roof(V_2')}$
$short(V_3)$	$\mathbf{1 : nm}$ $0 : short(V_3')$
$circular(V_3)$	$1 : nm$ $\mathbf{0 : circular(V_2')}$

Literals of the seed s (i.e., the variables of the WCSP) are shown in the first column of the table. For each literal of s, we describe candidate matching literals in e^-, i.e., the domains of the WCSP variables. Matching a literal corresponds to satisfying a unary constraint. To each literal of e^-, we associate the corresponding unary cost of matching it. In order to account for unmatched seed literals, we use an additional value *nm* for *not matched*, which indicates that the seed literal is not matched and does not belong to the *lgg*. For instance, if the literal $short(V_3)$ is unmatched, this will have a cost of 1. The binary costs (not shown here) are the same as for a CSP encoding of the subsumption test: they define that a pair of matchings is compatible to ensure that the solution tuple is a *lgg* of the seed and the negative example. To each matched literal of the seed, we associate a substitution θ_i. A pair (θ_i, θ_j) is compatible iff the substitution $\theta_i.\theta_j$ is a valid substitution under the partial order considered. The corresponding cost is zero or k otherwise. The solution of the WCSP which leads to construct e_2^- is outlined in boldface in the table, this solution has cost 3. e_2^- is among the solutions of lowest cost and is used to compute $(west(T) \leftarrow car(T, V_2), \#wheels(V_2, N))$. The computation of the next nearest-miss more specific than this hypothesis is computed by removing the *nm* values from the domain of the two literals $car(T, V_2)$ and $\#wheels(V_2, N)$, thus forcing them to be part of the nearest-miss.

4 Related Works

A first version of the Propal algorithm has been presented in [3], where the link between the TDD strategy and Propal was not made and no formalisation was proposed. Moreover, the algorithm could not deal with numerical data. It was also presented as a propositionalisation system and we plan to further investigate in the future the link between propositionalisation and computation of *lggs* between examples and a seed.

A first comparison has to be made with the learning systems PROGOL, Aleph and MIO. As we said, they use the same search space as Propal, by the mean of a seed example, but are rooted in the generate-and-test paradigm and do not use the TDD strategy. They have to deal with many refinements during the search that are not relevant with respect to the discrimination task.

A related approach to our system is the system STILL [33]. STILL is a propositionalisation system [19] which upgrades the attribute-value learning algorithm

DiVS [31]. DiVS makes use of the extension-against rules of Michalski in the following manner. For each example e (positive and negative) DiVS builds $G(e)$, the bound G covering e and rejecting all the negative examples with respect to its class by applying the extension-against rules. Each $G(e)$ votes to classify unseen examples. The upgrade of DiVS to ILP is done through the use of the propositionalisation technique in an indeterminate language prior to learning [40,32]: all matchings between a seed example and the examples to reformulate are computed and rewritten as attribute-value vectors. To avoid the exponential space requirement of propositionalisation in an indeterminate hypothesis space, the authors perform a sampling of k vectors in the matching space (k a user-supplied parameter). As the propositionalisation technique of STILL randomly selects matchings and is applied before learning, STILL does not benefit from the TDD strategy that focuses on *lggs* between the seed examples and the informative negative examples. As a consequence, STILL mostly extracts irrelevant vectors for the discrimination task as only the ones corresponding to *nearest-miss lggs* are relevant in the case of the TDD strategy (section 2). Therefore STILL, being a randomised polynomial-time algorithm, cannot ensure to output a correct theory with respect to the learning data. However, STILL has been shown to be successful on the "mutagenesis" dataset (B_2 and B_3 only, see section 5) with some parameters inherited from DiVS.

5 Experiments

The TDD strategy implemented in Propal to run the experiments detailed below is the same as AQ's presented figure 2. Propal conducts a beam search in the hypothesis space, guided by the Laplace function[4]. The default beam size is fixed to 5. We extended this basic algorithm to handle missing values in constraint variables (or attributes) with the same technique as AQ's [23] and Ripper's [6]: all tests involving the constraint variable V are defined to fail on examples for which the value of V is missing.

To solve the WCSPs, Propal's implementation of nearest-miss extraction relies on the state-of-the-art complete algorithm Toolbar[5] [8]. We have used in the experiments the default parameters of Toolbar. However, we set the timer of Toolbar to 60 seconds to keep computation of a nearest-miss within a reasonable amount of time. This is usually needed for the one or two last seeds of problems like "mutagenesis", that can be quite large compared to the other positive examples. When the time limit is reached, Toolbar returns the best solution (i.e., the most specific negative example) found so far. As shown in [3], this approximation degrades the heuristic search by increasing the branching factor but still ensures the correctness of the output theory.

We validate our implementation of the TDD strategy in ILP by comparing Propal's performances with the ILP systems FOIL, PROGOL, STILL and

[4] The Laplace function is defined as $\frac{p+1}{p+n+2}$, with p and n the number of positive and negative examples covered by the hypothesis.

[5] http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro

TILDE on the "mutagenesis" datasets [36]. The "mutagenesis" dataset used is *regression-friendly* with the 4 versions of background knowledge (from B_1 to B_4). We made additional comparisons with the two propositionalisation systems RSD and RELAGGS, on the "KRK illegal chess position" [25] datasets and two learning problems extracted from the PKDD99 financial challenge by [20]. The two last problems involve learning to classify bank loans into profitable and non-profitable loans. For the last three problems, we performed a 10-fold cross-validation averaged over 10 runs as in [20]. For the "mutagenesis" dataset, we followed the protocol described in [36].

On the "mutagenesis" datasets, the results for FOIL and PROGOL have been taken from [36], for TILDE from [4] and for STILL from [33]. On the "KRK illegal position" and the two tasks from PKDD99, the results are taken from [20]. When several values of parameters were tried for these systems, we chose their best results. As noted in [29], this can produce an optimistic bias in favour of the other algorithms compared to PROPAL, which is run on all datasets with a standard size of beam of 5. This is the only parameter of PROPAL for now as we recall that no noise-coping strategy has been implemented.

Table 1. Accuracy in % of learnt theories by PROGOL, FOIL, TILDE, STILL, RSD, RELAGGS and PROPAL on the "mutagenesis", "KRK" and "loans" datasets, and time for PROPAL to output the theories

	$B1$	$B2$	$B3$	$B4$	KRK.illegal	Loan (AvB)	Loan (ACvBD)
PROGOL	76	81	83	88	n.a.	45.7	n.a.
FOIL	-	75.8	83	86	97.2	-	87.3
TILDE	75	79	85	86	75.1	-	n.a.
STILL	-	86.5	88.8	-	-	-	-
RELAGGS	-	-	-	-	72.3	88	94.1
RSD	-	-	-	-	76.2	n.a.	n.a.
PROPAL	85.5	86.7	88.2	85.1	100	84,4	85,19
Time (s.)	3692	60698	40949	8274	179	117	564

Table 1 summarises the results. The symbol "-" indicates that the result is not available or that the experiments have been done with a different protocol. The symbol "n.a." indicates that the learner exhausted the time limit of 2 days of computation on at least one of the fold as reported in [20].

We can see from table 1 that PROPAL's performance is competitive with the state-of-the-art generate-and-test approaches which use sophisticated heuristic search and pruning techniques. On B_1, which is the hardest domain for learning in the "mutagenesis" domain, PROPAL performed as well as the other systems with B_3, which uses expert attributes; the performance on B_1, B_2 and B_3 are among the best reported. We see a lower performance on the richest domain B_4, where descriptions of higher-level structures that appear in a molecule are added. This over-fitting may be explained by the large increase in the size of the hypothesis space, as PROPAL does not restrict the search space beyond the choice of a seed example, and the fact that no noise-coping strategies are implemented.

The "KRK" dataset is a good example where the TDD strategy pays off: the dataset provides a lot of near-misses (the branching factor being often reduced to 1) and it can be considered noise-free. The result largely improves those of the propositionalisation systems and of TILDE.

Another example of a good performance of Propal is on the "Loan" datasets, which is advocated in [20] as representative of large datasets where current ILP systems do not perform well: on "loanAvB", PROGOL has an accuracy below 50% and RSD cannot solve at least one fold after two days of computation; on "loanACvDB", they run out of time, as well as TILDE and only FOIL performs well on it. RELAGGS [20] performs best with 88% and 94.1% respectively. Propal is able to solve the two problems quickly with rather good performance comparatively.

6 Conclusion

We have studied in this paper the TDD strategy, popularised by the AQ family, in the context of ILP. We made a link between AQ, Winston's work on near-misses and a change of representation of the negative examples through *lggs* computed with a seed example. This *lgg*-based reformulation merges the instance space and the search space into a simpler learning space, where the learning problem can be solved algebraically. This formalisation allowed us to propose a simple extension of the TDD strategy to ILP in languages as expressive as non-recursive Datalog clauses with negation. The TDD strategy offers a theoretical advantage over generate-and-test systems such as PROGOL, Aleph and MIO, by making it possible to prune irrelevant branches of the refinement graph by using most relevant negative examples. The extraction of nearest-miss examples through a *lgg*-based reformulation has been formalised as a Weighted CSP, allowing a flexible implementation of the AQR strategy within Propal using a state-of-the-art WCSP solver, Toolbar. This implementation, which does not include any noise-handling mechanism, has been shown to be competitive with generate-and-test FOL learners and propositionalisation systems. However, it is known that data-driven strategies are more prone to noise issues than their generate-and-test counterparts. We plan to further validate the approach by studying the impact of noise. In particular, we plan to investigate the works in this domain proposed for the AQ system [16] and for rule learning [11]. Secondly, now that the mechanism for extracting nearest-miss examples has been implemented within Toolbar, we plan study the impact of various propagation mechanisms and various approximation strategies on Propal's running time and performance.

Acknowledgements

We are very grateful to H. Soldano for the numerous discussions on the Top-Down Data-Driven strategy and its relevance to ILP, and to A. Osmani for sharing with us his many insights concerning WCSP. We also would like to thank Mark-André Krogel for providing the "loan" datasets and for helping us with the experiments. Finally, we thank Christophe Caron of the MIG-INRA group and the LRI-UMR8623, for providing us access to their computational resources.

References

1. Alphonse, E.: Macro-opérateurs et Sélection Relationnelle en Programmation Logique Inductive: théorie et algorithmes. PhD thesis, Université Paris-Sud (2003)
2. Alphonse, E., Osmani, A.: On the connection between the phase transition of the covering test and the learning success rate. In: Proc. 16th Conf. of Inductive Logic Programming (2006)
3. Alphonse, E., Rouveirol, C.: Lazy propositionalization for relational learning. In: Proc. ECAI'2000, pp. 256–260. IOS Press, Amsterdam (2000)
4. Blockeel, H., De Raedt, L.: Top-down induction of first order decision trees. Artificial Intelligence 101, 285–297 (1998)
5. Clark, P., Niblett, T.: The CN2 induction algorithm. Machine Learning 3, 261–283 (1989)
6. Cohen, W.W.: Fast effective rule induction. In: Proc. 12th ICML, pp. 115–123. Morgan Kaufmann, San Francisco (1995)
7. de Givry, S., Larrosa, J., Meseguer, P., Schiex, T.: Solving Max-SAT as weighted CSP. In: Proc. CP 2003, pp. 363–376 (2003)
8. de Givry, S., Zytnicki, M., Heras, F., Larrosa, J.: Existential arc consistency: Getting closer to full arc consistency in weighted CSP. In: Proc. of IJCAI-05 (2005)
9. Eisinger, N.: Subsumption and connection graphs. In: Proc. of IJCAI'81, pp. 480–486. William Kaufmann (1981)
10. Esposito, F., Laterza, A., Malerba, D., Semeraro, G.: Refinement of Datalog programs. In: Proc. of the MLnet Familiarization Workshop on ILP for KDD, pp. 73–94 (1996)
11. Fürnkranz, J.: Pruning methods for rule learning algorithms. In: Proc. 4th Int. Workshop on ILP, pp. 321–336 (1994)
12. Geibel, P., Wysotzki, F.: A Logical Framework for Graph Theoretical Decision Tree Learning. In: Proc. ILP'97 (1997)
13. Giordana, A., Saitta, L., Sebag, M., Botta, M.: Analyzing relational learning in the phase transition framework. In: Proc. ICML, pp. 311–318 (2000)
14. Haussler, D.: Learning conjunctive concepts in structural domains. Machine Learning 4(1), 7–40 (1989)
15. Jagota, A.: Constraint satisfaction and maximum clique. In: Working Notes, AAAI Spring Symposium on AI and NP-hard Problems, pp. 92–97 (1993)
16. Kaufman, K.A., Michalski, R.S.: Learning from inconsistent and noisy data: The AQ18 approach. In: Proc. of the Eleventh ISMIS, pp. 411–419 (1999)
17. Kietz, J.-U.: A comparative study of structural most specific generalisations used in machine learning. In: Proc. Third Workshop on ILP, pp. 149–164 (1993)
18. Kietz, J.-U.: Some computational lower bounds for the computational complexity of inductive logic programmming. In: Proc. 6th ECML, Vienna, Austria (1993)
19. Kramer, S., Lavrac, N., Flach, P.: Propositionalization approaches to relational data mining. In: Dzeroski, S., Lavrac, N. (eds.) Relational Data Mining, pp. 262–291. Springer, Heidelberg (2001)

20. Krogel, M.: On Propositionalization for Knowledge Discovery in Relational Databases. PhD thesis, Univ. Magdeburg (2005)
21. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Berlin (1987)
22. Michalski, R.S.: A theory and methodology of inductive learning. Machine Learning: An Artificial Intelligence Approach I, 83–134 (1983)
23. Michalski, R.S., Wojtusiak, J.: Reasoning with meta-values in AQ learning. Technical report, George Mason University (2006)
24. Muggleton, S.: Inverse entailment and PROGOL. New Generation Computing 13, 245–286 (1995)
25. Muggleton, S.H., Bain, M., Hayes-Michie, J., Michie, D.: An experimental comparison of human and machine learning formalisms. In: Proc. 6th IWML, San Mateo, CA, pp. 113–118. Morgan Kaufmann, San Francisco (1989)
26. Castillo, L.P., Wrobel, S.: On the stability of example-driven learning systems: A case study in multirelational learning. In: Coello Coello, C.A., de Albornoz, Á., Sucar, L.E., Battistutti, O.C. (eds.) MICAI 2002. LNCS (LNAI), vol. 2313, pp. 321–330. Springer, Heidelberg (2002)
27. Plotkin, G.: A note on inductive generalization. In: Machine Intelligence, vol. 5, Edinburgh University Press, Edinburgh (1970)
28. Quinlan, J.R.: Learning logical definitions from relations. Machine Learning 5(3), 239–266 (1990)
29. Scheffer, T., Herbrich, R.: Unbiased assessment of learning algorithms. In: Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI'97), pp. 798–803 (1997)
30. Scheffer, T., Herbrich, R., Wysotzki, F.: Efficient θ-subsumption based on graph algorithms. In: Inductive Logic Programming. LNCS, vol. 1314, pp. 312–329. Springer, Heidelberg (1997)
31. Sebag, M.: Delaying the choice of bias: a disjunctive version space approach. In: Proc. 13th ICML, pp. 444–452 (1996)
32. Sebag, M., Rouveirol, C.: Constraint inductive logic programming. In: Advances In Inductive Logic Programming, pp. 277–294. IOS Press, Amsterdam (1996)
33. Sebag, M., Rouveirol, C.: Resource-bounded relational reasoning: Induction and deduction through stochastic matching. Machine Learning 38(1/2), 41–62 (2000)
34. Smith, B.D., Rosenbloom, P.S.: Incremental non-backtracking focusing: A polynomially bounded generalization algorithm for version spaces. In: Proc. 8th AAAI, pp. 848–853 (1990)
35. Srinivasan, A.: A learning engine for proposing hypotheses (Aleph) (1999)
36. Srinivasan, A., Muggleton, S., King, R.D.: Comparing the use of background knowledge by inductive logic programming systems. In: De Raedt, L. (ed.) Proc. of the 5th ILP Workshop, pp. 199–230. Scientific Report, K.U.Leuven (1995)
37. van der Laag, P.R.J., Nienhuys-Cheng, S-H.: Existence and nonexistence of complete refinement operators. In: Proc. of the 7th ECML, pp. 307–322. Springer, Heidelberg (1994)
38. VanLehn, K.: Efficient specialization of relational concepts. Machine Learning 4, 99–106 (1989)
39. Winston, P.H.: Learning structural descriptions form examples. In: Winston, P.H. (ed.) The Psychology of Computer Vision, pp. 157–209. McGraw-Hill, New York (1975)
40. Zucker, J.-D., Ganascia, J.-G.: Selective reformulation of examples in concept learning. In: Proc. 11th ICML, pp. 352–360 (1994)