Mach Learn (2009) 76: 73–107
DOI 10.1007/s10994-009-5116-8

Using the bottom clause and mode declarations in FOL theory revision from examples

Ana Luísa Duboc · Aline Paes · Gerson Zaverucha

Received: 15 November 2008 / Revised: 3 April 2009 / Accepted: 17 April 2009 /
Published online: 12 June 2009

Abstract Theory revision systems are designed to improve the accuracy of an initial theory, producing more accurate and comprehensible theories than purely inductive methods. Such systems search for points where examples are misclassified and modify them using revision operators. This includes trying to add antecedents to clauses usually following a top-down approach, considering all the literals of the knowledge base. Such an approach leads to a huge search space which dominates the cost of the revision process. ILP Mode Directed Inverse Entailment systems restrict the search for antecedents to the literals of the bottom clause. In this work the bottom clause and mode declarations are introduced in a first-order logic theory revision system aiming to improve the efficiency of the antecedent addition operation and, consequently, also of the whole revision process. Experimental results compared to revision system FORTE show that the revision process is on average 55 times faster, generating more comprehensible theories and still not significantly decreasing the accuracies obtained by the original revision process. Moreover, the results show that when the initial theory is approximately correct, it is more efficient to revise it than learn from scratch, obtaining significantly better accuracies. They also show that using the proposed theory revision system to induce theories from scratch is faster and generates more compact theories than when the theory is induced using a traditional ILP system, obtaining competitive accuracies.

Keywords ILP · Theory revision · Mode directed inverse entailment (MDIE)

Editors: Filip Zelezny and Nada Lavrac.

This is an extended and revised version of the ILP 2008 paper (Duboc et al. 2008).

A.L. Duboc · A. Paes (✉) · G. Zaverucha
Department of Systems Engineering and Computer Science–COPPE, Federal University of Rio de Janeiro (UFRJ), P.O. Box 68511, 21945-970 Rio de Janeiro, RJ, Brazil
e-mail: ampaes@cos.ufrj.br

A.L. Duboc
e-mail: aduboc@cos.ufrj.br

G. Zaverucha
e-mail: gerson@cos.ufrj.br

1 Introduction

Inductive Logic Programming algorithms learn a First-order Logic (FOL) theory from a set of positive and negative examples and background knowledge (BK) (Muggleton and De Raedt 1994; Nienhuys-Cheng et al. 1997; Dzeroski and Lavrac 2001; De Raedt 2008). The BK is composed of a set of clauses assumed as correct and therefore it cannot be modified. On the other hand, FOL theory revision from examples systems divide the initial theory in two parts: the first one is assumed as correct and therefore is fixed and the second one may contain incorrect rules which need to be modified in order to correctly explain the dataset (Wrobel 1996). Theory revision systems operate by searching for revision points, that is, the points which explain faults in the theory, and then proposing revisions to such points, through applying at each point a number of revision operators. As the initial theory is a good starting point and the revision process takes advantage of it, the theories returned by revision systems are usually more accurate than theories learned from standard ILP systems using the same dataset. Moreover, if the initial theory is approximately correct, it is more efficient to revise it than learn a whole new theory from scratch.

Theory revision systems such as FORTE (Richards and Mooney 1995) include addition of antecedents in a clause as one of the possible modifications in failing points of the theory. In that system, the search for antecedents follows FOIL's (Quinlan 1990) top-down approach which eventually generates a huge search space of literals, with some of these literals not covering even one positive example. On the other hand, ILP algorithms such as Progol (Muggleton 1995) and Aleph (Srinivasan 2001), restrict the search for literals to those belonging to the bottom clause. The bottom clause contains literals relevant to a positive example, collected from a Mode Directed Inverse Entailment (MDIE) search in the BK. This hybrid bottom-up and top-down approach often generates many fewer literals, and they are also guaranteed to cover at least one positive example (the one used to generate the bottom clause). Just as the use of the bottom clause brings the advantage of reducing the search space of Progol when compared to a top-down approach as FOIL, it could also reduce the search space of antecedents of FORTE, which presently generates antecedents in the FOIL manner. Thus, in this work, we propose to use the literals of the bottom clause to define the search space for antecedents to add in the clauses being revised. Additionally, we use mode declarations to validate if the antecedents of the bottom clause can effectively be added to such clauses.

Dietterich et al. (2008) pointed out that although there were several theory revision systems developed in the past (Wrobel 1996; De Raedt 2008) they were not widely used for the lack of applications with substantial codified initial theory and that the situation has changed recently with the availability of large-scale resources of background knowledge in areas such as biology (Muggleton 2005). Therefore, they concluded that with the availability of public databases in the sciences there is an increasing need for the development of efficient theory revision ILP systems. This work contributes towards this goal of bringing a theory revision system to be as efficient as a state of the art ILP system and achieving significantly higher accuracies.

Moreover, standard ILP systems search for the final hypothesis through a covering approach, where at each iteration a single clause is learned and the positive examples covered by the clause are removed from the list of examples. Although the covering algorithm is considered to be faster than non-covering approaches, where the hypothesis (a clausal theory) is constructed as a whole and globally evaluated, it can generate many clauses unnecessarily long and only locally optimal (Bratko 1999). Besides that, the covering approach usually learns only single predicates. A solution to this problem would be to construct hypotheses as a whole by using a refinement operator for entire theories. Unfortunately, searching

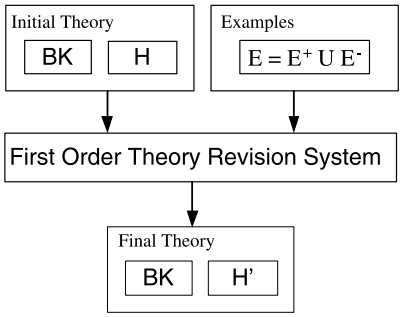

Fig. 1 Theory revision schema

in the space of whole theories is known to be more expensive than using a covering approach (Badea 2001). Theory revision systems do not use a covering approach, since they consider the whole theory instead of individual clauses when proposing modifications to the failing points. However, the revision approach inherently works with a reduced search space, since the focus in such systems is only on the misclassified examples and the failing points of the theory. Therefore, another contribution of this work is to show through experimental results that using a theory revision approach to induce whole theories from scratch can be faster than using a covering approach while obtaining competitive accuracy and more compact theories.

The rest of this paper is organized as follows. Sections 2 and 3 provide background concerning theory revision and inverse entailment, respectively. Then, the proposed approach of this work is devised in Sect. 4, followed by the experimental results in Sect. 5. Finally, the work is concluded in Sect. 6.

2 First-order logic theory revision

Figure 1 presents a schema for theory revision. The theory revision system receives an *initial theory* and a set of examples $E^+ \cup E^-$, divided into positive and negative sets, respectively. The initial theory includes two components: an invariant component, named *background knowledge (BK)*, and *the Hypothesis (H)* component that can be modified. H is composed of function-free first-order Horn clauses, while BK, E^+ and E^- consist of Horn clauses.

Ideally, the revision process should generate a final hypothesis H' that, together with BK, will entail all the positive examples in E^+ and none of the negative examples, E^-, that is, the final theory will be consistent with the dataset. Notice that learning in Inductive Logic Programming (ILP) can be seen as a particular case of theory revision where H is initially empty.

2.1 Revision points

As theory revision deals with multiple predicates, many clauses can be involved in the proof of a negative example as well as many clauses could be used to prove a positive example. Nevertheless many clauses can be already correct and do not need to be modified. Therefore, it is necessary to find the points in the initial theory that need to be corrected, which are called revision points. There are two types of revision points:

– *Generalization*—the literal in a clause responsible for a failure in the proof of some positive example (failure point) and other literals in the theory that may have contributed to this failure (contributing points);

– *Specialization*—clauses used in successful proofs of negative examples.

The specification of the revision point determines how the theory should be modified to be consistent with the dataset. The modifications in the theory are devised by the *revision operators*, which could be of two types. They may add previously missing answers through *generalization* or remove incorrect answers through *specialization* (Wrobel 1996; Richards and Mooney 1995).

2.2 Revision operators

Theory revision relies on operators that propose modifications at each *revision point*. Any operator used in first-order machine learning can be used in a theory revision system. Below, we briefly describe some operators used in this work.

The operators for specialization are:

- *Delete-rule*—this commonly used operator removes a clause that was used in the proof of a negative example.
- *Add-antecedent*—this operator adds antecedents to an incorrect clause, where an antecedent is defined as a literal in the body of the clause.

The generalization operators are:

- *Delete-antecedent*—this operator removes failed antecedents from clauses that could be used to prove positive examples.
- *Add-rule*—this operator generates new clauses from existing ones using deletion of antecedents followed by addition of antecedents. It is also possible to create an entirely new clause.

There are other approaches of generalization and specialization operators. For more details on revision operators we refer the reader to Richards and Mooney (1995) and Wrobel (1996).

2.3 FORTE

Although there are several theory revision systems described in the literature (De Raedt 2008) in this work we follow the First Order Revision of Theories from Examples (FORTE) system (Richards and Mooney 1995). For an extensive comparison among FOL Revision Systems, including MIS (Shapiro 1983), RUTH (Adé et al. 1994), MOBAL (Morik et al. 1993), among others, we refer the reader to Wrobel (1996).

FORTE performs a Hill-Climbing search through a space of both specialization and generalization operators in an attempt to find a minimal revision to a theory that makes it consistent with the set of training examples. The hypothesis is composed of function-free Horn clauses, background knowledge consisting of Horn clauses and the examples of ground definite clauses. Unlike some theory revision systems that are incremental, FORTE is a batch revision system in the sense that the examples are all processed at once. The skeleton of FORTE is shown in Algorithm 1. The key ideas are:

1. Identify all the revision points in the current theory.
2. Generate a set of proposed revisions for each revision point starting from the one with the highest potential and working down the list. *Potential* is defined as the number of misclassified examples that could be turned into correctly classified from a revision in that point.
3. Score each revision through the actual increase in theory accuracy it achieves.

Algorithm 1 FORTE Algorithm (Richards and Mooney 1995)

```
repeat
    generate revision points;
    sort revision points by potential (high to low);
    for each revision point do
        generate revisions;
        update best revision found;
    until potential of next revision point is less than the score of the best revision to date
    if best revision improves the theory then
        implement best revision;
until no revision improves the theory;
```

4. Retain the revision which increases the score most.

Each revision is scored considering the examples correctly and incorrectly classified before and after the revision. Thus, before performing the revision, the examples—correct or incorrect—whose provability can be affected by it are collected. Then, after proposing the revision, the difference between the incorrect examples which become correct and the correct examples which become incorrect is the actual score of the revision. The score is expressed by the formula

$$Improvement\_on\_accuracy = right - wrong \tag{1}$$

where *right* is the number of examples which were incorrectly classified before the revision and become correctly classified after it and *wrong* is the number of examples which were correctly classified before the revision and become incorrectly classified after it.

FORTE stops when the potential of next revision point is less than the score of the best revision so far. If the best revision really improves the theory it is implemented.

Antecedents addition Now we further discuss the add antecedents operation of FORTE since this is the operator modified in this work to make the revision process more efficient. There are two algorithms for adding antecedents in a clause:

1. Hill Climbing—This algorithm is based on FOIL and adds one antecedent at time. It works as follows. First, all possible antecedents are created and scored using a slightly modified version of the FOIL score, displayed in formula (2), where *Old_score* is the score of the clause without the literal being evaluated, *Positives* is the number of positive examples proved by the clause with the literal added to it and *Negatives* is the number of negative examples proved by the clause with the literal. The difference concerns the fact that FOIL score counts the number of proofs of instances, whereas FORTE counts the number of provable instances, ignoring the fact that one instance may be provable in several different ways. Next, the antecedent with the best score is selected. If the best score is better than the current clause score, the antecedent is added to the clause. This process continues until either there are no further antecedents to be added in the clause or no antecedent can improve the current score. This approach is susceptible to local maxima.

$$foil\_based\_score = Positives * (Old\_score - \log(Positives/(Positives + Negatives))) \tag{2}$$

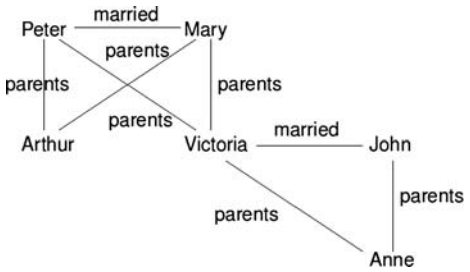

Fig. 2 An example of a relational graph representing part of the family domain (Richards and Mooney 1995)

2. Relational Pathfinding—In this approach a sequence of antecedents is added to a clause at once in attempt to skip local maxima, as, sometimes, none of the antecedents put individually in the clause improves its performance.

   The Relational Pathfinding algorithm is based on the assumption that generally in relational domains there is a path with a fixed set of relations connecting a set of terms and such path satisfies the target concept. A relational domain can be represented as a graph where the nodes are the terms and the edges are the relations among them. Thus, we can define a *relational path* as the set of edges (relations) which connect nodes (terms) of the graph. To better visualize such an approach, consider, for instance, the graph in Fig. 2, which represents a part of a family domain. In this graph, horizontal lines denote marriage relationships, and the remaining lines denote parental relationships.

   Now, suppose we want to learn the target concept *grandfather*, given an empty initial rule and a positive example *grandfather(peter,anne)*. The relational path between the terms *peter* and *anne* is composed of the relation *parents* connecting *peter* to *victoria*, and also of the relation *parents* connecting *victoria* to *anne*. From these relations, the path *parents(peter,victoria), parents(victoria,anne)* is formed, which can be used to define the target concept $grandfather(A, B) : -parents(A, C), parents(C, B)$.

   Therefore, the *Relational Pathfinding* algorithm aims to find such paths given a relational domain, since important concepts are represented by a small set of fixed paths between terms defining a positive example. From the point of view of theory revision, this algorithm can be used whenever a clause needs to be specialized and it does not have relational paths connecting its variables. In this case, a positive example proved by the clause is chosen to instantiate it, and then, from the ground clause, relational paths to the terms without a relationship in the clause are searched.

   If the new relations found have introduced new terms that appear only once, FORTE tries to complete the clause by adding relations that hold between these singletons and other terms in the clause; these new relations are not allowed to eliminate any of the currently provable positive instances. If FORTE is unable to use all of the new singletons, the relational path is rejected.

When specializing a clause using any of these algorithms, if the antecedents added to the clause make some positive example unprovable, the just revised clause is added to the set of modifications proposed to the theory and a new search for antecedents starts from the original clause, in an attempt to recover the provability of the positive examples. This process continues until all the positive examples originally covered by the initial clause are again provable.

Antecedents generation When creating literals to add in the body of a clause, all the predicates defined in the knowledge base are considered. A literal is created from a predicate by instantiating its arguments by variables, while respecting the following constraints:

Algorithm 2 Hill Climbing Antecedent Generation Algorithm

```
1:  C = Clause to be specialized
2:  for each literal in the knowledge base do
3:     return the variables and their types from C
4:     return the arguments and their types from the particular literal
5:     for each combination of variables from C do
6:        Verify if the combination is valid as possible arguments of the literal
7:        if the combination is valid then
8:           Replace the arguments of the literal for the combination of variables found
9:           return the literal as a new antecedent
10:       else
11:          Go to the next combination of variables available
12:    N = number of arguments of the particular literal
13:    i = 1
14:    while i ≤ N − 1 do
15:       Create a new variable L
16:       V = Variables of C + L
17:       for each combination in V that includes L and some variable of C do
18:          Verify if the combination is valid as possible arguments of the literal
19:          if the combination is valid then
20:             Replace the arguments of the literal for the combination of variables found
21:             return the literal as a new antecedent
22:          else
23:             Go to the next combination of variables available
24:       i = i + 1
25:    Go to the next literal in the knowledge base
```

1. At least one variable of the new literal must be in the clause being revised;
2. The arguments of the literals must obey the types defined in the knowledge base.

Such constraints do not explore properly the connections among variables, since they are not defined as input or output variables (there are no *mode declarations*). Besides, several different variations of the same literal are generated from the existing variables and the new ones. Clearly, the larger the number of new variables in the clause, the more antecedents are created. Actually, the space complexity grows exponentially in the number of new variables since the complexity of enumerating all possible combinations of variables is exponential in the arity of the predicate. Thus, following such an approach, the antecedent addition operation is inefficient, since all the literals created must be scored in order to choose the best one. Moreover, common scoring functions in ILP are very expensive since they involve attempts to prove the examples.

The process of generating literals is slightly different for the two antecedents generation algorithms. The algorithms for these processes can be seen in Algorithm 2 and Algorithm 3, for Hill Climbing and Relational Pathfinding algorithms, respectively.

As already mentioned, the Relational Pathfinding algorithm starts from a clause grounded from a positive example covered by the clause. The terms in the ground clause will be the nodes in the graph, connected by the relations defined in the body of the clause. The algorithm constructs the graph iteratively, starting from these initial nodes and expanding them until finding the relational paths. The *end values* are the terms (nodes) created when a node is expanded.

Algorithm 3 Relational Pathfinding Antecedent generation

```
for each literal in the knowledge base do
    return the end values and their types in the node been expanded
    return the arguments and their types from the particular literal
    for each combination of end values in the node do
        Verify if the combination is valid as possible arguments of the literal
        if the combination is valid then
            Replace the arguments of the literal for the combination of end values found
            return the literal as a new antecedent
        else
            Go to the next combination of end values
    N = number of arguments of the particular literal
    i = 1
    while i ≤ N − 1 do
        Create a new variable L
        V = end values of the node + L
        for each combination in V that includes L and some end value of the node do
            Verify if the combination is valid as possible arguments of the literal
            if the combination is valid then
                Replace the arguments of the literal for the combination of end values found
                Search in the background knowledge for a fact that unifies with the literal created
                if a fact is found then
                    return the literal as a new antecedent
                else
                    Go to the next combination of variables available
            else
                Go to the next combination of variables available
        i = i + 1
    Go to the next literal in the knowledge base
```

3 Mode directed inverse entailment and the bottom clause

This section briefly reviews the method of mode-directed inverse entailment originally described in Muggleton (1995). Given background knowledge BK, a set of positive examples E^+ and negative examples E^-, the goal of inductive logic programming systems is to find a hypothesis such that

$$BK \wedge H \models E^+ \quad \text{and} \quad BK \wedge H \not\models E^-$$

As each clause in the simplest H should explain at least one example, let us consider the case of H and E^+ each being single Horn clauses. Then, using the contraposition law, the first formula above is inverted to

$$BK \wedge \overline{E^+} \models \overline{H}$$

If H and E^+ are restricted to be single Horn clauses, $\overline{H}$ and $\overline{E^+}$ will be ground skolemised unit clauses. Considering $\overline{\perp}$ as the conjunction of ground literals which are true

in all models of $BK \wedge \overline{E^+}$, we have

$$BK \wedge \overline{E^+} \models \overline{\bot}$$

The *bottom clause* $\bot$ with respect to the example E^+ and BK is the most specific clause such that

$$BK \wedge \bot \models E^+$$

Since $\overline{H}$ must be true in every model of $BK \wedge \overline{E^+}$, $\overline{H}$ must contain a subset of the ground literals in $\overline{\bot}$. Thus,

$$BK \wedge \overline{E^+} \models \overline{\bot} \models \overline{H}$$

and so

$$H \models \bot$$

Therefore, a subset of solutions for H can be found by considering the clauses Θ-subsuming $\bot$.

Suppose, for example $BK = animal(X) \leftarrow pet(X), pet(X) \leftarrow dog(X)$ and $E = nice(X) \leftarrow dog(X)$. In this case, $\bot = nice(X) \leftarrow dog(X), pet(X), animal(X)$.

In general, $\bot$ could have an infinite cardinality. Thus, *Mode Directed Inverse Entailment* (Muggleton 1995) systems such as Aleph and Progol, use modes declaration together with other settings to constrain the search for a good hypothesis. Mode declarations describes the relations between the arguments and their types and also defines if a predicate can be used in the head of the clause (*modeh*) or in the body of the clause (*modeb*). They can also constrain the number of different instantiations of a predicate in a clause, through the *recall number*. To do so, the arguments of a literal can be of three modes:

- Input(+)—an input variable of type T in a body literal B_i appears as an output variable of type T in a body literal that appears before B_i, or appears as an input variable of type T in the head of the clause.
- Output(−)—an output variable of type T in the head of the clause must appear as an output variable of type T in any literal of the body of the clause.
- Constant(#)—an argument denoted by $\#T$ must be ground with terms of type T.

Algorithm 4 illustrates the saturation phase of Progol (Muggleton 1995) system, corresponding to the bottom clause generation from a positive example. The list *InTerms* keeps the terms responsible for instantiating the input terms of the predicate in the head of the clause and the terms instantiating output terms in the body of the clause. The function *hash* associates a different variable to each term.

Suppose, for example, the mode declarations below, expressing a fatherhood relationship, where the first argument is the recall number

$modeh(1, father(+person, +person))$
$modeb(20, parent\_of(+person, +person))$
$modeb(20, parent\_of(-person, +person))$

and the background knowledge

$parent\_of(jack, anne)$
$parent\_of(juliet, anne)$
$parent\_of(jack, james)$
$parent\_of(juliet, james)$

Algorithm 4 Bottom clause construction Algorithm (Muggleton 1995)

1: let Ex be a positive example, where $\overline{Ex}$ is a clause normal form logic program $\overline{a} \wedge b_1 \wedge \cdots \wedge b_n$
2: let i be 0, corresponding to the variables depth
3: $BK \leftarrow BK \cup Ex$ (1)
4: $InTerms \leftarrow \emptyset$, $\bot \leftarrow \emptyset$ (2)
5: find the first modeh h such that h subsumes a with substitution θ (3)
6: **for each** v/t in θ **do**
7: **if** v is a $\sharp$ type **then**
8: replace v in h by t
9: **if** v is a $+$ or $-$ type **then**
10: replace v in h by v_k, where v_k is a variable such that $k = hash(t)$
11: **if** v is a $+$ type **then**
12: $InTerms \leftarrow InTerms \cup t$
13: $\bot \leftarrow \bot \cup h$
14: **for each** modeb declaration b **do**
15: **for** all possible substitution θ of arguments corresponding to $+$ type by terms in the set $InTerms$ **do**
16: **repeat**
17: **if** b succeeds with substitution θ' **then**
18: **for each** v/t in θ and θ' **do**
19: **if** v corresponds to $\sharp$ type **then**
20: replace v in b by t
21: **else**
22: replace v in b by v_k, where $k = hash(t)$
23: **if** v corresponds to $-$ type **then**
24: $InTerms \leftarrow InTerms \cup t$
25: $\bot \leftarrow \bot \cup \overline{b}$
26: **until** reaches recall times
27: $i \leftarrow i + 1$ (5)
28: Go to step (4) if the maximum depth of variables is not reached
return $\bot$.

and the example $father(jack, anne)$. The most specific clause is

$$\bot = father(jack, anne) \leftarrow parent\_of(jack, anne), parent\_of(juliet, anne)$$

where the first literal on the body was obtained by $modeb(20, parent\_of(+person, +person))$ and the second literal by $modeb(20, parent\_of(-person, +person))$.

The bottom clause above with the constants replaced by variables is

$$\bot = father(A, B) \leftarrow parent\_of(A, B), parent\_of(C, B)$$

To further reduce the search space of literals to be part of a bottom Clause, one can take advantage of determination declarations. Determination statements declare the predicates that can be used to construct a hypothesis (Srinivasan 2001). They take the form:

$$determination(TargetName/Arity, BackgroundName/Arity)$$

The first argument is the name and arity of the target predicate, that is, the predicate that will appear in the head of hypothesised clauses. The second argument is the name and arity of a predicate that can appear in the body of such clauses.

For more details on formal definitions of the Bottom Clause and Inverse Entailment we refer the reader to Muggleton (1995).

4 Using the bottom clause to search for antecedents when revising a FOL theory

Aiming to reduce search cost, we propose the following modifications to FORTE:

1. To use the variabilized bottom clause as search space of literals, which reduces the search space and also impose the following constraints:
   - to limit the maximum number of different instantiations of a literal (the recall number);
   - to limit the number of new variables in a clause;
   - to guarantee that at least one positive example is covered (the one which generates the bottom clause).
2. To declare modes to the arguments of a literal. As a result, the arguments are defined as input, output and constant.

We call the modified version of FORTE considering mode declarations and the bottom clause *FORTE_MBC*.

In order to generate the bottom clause in FORTE, we use the saturation phase, defined by Progol/Aleph. The bottom clause is created immediately before the search for antecedents begins, from a positive example covered by the currently revised clause (base clause). Note that as we are revising an existing theory, the body of the base clause is usually not empty. A covered positive example is selected because when specializing a clause the goal is to make the negative examples covered by the clause become unprovable while the originally provable positive examples to still be covered. Thus, naturally, the bottom clause must be composed of the relevant literals for at least one positive example covered by the clause. Also, in this way, the base clause is a subset of the bottom clause. The created bottom clause becomes the search space for antecedents, which improves the efficiency of the addition antecedents operation since it usually has many fewer literals than the previous space of the whole knowledge base. Additionally, we have the guarantee that at least one positive example continues covered by the generated literals, which was not guaranteed before by the top-down search for antecedents. Also, as the bottom clause is already variabilized it is not necessary to generate either new literals or new variables. Previously, such operation had an exponential cost according the arity of predicates. It is important to emphasize that the constraints of FOIL remain satisfied, as the arguments of the literals in the bottom clause must obey their types and there is a linking variable between the literal being added in the clause and the literals of the current clause.

4.1 Using the bottom clause in Hill Climbing add antecedents algorithm

The Hill Climbing algorithm modified to take into account the bottom clause as search space is detailed in Algorithm 5.

The algorithm starts with the clause considered as a revision point for adding antecedents. The goal of adding antecedents in a clause is to eliminate the existence of a proof of the negative examples while still keeping the proof of positive examples. Antecedents are added in a clause in two situations:

Algorithm 5 Hill Climbing Add Antecedents Algorithm Using the Bottom Clause

Input: C—a clause to be specialized; POS—the set of positive examples proved by C; NEG—the set of negative examples proved by C

Output: C'—one or more specialized versions of C

```
1: repeat
2:    Ex ← an example from POS
3:    BC ← bottom clause created from Ex
4:    unify the variables of BC with C
5:    repeat
6:       Ante ← best antecedent from BC
7:       if score(C + Ante) > score(C) then
8:          add Ante to C
9:       remove Ante from BC
10:   until there are no more antecedents in BC or it is not possible to improve the score of
      the current clause
11:   if the final clause is different from the original one then
12:      add the final specialized clause to C'
13: until it is not possible to specialize C or all the positive instances originally proved by
    C are proved by C'
```

1. The clause is used in a proof of negative examples. Antecedents are added to it aiming to eliminate the proof of negative examples.
2. The theory does not prove some positive examples and a clause is added to it so that such positive examples become provable. A clause might be added to the theory through two different ways:
   (a) A completely new clause is added to the theory. The head of the clause is the variabilized example and therefore it is proving all positive and negative examples with such a predicate. It is necessary to add antecedents to eliminate the proof of negative examples.
   (b) A clause is added to the theory from an existing clause. In order to prove positive examples it is necessary to delete the antecedents that prevent these examples from being proved. After that, non-proved negative examples may become provable. It is necessary to add antecedents to solve this problem.

The procedure begins by choosing a positive example currently covered by the clause being specialized. Then, it generates the bottom clause for this example using the background knowledge and the mode and determination declarations. As the clause being specialized may have a non-empty body, and the intersection of the bottom clause and such a body may be not empty, we must unify the variables of the bottom clause with the variables of the current clause. This is necessary to make the literals of the bottom clause identical to the literals of the clause under specialization. Then, the specialization of the clause starts and the literals of the bottom clause are evaluated so that the best one is chosen to be added in the clause. If the score of the current clause with this added literal is better than the score of the current clause, then this literal is added to the body of the clause. The process is over when it is not possible anymore to improve the score or there are no more antecedents left in the bottom clause. It is expected that the returned specialized clause does not cover any negative examples. At this moment, if there is an originally proved example which becomes unprovable, the algorithm tries to create a new specialized version of the clause. To do so, it restarts the process from the original clause also creating a new bottom clause. The whole

Algorithm 6 Relational Pathfinding Add Antecedents Algorithm Using the Bottom Clause

Input: C—a clause to be specialized; POS—the set of positive examples proved by C; NEG—the set of negative examples proved by C
Output: C'—specialized version of C'
1: $Ex1 \leftarrow$ an example from POS
2: $BC \leftarrow$ bottom clause created from $Ex1$
3: unify the variables of BC with C
4: $Ex2 \leftarrow$ an example from POS
5: search for relational paths between the arguments of $Ex2$ considering BC as search space
6: **for each** relational path found **do**
7: **PossibleClauses** $\leftarrow C +$ found relational path
8: $C' \leftarrow$ the best clause from **PossibleClauses**
9: **if** C' continue to prove negative examples **then**
10: add antecedents to C' using the Hill-climbing algorithm

procedure finishes when all positive examples originally provable are again provable by the specialized versions of the clause.

4.2 Using the bottom clause in Relational Pathfinding add antecedents algorithm

The Relational Pathfinding algorithm that adds more than one antecedents at once in a clause, using a search space constrained by the bottom clause, is exhibited in Algorithm 6.

Similarly to Algorithm 5, this procedure starts by choosing a positive example originally proved by the clause being specialized and generating a bottom clause for it. Also, the literals of the current clause and the bottom clause are unified, so that the current clause is a subset of the bottom clause. Then, the algorithm tries to find literals of the bottom clause which generate a relational path between the arguments of a another example. The best relational path is selected and if there are negative examples proved by this new clause, new antecedents will be added to it through the Hill Climbing algorithm explained in the last section, in an attempt to eliminate the proofs of those negative examples.

4.3 Using the bottom clause as a search space for antecedents generation

The new antecedent generation algorithm using bottom clause for Hill Climbing and Relational Pathfinding can be seen in Algorithms 7 and 8, respectively.

The first algorithm is responsible for returning a valid antecedent to be added in a clause during the course of Algorithm 5. Note that, here we consider one literal valid if it has at least one variable in common with the current clause. Thus, Algorithm 7 is executed as a loop in the step of line 6 of that algorithm, so that the valid antecedents are collected and only they are evaluated. The algorithm visits each literal of the bottom clause and returns only the ones which have a variable in common with the clause being specialized. The same goal is followed by Algorithm 8, but now the literal is returned to the Relational Pathfinding algorithm, and therefore it has to take into account the end values of the relational paths, as explained in Sect. 2.3.

4.4 Using the modes declaration to validate antecedents

Antecedents being added in the revised clause must be validated according to the specified mode declarations in the context of the revised clause. These operations are necessary as

Algorithm 7 Hill Climbing Antecedent Generation Algorithm Using the Bottom Clause

Input: C, a clause being specialized, BC, a bottom clause
Output: *Ante*, an antecedent to be added in C

```
1: for each literal in BC do
2:    collect the variables of C
3:    collect the variables of the particular literal
4:    if the literal has a variable in common with C then
5:       return the literal
6:    else
7:       Go to the next literal in BC
```

Algorithm 8 Relational Pathfinding Antecedent Generation Algorithm Using the Bottom Clause

Input: C, a clause being specialized, BC, a bottom clause
Output: *Ante*, an antecedent to be added in C

```
1: for each literal in BC do
2:    collect the end values with the respective variables they represent in the node been
      expanded
3:    collect the variables of the particular literal
4:    if the literal has a variable in common with the variables bound to the end values
      then
5:       Instantiate the variables in the literal that represent some end value
6:       Search in the background knowledge a fact that unify with the literal
7:       if a fact is found then
8:          return the literal
9:       else
10:         Go to the next literal in BC
11:   else
12:      Go to the next literal in BC
```

compliance with mode declarations within the bottom clause does not imply compliance with it in the context of the revised clause. For example, an input occurrence of a variable may be preceded by an output occurrence of it in another literal of the bottom clause, but there may be no such previous output occurrence found in the revised clause.

In the case of *Hill Climbing algorithm*, only one antecedent is added at once and therefore, before scoring the antecedent, the algorithm verifies if it obeys the modes declaration taking into account the current clause. This implies a change of line 4 of Algorithm 7. Now, instead of only verifying if the literal has a variable in common with the base clause, it is also necessary to verify if such literal obeys the modes definitions. Thus, fewer literals are evaluated which decreases the cost of the addition antecedents process.

In the *Relational Pathfinding* algorithm, on the other hand, the antecedents cannot be validated just after they are picked from the bottom clause, since more than one antecedent will be added at once and therefore the mode declarations can be valid in the whole path but not in just one of the antecedents. Thus, the whole path is validated according to mode declarations. If the relational path does not obey the mode declarations it is discarded, which makes many fewer clauses be evaluated and consequently decreases the runtime of the addition of antecedents. This operation does not imply a direct change on Algorithm 8, since the path is

validated according to the modes only after it is completely created. However, now, inside the cycle of line 6 of Algorithm 6, it is necessary to verify if the path plus the clause obeys the mode declarations, just after line 7.

5 Experimental results

We performed a set of experiments in order to verify the benefits of revising FOL theories using the bottom clause and mode declarations. Specifically, we would like to answer the following questions:

1. Is FORTE_MBC capable of producing theories at least with the same accuracy and size as FORTE in a significantly smaller runtime?
2. Is FORTE_MBC able to keep the advantage of theory revision and therefore achieve higher accuracies than inductive methods, without significantly compromising in terms of the size of generated theories and runtime?
3. If FORTE_MBC is used to learn a theory from scratch (since ILP is a particular case of theory revision as discussed in Sect. 2), will it achieve higher accuracies than a traditional ILP system, without significantly compromising in terms of the size of generated theories and runtime?

General experimental methodology We applied K-fold stratified cross validation to split the input data into disjoint training and test sets. Each fold keeps the original distribution of positive and negative examples (Kohavi 1995). The significance test used was the corrected two-tailed paired t-test (Nadeau and Bengio 2003), with $p < 0.05$. All the experiments were run on Yap Prolog (Santos Costa 2008).

5.1 Comparison to the original FORTE revision system

Datasets To answer the first question, we consider the Alzheimer (King et al. 1995) domain, which compares 37 analogues of Tacrine, a drug combating Alzheimer's disease, according to four properties as described below, where each property originates from a different dataset:

1. inhibit *amine* re-uptake, composed of 343 positive examples and 343 negative examples
2. low *toxicity*, with 443 positive examples and 443 negative examples
3. high acetyl *cholinesterase* inhibition, composed of 663 positive examples and 663 negative examples and
4. good reversal of *scopolamine*-induced memory deficiency, containing 321 positive examples and 321 negative examples.

We also considered the DssTox dataset (Fang et al. 2001), extracted from the EPA's DSSTox NCTRER Database. It contains structural information about a diverse set of 232 natural, synthetic and environmental estrogens and classifications with regard to their binding activity for the estrogen receptor. We follow the experiments reported in Landwehr et al. (2007) and use only structural information, that is, atom elements and bonds. We also provided a relation $linked(A1, A2, E, BT)$ in the background knowledge that represents that there is a bond of type BT from atom $A1$ to atom $A2$ and $A2$ is of element E. This was done to reduce the lookahead problem for greedy search algorithms. Following the cited source, we define the Aleph parameters $noise = 10$ and $clauselength = 10$.

Table 1 Runtime in seconds, predictive accuracy and size in number of literals of the revised theories, using the Hill Climbing algorithm for adding antecedents in a clause

Datasets	Initial accuracy (%)	Original FORTE			FORTE_MBC		
		Runtime (s)	Accuracy (%)	Size	Runtime (s)	Accuracy (%)	Size
Amine	62.67	470	70.81◇	80.1	25.53⋆	68.96◇	34.5⋆
Toxic	74.02	254	74.35	26.9	15.85	75.64	27.1
Choline	56.02	9672.9	65.61◇	115.5	150.63⋆	62.52◇	40.2⋆
Scopo	55.59	4672.4	64.46◇	137	72.11⋆	65.25◇	45.8⋆
DssTox	49.69	1.68	50.05	11.1	12.05⋆	74.52	13.6 ◇ ⋆

Experimental methodology To avoid overfitting during the revision process, similarly to Baião et al. (2003), we applied K-fold stratified cross validation to split the input data into disjoint training and test sets and, within that, a t-fold stratified cross-validation to split training data into disjoint training and tuning sets. The revision algorithm monitors the error with respect to the tuning set after each revision, always keeping a copy of the theory with the best tuning set accuracy, and the saved "best-tuning-set-accuracy" theory is applied to the test set. The initial theories were obtained from Aleph using its default parameters, except for DssTox, as explained before. To generate such theories, the whole dataset was considered but a 10-fold cross validation procedure was used. Thus, a different theory was generated for each fold and each one of these theories was revised considering its respective fold (the same fold is used to generate and revise the theories). We compared the runtime, the predictive accuracies and the size of the theories in the number of literals returned by original FORTE and FORTE_MBC. Each value in the tables is the average of the 10 folds of cross validation, where within those a 5-folds internal cross validation was applied. We also show in these tables the predictive accuracies of the initial theories, so that it is possible to see the significant improvement on accuracy achieved by the revision process. Both Hill Climbing and the Relational Pathfinding algorithms were considered, running independently. The runtime is the total time necessary to execute Algorithm 1 (including the bottom clause generation).

Results The results are presented in Tables 1 and 2, where the symbol ◇ expresses the cases where the final predictive accuracy is significantly better than the initial accuracy and the symbol ⋆ indicates that the difference between FORTE_MBC and the original FORTE is significant. DssTox is not included on Table 2, because the Relational Pathfinding algorithm cannot be applied to it since the target predicate has arity one.

The tables show that FORTE_MBC speeds up significantly the runtime of the revision process and also returns more comprehensible theories. Additionally, we can see that the initial accuracies are improved by both systems and there are no significant difference between the accuracies obtained by FORTE and FORTE_MBC. There are three exceptions: (1) when running the Hill Climbing algorithm for Toxic dataset, there is no significative difference between the size of the theories returned by FORTE and FORTE_MBC and the runtime of both systems, (2) when running the Hill Climbing algorithm the initial accuracy of Toxic dataset is not improved by both revision systems (3) Forte_MBC obtained an accuracy significantly better than original Forte in DssTox dataset, due to the inability of the original revision system to use constants, which are essential to this problem. The symbol

Table 2 Runtime in seconds, predictive accuracy and size in number of literals of the revised theories, using the Relational Pathfinding algorithm for adding antecedents in a clause

Datasets	Initial accuracy (%)	Original FORTE			FORTE_MBC		
		Runtime (s)	Accuracy (%)	Size	Runtime (s)	Accuracy (%)	Size
Amine	62.67	5421.43	72.49◇	175	72.34★	72.46◇	51.1★
Toxic	74.02	6065.2	81.91◇	151.2	42.27★	80.22◇	41.3★
Choline	56.02	>648000	?	?	622.24★	68.17◇	71.3
Scopo	55.59	18640.38	71.85◇	272.49	444.4★	65.39◇	81.6★

"?" in Table 2 represents the case where the algorithm ran for more than 180 hours without finishing even one fold.

Considering the Hill Climbing algorithm for adding antecedents, the biggest speedup was of 65× in Scopo dataset, the smallest speedup was of 16× in Toxic dataset and on average we obtained the speedup of 57×. On the other hand, the speedups for the Relational Pathfinding algorithms were much higher, since the latter algorithm is more computationally complex than Hill Climbing. Thus, the biggest speedup was of 143× in Toxic Dataset, the smallest speedup was of 68× in Scopolamine dataset and on average the speedup was of 78×. Note that we are not considering the Choline dataset, since we do not have the exact runtime value for the original algorithm (the lower bound on the speedup is thus 1000×). Moreover, we observed an average reduction of 2.2× in the size of the returned theories when running the FORTE_MBC system.

5.2 Comparison to inductive methods

In order to answer the second and third question we continue to consider the Alzheimer domain and DssTox datasets and also the initial theories learned from scratch using Aleph. First, we compare the revision performed by FORTE_MBC to that produced by the covering algorithm followed by Aleph. Next, we compare the induction performed by both systems in order to answer the third question. Finally, we revisit the second question to compare the revision performed by FORTE_MBC to that produced by a non-covering induction approach.

5.2.1 Comparing the revision process performed by FORTE_MBC to a covering approach

Here we want to verify if revising theories produces more accurate results than learning theories from scratch using a covering approach. To do so, we considered the Alzheimer domain and DssTox.

5.2.1.1 Using Aleph default parameters

Methodology Here we do not consider DssTox as it requires modifications of Aleph parameters. In order to verify the improvement in accuracy obtained by the revision system, the initial theories were obtained from Aleph by using various proportions of the example set. However, the whole example set was used for revising the theories as well as for the from-scratch induction by Aleph. The plotted accuracies are thus constant for the latter mode of learning, while they vary for the revision experiments pertaining to different initial theo-

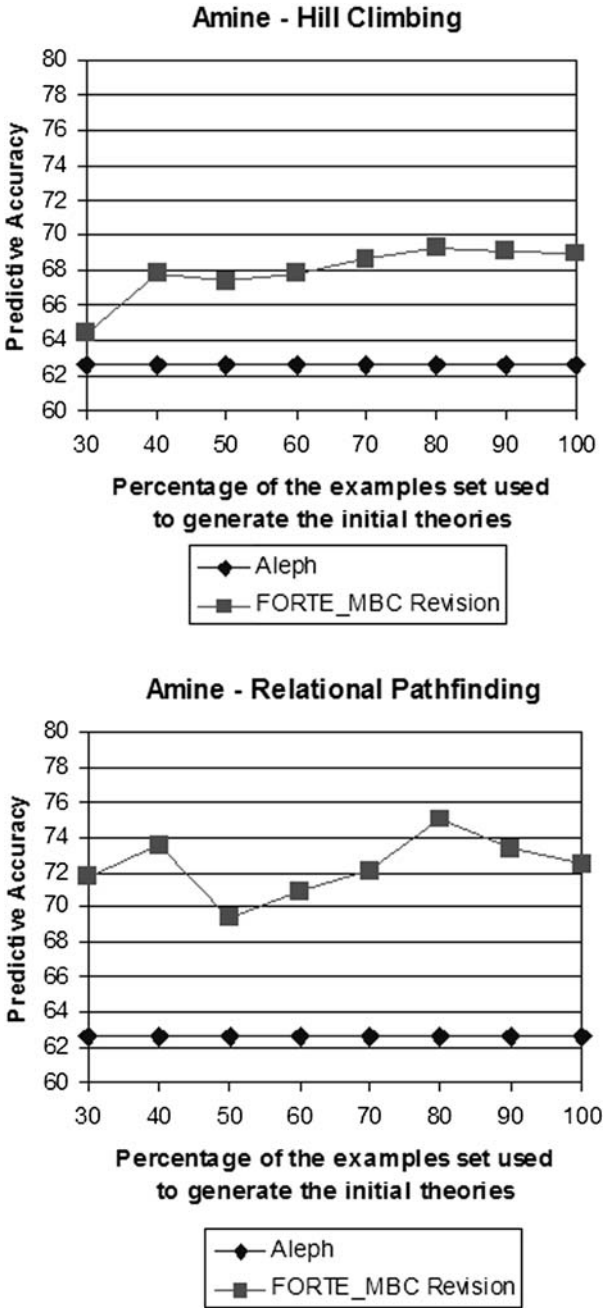

Fig. 3 Aleph X FORTE_MBC revision in Amine dataset

ries. The points on the curves represent the average predictive accuracies for 10-fold cross validation for both revision and induction.

Results We show in Figs. 3, 4, 5 and 6 the results obtained using both Hill Climbing and Relational Pathfinding algorithms for adding antecedents.

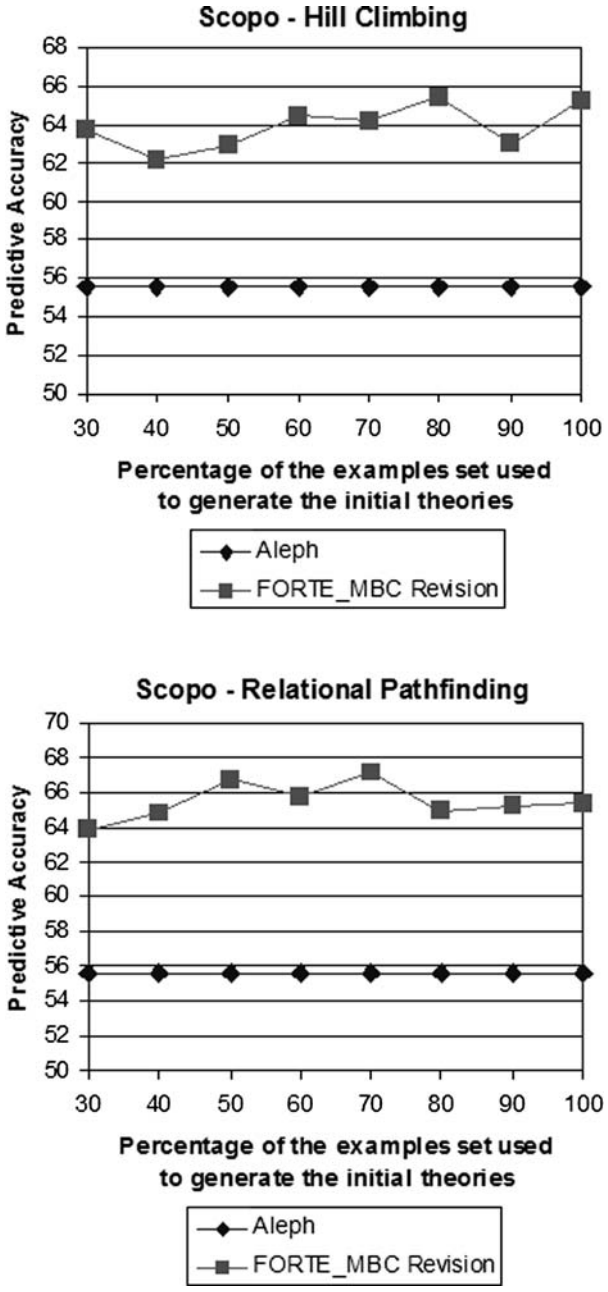

Fig. 4 Aleph X FORTE_MBC revision in Scopolamine dataset

As we can observe from the figures, the accuracies obtained by FORTE_ MBC are always significantly higher than accuracies obtained by Aleph, considering both algorithms for adding antecedents, except for the Toxic dataset using the Hill Climbing algorithm.

In Tables 3 and 4 below we compare the runtime and size of final theories obtained by revision with FORTE_MBC and induction with Aleph, considering both the Hill Climbing and

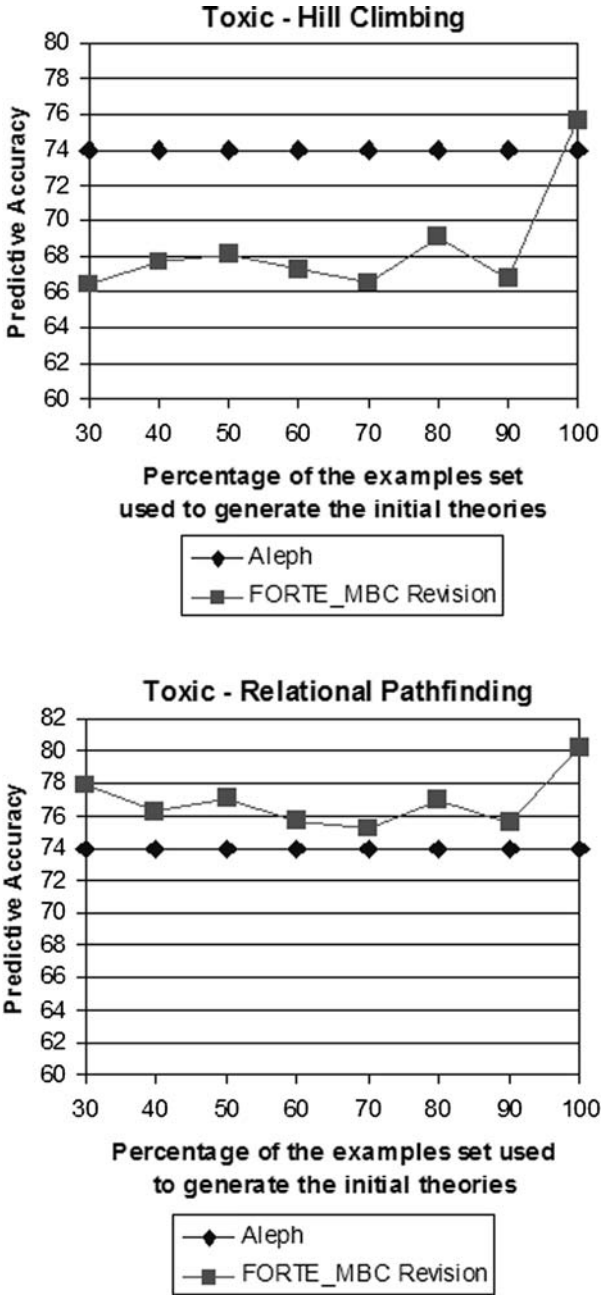

Fig. 5 Aleph X FORTE_MBC revision in Toxic dataset

Relational Pathfinding algorithms, respectively. We are considering only the last point of the curves above, i.e., the case where the initial theories were induced from the whole dataset. As Aleph does not have a built-in internal cross validation procedure as FORTE_MBC, we show the results with and without the internal cross validation procedure described in Sect. 5.1. The symbol $\star$ indicates the cases where Aleph performed significantly better than

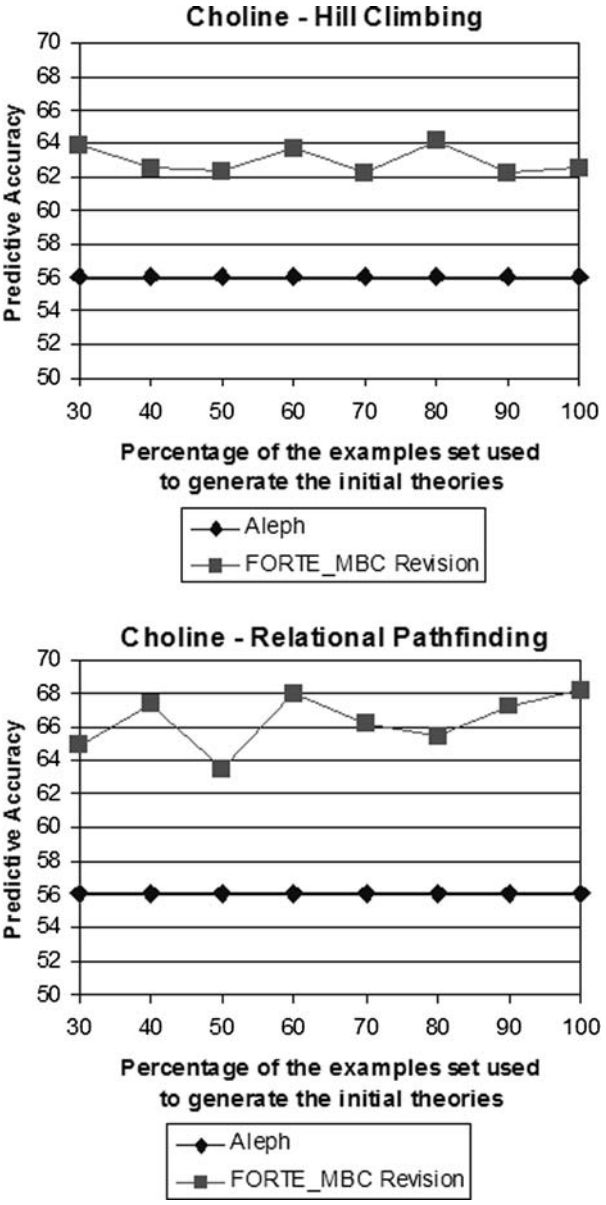

Fig. 6 Aleph X FORTE_MBC revision in Choline dataset

FORTE_MBC with internal cross validation and the symbol ◇ indicates the cases where Aleph performed significantly better than FORTE_MBC without internal cross validation.

Although the revision process is able to achieve much better accuracies than the covering approach, as we can see from the tables the runtime and size results do not follow this pattern, since, except for the Amine dataset when FORTE_MBC ran without internal cross validation, Aleph reached significantly better results than FORTE_MBC.

Table 3 Runtime in seconds and Size in number of literals obtained from Aleph and FORTE_MBC with and without internal cross validation, using Hill Climbing algorithm for adding antecedents in a clause

Datasets	Aleph		FORTE_MBC with internal CV		FORTE_MBC without internal CV	
	Runtime (s)	Size	Runtime (s)	Size	Runtime (s)	Size
Amine	14.84⋆	20.8⋆	25.53	34.5	15.93	23.8
Toxic	11.49 ⋆ ◇	24.7 ⋆ ◇	15.85	27.1	23.44	31
Choline	38.17 ⋆ ◇	30.7 ⋆ ◇	150.63	40.2	93.88	39.7
Scopo	15.31 ⋆ ◇	28.9 ⋆ ◇	72.11	45.8	85.68	38.6

Table 4 Runtime in seconds and Size in number of literals obtained from Aleph and FORTE_MBC with and without internal cross validation, using Relational Pathfinding algorithm for adding antecedents in a clause

Datasets	Aleph		FORTE_MBC with internal CV		FORTE_MBC without internal CV	
	Runtime (s)	Size	Runtime (s)	Size	Runtime (s)	Size
Amine	14.84 ⋆ ◇	20.8 ⋆ ◇	72.34	51.1	104.49	41.7
Toxic	11.49 ⋆ ◇	24.7 ⋆ ◇	42.27	41.3	84.7	36.3
Choline	38.17 ⋆ ◇	30.7 ⋆ ◇	622.24	71.3	463.67	59.9
Scopo	15.31 ⋆ ◇	28.9 ⋆ ◇	272.49	81.6	216.73	53.2

5.2.1.2 *Changing the clauselength parameter in Aleph*

Methodology In the results above, we rely on Aleph's default parameters to avoid introducing bias by their choice. In most of the cases, we have a fair comparison between both systems, since the setting of FORTE_MBC (without internal cross validation) is similar to the Aleph default parameters. For example, the default value of the Aleph parameter *noise* is 0,[1] while FORTE_MBC does not consider *noise*, always trying to find a theory covering none of the negative examples. On the other hand, the default value of the Aleph parameter *nodes* is 5000,[2] which is a number large enough for the problems presented here. FORTE_MBC has no limit on the number of nodes explored when searching for a clause. However, the parameter *clauselength* is able to produce different results in the datasets, since its default value is small (4) and in most of the cases FORTE returns clauses larger than 4 literals because it has no limit on the size of clauses. Therefore, we run Aleph considering $clauselength = 10$ to simulate the behavior of FORTE.

Results The results are presented in Tables 5 and 6, the first one considering the Hill Climbing algorithm and the second one considering Relational Pathfinding. The first table also includes the results for DssTox, since when running this dataset the *clauselength* parameter is always 10 (and also $noise = 10$). The symbol ⋆ expresses the cases where there

[1]The noise parameter defines an upper bound on the number of negative examples allowed to be covered by an acceptable clause.

[2]The nodes parameter sets an upper bound on the number of nodes to be explored when searching for an acceptable clause.

Table 5 Runtime in seconds and Size in number of literals obtained from Aleph with *clauselength* parameter set to 10 and FORTE_MBC with and without internal cross validation, using Hill Climbing algorithm for adding antecedents in a clause

Datasets	Aleph			FORTE_MBC with internal CV			FORTE_MBC without internal CV		
	Acc	Run (s)	Size	Acc	Run (s)	Size	Acc	Run (s)	Size
Amine	69.7	326.03	54.1 ⋆ ◇	78.41⋆	168.36⋆	69.5	74.97◇	119.66◇	66.3
Toxic	68.26	408.9	61.9◇	72.77	275.4⋆	69.2	76.73◇	303.2	88.7
Choline	60.85	1014.02	76.5 ⋆ ◇	65.75⋆	1155.23	91.7	64.48◇	1008.1	96.2
Scopo	59.36	370.94	61.8 ⋆ ◇	65.26⋆	266.38⋆	79.9	65.88◇	220.58◇	81.1
DssTox	49.69	222.32	10.5⋆	74.52⋆	12.5⋆	13.6	78.43◇	13.37◇	13.8

Table 6 Runtime in seconds and Size in number of literals obtained from Aleph with *clauselength* parameter set to 10 and FORTE_MBC with and without internal cross validation, using Relational Pathfinding algorithm for adding antecedents in a clause

Datasets	Aleph			FORTE_MBC with internal CV			FORTE_MBC without internal CV		
	Acc	Run (s)	Size	Acc	Run (s)	Size	Acc	Run (s)	Size
Amine	69.7	326.03	54.1 ⋆ ◇	74.82⋆	132.79⋆	77.3	77.53◇	241.73	76.2
Toxic	68.26	408.9	61.9 ⋆ ◇	78.63⋆	455.5	83.4	76.76◇	511.85	93.9
Choline	60.85	1014.02	76.5 ⋆ ◇	67.12⋆	1610.84	105.8	64.85◇	1820.24	114.9
Scopo	59.36	370.94⋆	61.8 ⋆ ◇	63.52	692.8	111.3	66.04◇	175◇	89.7

is a significant difference between Aleph and FORTE_MBC with internal cross validation and the symbol ◇ expresses the cases where there is a significant difference between Aleph and FORTE_MBC without internal cross validation. We can conclude from the tables that Aleph had its accuracies improved in most of the cases, but the revision also took advantage of the higher initial theories, reaching better accuracies than previous results based on the *clauselength* default. The quality of initial theories thus clearly correlates with the quality of their revised counterparts. Additionally, the revision process executed faster than the induction in most of the cases and the size of theories returned from both systems is comparable.

5.2.2 Comparing FORTE_MBC induction to a covering approach

In order to answer the last question, we compare the runtime, predictive accuracy and size of generated theories using FORTE_MBC induction and Aleph induction in the Alzheimer and DssTox domains cited above. Both Hill Climbing and Relational Pathfinding algorithms were considered.

Methodology We start the experiments using Aleph with default parameters. The values are obtained from a 10-fold cross validation procedure. FORTE_MBC was run using the internal cross validation procedure described in Sect. 5.1 and also without it.

Results The results are presented in Tables 7 and 8. The symbol ⋆ expresses the cases where there is a significant difference between Aleph and FORTE_MBC with internal

Table 7 Runtime in seconds, predictive accuracy and size of the theories when learning from scratch, using Hill Climbing algorithm for adding antecedents in FORTE_MBC

Datasets	Aleph			FORTE_MBC with internal CV			FORTE_MBC without internal CV		
	Runn	Acc	Size	Run	Acc	Size	Run	Acc	Size
Amine	14.84	62.67	20.8	2.97⋆	67.97⋆	7.2⋆	5.73◇	67.82◇	15◇
Toxic	11.49	74.02 ⋆ ◇	24.7	4.26⋆	67.6	8⋆	4.61◇	68.63	10.3◇
Choline	38.17	56.02	30.7	8.67⋆	63.05⋆	10.5⋆	10.22◇	63.2◇	11.5◇
Scopo	15.31	55.59	28.9	3.42⋆	61.66⋆	10.2⋆	3.37◇	55.44	6.6◇

Table 8 Runtime in seconds, predictive accuracy and size of the theories when learning from scratch, using Relational Pathfinding algorithm for adding antecedents in FORTE_MBC

Datasets	Aleph			FORTE_MBC with internal CV			FORTE_MBC without internal CV		
	Run	Acc	Size	Run	Acc	Size	Run	Acc	Size
Amine	14.84⋆	62.67	20.8⋆	35.81	73.66⋆	41.3	98.63	71.65◇	35.6
Toxic	11.49	74.02	24.7	47.28	77.29	27.3	55.84	71.89	23.9
Choline	38.17	56.02	30.7	65.16	62.37⋆	35	183.02	61.23◇	18.9
Scopo	15.31◇	55.59	28.9	44.03	64.64⋆	32.8	56.36	63.84◇	19

cross validation and the symbol ◇ expresses the cases where there is a significant difference between Aleph and FORTE_MBC without internal cross validation. Each value in the FORTE_MBC column is the average of the 10 folds of cross validation.

As we can see from tables, when using the Hill Climbing algorithm, FORTE_MBC is significantly better than Aleph, for runtime, theory size and accuracy in most of the cases. The biggest speedup was of 44.6× in the DssTox dataset without internal cross validation, and the smallest speedup was of 2.5× in the Toxic dataset without internal cross validation. On average, a speedup of 10.5× was obtained. When using the Relational Pathfinding algorithm the size of theories generated by FORTE_MBC has no significant difference compared to Aleph (except for the Amine dataset), but the accuracy of theories generated by FORTE_MBC are significantly better than the ones generated by Aleph (except for Toxic). However, the runtime of Aleph is significantly better than FORTE_MBC in half of the cases, just because Relational Pathfinding is a very expensive algorithm.

Additionally, we show in Tables 9 and 10 the results of experiments considering *clauselength* = 10 when running Aleph. FORTE_MBC results are the same as the previous table, since we do not take the initial theories into account when running this system from scratch. When running DssTox, the *noise* parameter is set to 10. Note that with this new setting, the accuracy of both systems are not significantly different, except for the Toxic Dataset, whose accuracy was significantly better in FORTE_MBC system. However, the runtime of Aleph system is much bigger than when using the *clauselength* default value. The same is true for the size of returned theories.

Table 9 Runtime in seconds, predictive accuracy and size of the theories when learning from scratch, using Hill Climbing algorithm for adding antecedents in FORTE_MBC and *clauselength* = 10 when running Aleph

Datasets	Aleph			FORTE_MBC with internal CV			FORTE_MBC without internal CV		
	Run	Acc	Size	Run	Acc	Size	Run	Acc	Size
Amine	326.03	69.7	54.1	2.97⋆	67.97	7.2⋆	5.73◇	67.82	15◇
Toxic	408.9	68.26	61.9	4.26⋆	67.6	8⋆	4.61◇	68.63	10.3◇
Choline	1014.02	60.85	76.5	8.67⋆	63.05	10.5⋆	10.22◇	63.2	11.5◇
Scopo	370.94	59.36◇	61.8	3.42⋆	61.66	10.2⋆	3.37◇	55.44	6.6◇
DssTox	222.32	49.69	10.5	9.55⋆	78⋆	7.8⋆	4.97◇	78.43◇	4◇

Table 10 Runtime in seconds, predictive accuracy and size of the theories when learning from scratch, using Relational Pathfinding algorithm for adding antecedents in FORTE_MBC and *clauselength* = 10 when running Aleph

Datasets	Aleph			FORTE_MBC with internal CV			FORTE_MBC without internal CV		
	Run	Acc	Size	Run	Acc	Size	Run	Acc	Size
Amine	326.03	69.7	54.1	35.81⋆	73.66	41.3	98.63◇	71.65	35.6◇
Toxic	408.9	68.26	61.9	47.28⋆	77.29⋆	27.3⋆	55.84◇	71.89	23.9◇
Choline	1014.02	60.85	76.5	65.16⋆	62.37	35⋆	183.02◇	61.23	18.9◇
Scopo	370.94	59.36	61.8	44.03⋆	64.64⋆	32.8⋆	56.36◇	63.84	19◇

5.2.3 Comparing the revision process performed by FORTE_MBC to a non-covering inductive approach

Now we would like to observe the performance of theory revision compared to an algorithm inducing a theory from scratch, but without using a covering approach. Thus, we compare in this section FORTE_MBC learning a theory from scratch and FORTE_MBC revising theories. In order to generate the initial theories provided to the revision task, we considered two approaches:

1. The initial theories are learned by an ILP system.
2. The initial theories are provided by domain experts.

Initial theories learned by an ILP system

Methodology In order to obtain the results presented in this section, FORTE_MBC revised theories generated by Aleph with the whole set of examples from Alzheimer domain and DssTox dataset, and then we compare it to the results of FORTE_MBC inducing theories from these same domains, from scratch. The internal cross validation procedure was applied on both systems.

Results The results can be seen in Tables 11 and 12. The symbol ⋆ indicates the statistically significant values.

For both add antecedents algorithms, FORTE_MBC learning from scratch achieves better results in runtime and size of theories in most of the cases. However, note that the initial

Table 11 Runtime in seconds, predictive accuracy and size of the theories in number of literals for FORTE_MBC revising and learning from scratch, both using Hill Climbing algorithm for adding antecedents in a clause

Datasets	FORTE_MBC revising			FORTE_MBC learning from scratch		
	Runtime (s)	Accuracy (%)	Size	Runtime (s)	Accuracy (%)	Size
Amine	25.53	68.96	34.5	2.97⋆	67.97	72⋆
Toxic	15.85	75.64⋆	27.1	4.26⋆	67.6	8⋆
Choline	150.63	62.52	40.2	8.67⋆	63.05	10.5⋆
Scopo	72.11	65.25	45.8	3.42⋆	61.66	10.2⋆
DssTox	12.05	74.52	13.6	9.55	78	7.8⋆

Table 12 Runtime in seconds, predictive accuracy and size in number of literals for FORTE_MBC revising and learning from scratch, both using Relational Pathfinding algorithm for adding antecedents in a clause

Datasets	FORTE_MBC revising			FORTE_MBC learning from scratch		
	Runtime	Accuracy	Size	Runtime	Accuracy	Size
Amine	72.34	72.46	51.1	35.81	73.66	41.3
Toxic	42.27	80.22⋆	41.3	47.28	77.29	27.3
Choline	622.24	68.17⋆	71.3	65.16⋆	62.37	35⋆
Scopo	272.49	65.39	81.6	44.03⋆	64.64	32.8⋆

Table 13 Runtime in seconds, predictive accuracy and size of the theories in number of literals for FORTE_MBC revising and learning from scratch, both using Hill Climbing algorithm for adding antecedents in a clause. The theories provided to the revision process were obtained from Aleph with *clauselength* parameter set to 10

Datasets	FORTE_MBC revising			FORTE_MBC learning from scratch		
	Runtime (s)	Accuracy (%)	Size	Runtime (s)	Accuracy (%)	Size
Amine	168.36	78.41⋆	69.5	2.97⋆	67.97	7.2⋆
Toxic	275.4	72.77⋆	69.2	4.26⋆	67.6	8⋆
Choline	1155.23	65.75	91.7	8.67⋆	63.05	10.5⋆
Scopo	266.38	65.26	79.9	3.42⋆	61.66	10.2⋆
DssTox	12.5	74.52	13.6	9.55	78	7.8⋆

theories provided to the revision systems were induced by Aleph, which follows a greedy covering approach, where the best local clause found at each cycle is aggregated to the theory being induced. The covering approach tends to produce theories with a large number of clauses, each representing a possible revision point, thus making the subsequent revision process computationally difficult. The large size and low quality of initial theories achieved by the covering procedure is also an explanation for the subsequent low quality of the final revised theories, compared to theories achieved by the revision algorithm starting from scratch.

Additionally, we show the results of experiments where the initial theories provided to FORTE_MBC revision were generated by Aleph with *clauselength* = 10. The results are presented in Tables 13 and 14.

Table 14 Runtime in seconds, predictive accuracy and size in number of literals for FORTE_MBC revising and learning from scratch, both using Relational Pathfinding algorithm for adding antecedents in a clause. The theories provided to the revision process were obtained from Aleph with *clauselength* parameter set to 10

Datasets	FORTE_MBC revising			FORTE_MBC learning from scratch		
	Runtime	Accuracy	Size	Runtime	Accuracy	Size
Amine	132.79	74.82	77.3	35.81⋆	73.66	41.3⋆
Toxic	455.5	78.63	83.4	47.28⋆	77.29	27.3⋆
Choline	1610.84	67.12⋆	105.8	65.16⋆	62.37	35⋆
Scopo	692.8	63.52	111.3	44.03⋆	64.64	32.8⋆

The results shown in this section indicate that applying a revision algorithm on theories initially produced by a standard covering algorithm does not result in better theories than those produced by a revision algorithm starting with an empty theory.

Using approximately correct initial theories given by domain experts Here we want to verify if revising approximately correct theories guides to more accurate results than learning theories from scratch. FORTE_MBC revising initial theories was then compared to FORTE_MBC inducing theories from scratch.[3] Two datasets were considered: The classical family domain (Quinlan 1990; Richards and Mooney 1995) and a dataset from a Distributed Database problem, the 007 benchmark DDOODB (Baião et al. 2003, 2004).

Family domain The Family domain contains 744 positive examples and 1479 negative examples, related to 12 family relationship concepts: *wife, husband, mother, father, son, daughter, brother, sister, uncle, aunt, niece and nephew*. The target theory also contains 2 intermediate predicates: *aunt_or_uncle and sibling*.

Methodology To analyze the results with this dataset, we first introduced the following 3 random errors in the target theory:

1. The literal *aunt_or_uncle*(A, B) was deleted from the rule defining the concept *uncle*;
2. The literal *married*(A, B) was replaced by *married*(A, C) in the rule defining the concept *husband*;
3. The literal *parent*(A, B) was removed from the rule defining the concept *father*.

We run theory revision using a 10-fold cross validation procedure encompassing an internal 5-fold cross validation procedure. In the learning curve, the example set size varies from 150 to 400 and each point in the curve is the average predictive accuracy of the 10 folds of cross validation. We do not present the results for Relational Pathfinding algorithm since the Hill Climbing algorithm already achieved 100% of accuracy. The curves can be seen in Fig. 7.

From the curves we can see that the predictive accuracy obtained by revision through FORTE_MBC is always significantly better than the induction using FORTE_MBC. These results show that when the initial theory is approximately correct the revision process achieves better accuracies than the induction process, since the former takes advantage of the initial theory as a starting point, while the latter starts its process from scratch.

[3]We do not run Aleph in these experiments because both datasets contain multiple-predicate to be learned and Aleph is a single predicate learner.

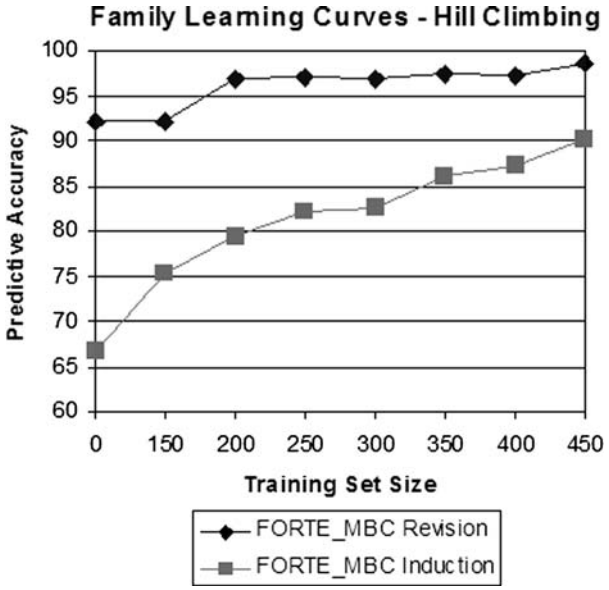

Fig. 7 Learning curves of Family dataset using Hill Climbing algorithm

Additionally, Fig. 8 shows the curves pertaining to introduction of random errors. There, 2 to 10 random errors were introduced in the target theory and such corrupted theories were revised using 200 examples from the main dataset. The same examples were used to learn theories from scratch using FORTE_MBC. Five types of errors were introduced:

1. Deletion of rules: for instance the rule defining the concept *niece* was removed from the theory;
2. Addition of rules: for instance, the rule $mother(A, B) \leftarrow parent(B, A)$, wrongly defining the concept *mother* was introduced in the theory;
3. Deletion of antecedents: for instance, the rule $father(A, B) \leftarrow parent(A, B)$, $gender(A, male)$ was replaced by the rule $father(A, B) \leftarrow gender(A, male)$;
4. Change of antecedents (delete plus add): for instance the rule $uncle(A, B) \leftarrow gender(A, male), aunt\_uncle(A, B)$ was replaced by $uncle(A, B) \leftarrow gender(A, male), parent(A, B)$;
5. Change of variables: for instance the rule $husband(A, B) \leftarrow gender(A, male), married(A, B)$ was replaced by $husband(A, B) \leftarrow gender(A, male), married(A, C)$.

As we can see from the figures, increasing the number of errors in the initial theory do lower the accuracy of FORTE_MBC's revised theories for a given training set size. However, the theories returned by FORTE_MBC revision are still always significantly better then the ones returned by FORTE_MBC induction up to 10 errors. So, when the theory is approximately correct it is more effective to revise them than to learn them from scratch, a result also obtained in Richards and Mooney (1995).

DDOODB dataset Finally, we considered an initial theory concerning the design of distributed databases on the 007 benchmark (Baião et al. 2003, 2004), with 19 positive examples and 29 negative examples related to three concepts: to choose a Vertical Fragmentation, to choose a Primary Horizontal Fragmentation and to choose a Derived Horizontal Fragmentation.

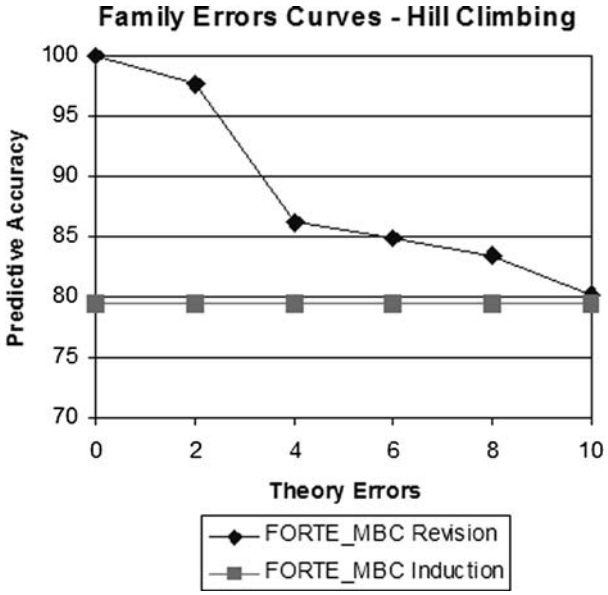

Fig. 8 Errors curves of Family dataset using Hill Climbing algorithm

Methodology In this dataset we considered $k = 12$ and $t = 4$ and the Hill Climbing algorithm for adding antecedents, following the methodology of the original paper. We plot learning curves for the average predictive accuracies of FORTE_MBC revising the initial theory defined by a Database expert and FORTE_MBC learning the theory from scratch, both cases considering the same folds of the examples.

The curves are presented in Fig. 9. It is possible to see that the revising curve is always above the learning curve, and the results of the revision are significantly better than the results of induction. This shows the importance of the initial theory, specially when facing a small set of examples.

From the experiments of this section, we can conclude that it is possible to achieve better predictive accuracies when revising an approximately correct theory instead of inducing it from scratch.

6 Conclusions and future work

With the availability of large-scale resources of background knowledge in areas such as biology (Muggleton 2005) there is an increasing need for development of efficient theory revision ILP systems (Dietterich et al. 2008). This work contributes towards this goal of bringing a theory revision system to be as efficient as a state of the art ILP system and achieving significantly higher accuracies than an ILP covering system.

The revision system considered in this work was FORTE (Richards and Mooney 1995). The antecedent addition operation of FORTE follows the top-down approach of FOIL, which leads to a huge search space dominating the cost of the revision process; moreover, it does not properly explore the connections among variables. In this work the efficiency of theory revision antecedent addition was improved by the introduction of the bottom clause (Muggleton 1995) to define the search space of antecedents for both algorithms Hill Climbing and Relational Pathfinding, and the introduction of mode declarations, to define which literals

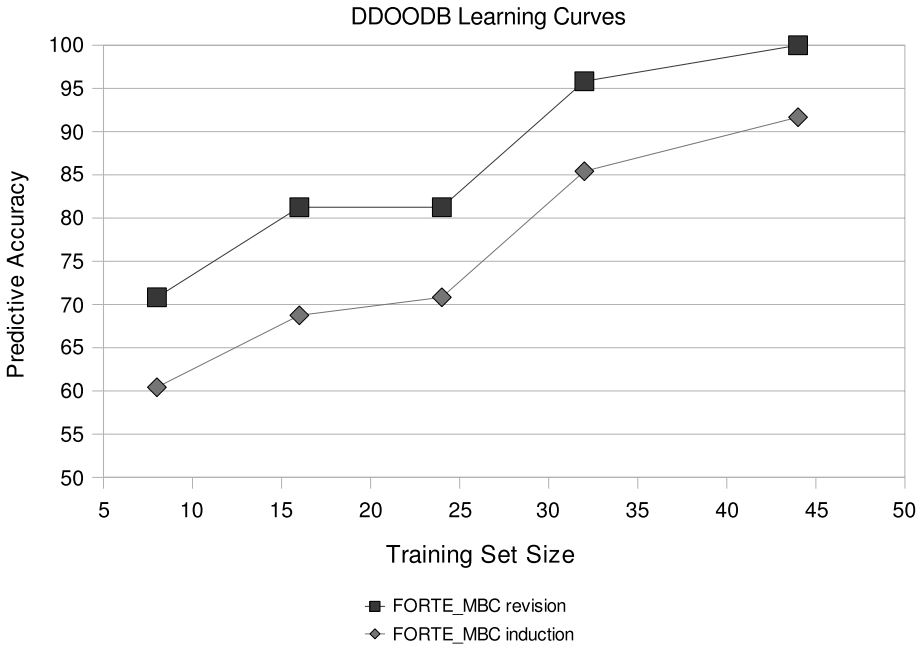

Fig. 9 Learning curves of DDOODB dataset

of the bottom clause can effectively be added to the clause being revised. This new revision system was named FORTE_MBC.

Experimental results show that a significant increase on efficiency was reached when comparing FORTE_MBC to the original FORTE: we obtained a speedup of 57 times on average with respect to the Hill Climbing algorithm (65 times in the best case) and an average speedup of 78 times with respect to the Relational Pathfinding algorithm (143 times in the best case). Moreover, the accuracies obtained by the modified revision process were at least maintained compared to the original system. Also, more comprehensible theories were generated when using the modified revision system.

Additionally, we compared the revision process followed by FORTE_MBC with induction algorithms using a covering and a non-covering approach. In both cases, the revision process is slower than the induction when the initial theories are not approximately correct, although the revision produces more accurate results. But, when the theories are approximately correct, revision always produces more accurate results than induction, which is a result compatible with previous literature of first-order theory revision.

Finally, induction by FORTE_MBC was compared to induction by the traditional ILP system Aleph (i.e., both systems learning from scratch). Induction by FORTE_MBC is faster than the induction of Aleph and also generates more compact theories in most of the cases, while obtaining comparable accuracies.

Future work Our work in progress tackles the problem of overly large bottom clauses, incorporating the ideas of the BETH system (Tang et al. 2003) in the revision process of FORTE, where the bottom clause becomes "virtual" since it is not constructed beforehand, as in done in Aleph, but it is discovered during the search for a good clause.

In Ong et al. (2005), the Relational Pathfinding algorithm was incorporated into Aleph. As future work, we intend to compare this approach to the Relational Pathfinding with bottom clause and modes used in FORTE_MBC.

In Paes et al. (2008), stochastic local search techniques were applied to the FORTE system and was obtained a significant improvement in running time as well as in accuracy. The goal of that work was to reduce the search space for misclassified examples, for revision points and for proposed revisions, since all of those are expensive tasks performed during the revision process. As in the original system, the antecedent addition operation used all the knowledge base. Therefore, to further improve the efficiency a future work will merge the idea of BETH and use the ("virtual") bottom clause as the search space when generating antecedents in the stochastic revision system. In this case, we intend to benefit from some recent results concerning stochastically searched spaces constrained by a bottom clause (Tamaddoni-Nezhad and Muggleton 2008).

Finally, we intend to formally characterize the refinement operators of FORTE_MBC (Badea 2001; Tamaddoni-Nezhad and Muggleton 2008), so that the benefits of searching whole theories using these operators are more theoretically founded.

Acknowledgements The first author is financially supported by Capes, the second author is financially supported by Capes during the Ph.D. and by CNPq during the doctorate sandwich and the third author by CNPq and FAPERJ. We would like to thank Bradley Richards and Raymond Mooney, Ashwin Srinivasan, Stephen Muggleton and Vítor Santos Costa for making FORTE, Aleph, Progol and YAP systems available, respectively. We would also like to thank Eric Silva, Rafael Pereira and Guilherme Niedu for helping on the implementation of modes, Fernanda Baião, Marta Mattoso and Jude Shavlik for the DDOODB dataset and Vítor Santos Costa for useful discussions. Finally, we would like to thank the anonymous reviewers for their remarks and very helpful suggestions.

Appendix A: FORTE_BC results

In Tables 15, 16, 17 and 18 below we compare the runtime, the predictive accuracies and the size of the theories in number of literals returned by FORTE_MBC and FORTE_BC (FORTE_MBC without validating literals returned by the bottom clause as to their compliance with mode declarations), considering revision in the first two tables and learning from scratch in the remaining tables. Each value in the tables is the average of 10-folds cross validation. Both Hill Climbing and Relational Pathfinding algorithms were considered, running independently. When revising, the theories provided to the revision process were obtained from Aleph with default parameters. The symbol ⋆ indicates that the difference between FORTE_MBC and FORTE_BC is significant.

Table 15 Runtime in seconds, predictive accuracy and size in number of literals for FORTE_MBC e FORTE_BC revising with internal CV, both using Hill Climbing algorithm for adding antecedents in a clause

Datasets	FORTE_MBC revising			FORTE_BC revising		
	Runtime	Accuracy	Size	Runtime	Accuracy	Size
Amine	25.53	68.96	34.50	25.83	68.96	34.50
Toxic	15.85	75.64	27.10	15.77	75.64	27.10
Choline	150.63	62.52	40.20	81.09	63.18	44.10
Scopo	72.11	65.25	45.80	71.06	65.25	45.80
DssTox	12.05⋆	74.52	13.60	40.11	78.00	15.00

Table 16 Runtime in seconds, predictive accuracy and size in number of literals for FORTE_MBC e FORTE_BC revising with internal CV, both using Relational Pathfinding algorithm for adding antecedents in a clause

Datasets	FORTE_MBC revising			FORTE_BC revising		
	Runtime	Accuracy	Size	Runtime	Accuracy	Size
Amine	72.34	72.46	51.10	76.76	74.46	55.30
Toxic	42.27	80.22	41.30	49.92	80.00	43.70
Scopo	272.49	65.39	81.60	127.94	67.12	80.70

Table 17 Runtime in seconds, predictive accuracy and size in number of literals for FORTE_MBC e FORTE_BC learning from scratch with internal CV, both using Hill Climbing algorithm for adding antecedents in a clause

Datasets	FORTE_MBC learning from scratch			FORTE_BC learning from scratch		
	Runtime	Accuracy	Size	Runtime	Accuracy	Size
Amine	2.97	67.97	7.20	2.96	67.82	5.20
Toxic	4.26	67.60	8.00	3.99	67.60	8.00
Choline	8.67	63.05	10.50	13.70	61.69	23.60
Scopo	3.42	61.66	10.20	3.15	61.66	10.20
DssTox	9.55⋆	78.00	7.80	25.51	78.00	8.40

Table 18 Runtime in seconds, predictive accuracy and size in number of literals for FORTE_MBC e FORTE_BC learning from scratch with internal CV, both using Relational Pathfinding algorithm for adding antecedents in a clause

Datasets	FORTE_MBC revising			FORTE_BC revising		
	Runtime	Accuracy	Size	Runtime	Accuracy	Size
Amine	35.81	73.66	41.30	38.93	75.67	52.10
Toxic	47.28	77.29	27.30	47.07	75.05	36.80
Scopo	44.03	64.64	32.80	52.80	63.57	49.10

From the tables we can see that there is no significant difference between FORTE_MBC and FORTE_BC, although in some cases FORTE_BC returns larger theories, which leads to the conclusion that in FORTE_MBC the theories are smaller because of the use of mode declarations.

We performed experiments with FORTE using only modes definitions without the bottom clause. However, in most of the cases, the results were not significantly different from the original FORTE. On the other hand, the results obtained from FORTE_MBC were always significantly better than FORTE using only mode declarations. Thus, it is the bottom clause, rather than the mode declarations, that accounts for the improvements of the revision process.

Appendix B: Modes definitions

B.1 Alzheimer domain

For the relations on the background knowledge the modes definitions are the same and come from the literature. They are defined as follows.

$modeb(*, x\_subst(+a, -n, -b))$
$modeb(*, alk\_groups(+a, -n))$
$modeb(*, r\_subst\_1(+a, -l))$
$modeb(*, r\_subst\_2(+a, -m))$
$modeb(*, r\_subst\_3(+a, -n))$
$modeb(*, ring\_substitutions(+a, -n))$
$modeb(*, ring\_subst\_1(+a, -b))$
$modeb(*, ring\_subst\_2(+a, -b))$
$modeb(*, ring\_subst\_3(+a, -b))$
$modeb(*, ring\_subst\_4(+a, -b))$
$modeb(*, ring\_subst\_5(+a, -b))$
$modeb(*, ring\_subst\_6(+a, -b))$
$modeb(*, polar(+b, -c))$
$modeb(*, size(+b, -d))$
$modeb(*, flex(+b, -e))$
$modeb(*, h\_doner(+b, -f))$
$modeb(*, h\_acceptor(+b, -g))$
$modeb(*, pi\_doner(+b, -h))$
$modeb(*, pi\_acceptor(+b, -i))$
$modeb(*, polarisable(+b, -j))$
$modeb(*, sigma(+b, -k))$
$modeb(*, n\_val(+a, -n))$
$modeb(*, gt(+n, -n))$
$modeb(*, great\_polar(+c, -c))$
$modeb(*, great\_size(+d, -d))$
$modeb(*, great\_flex(+e, -e))$
$modeb(*, great\_h\_don(+f, -f))$
$modeb(*, great\_h\_acc(+g, -g))$
$modeb(*, great\_pi\_don(+h, -h))$
$modeb(*, great\_pi\_acc(+i, -i))$
$modeb(*, great\_polari(+j, -j))$
$modeb(*, great\_sigma(+k, -k))$

For the target predicates, the modeh definition is different in each dataset as follows.

1. Amine Domain: $modeh(((1, great\_ne(+a, +a))$.
2. Toxic Domain: $modeh((1, less\_toxic(+a, +a))$.
3. Scopolamine Domain: $modeh((1, great\_rsd(+a, +a))$.
4. Choline Domain: $modeh((1, great(+a, +a))$.

B.2 DssTox domain

$modeh((1, active(+molecule))$
$modeb(*, atom(+molecule, -atomid, \#element))$

modeb(∗, *sbond*(+*molecule*, +*atomid*, −*atomid*, #*bondtype*))
modeb(∗, *linked*(+*molecule*, +*atomid*, −*atomid*, #*element*, #*bondtype*))

B.3 Family domain

modeh((1, *wife*(+*person*, +*person*))
modeh((1, *husband*(+*person*, +*person*))
modeh((1, *mother*(+*person*, +*person*))
modeh((1, *father*(+*person*, +*person*))
modeh((1, *daughter*(+*person*, +*person*))
modeh((1, *son*(+*person*, +*person*))
modeh((1, *sister*(+*person*, +*person*))
modeh((1, *brother*(+*person*, +*person*))
modeh((1, *aunt*(+*person*, +*person*))
modeh((1, *uncle*(+*person*, +*person*))
modeh((1, *niece*(+*person*, +*person*))
modeh((1, *nephew*(+*person*, +*person*))
modeb(∗, *sibling*(+*person*, +*person*))
modeb(∗, *au*(+*person*, +*person*))
modeb(∗, *parent*(+*person*, −*person*))
modeb(∗, *parent*(−*person*, +*person*))
modeb(∗, *married*(+*person*, −*person*))
modeb(∗, *married*(−*person*, +*person*))
modeb(∗, *gender*(+*person*, #*gender*))

B.4 DDOODB domain

modeh((1, *chooseVerticalFragmentationMethod*(+*operation*, +*class*))
modeh((1, *choosePrimaryHorizontalFragmentationMethod*(+*operation*, +*class*))
modeh((1, *chooseDerivedHorizontalFragmentationMethod*(+*operation*, +*class*,
+ *class*))
modeb(∗, *relationshipAccess*(−*relationship*, +*class*, +*class*))
modeb(∗, *query*(+*id*, −*freq*, −*list*(*operation*)))
modeb(∗, *operationAccess*(+*operation*, −*list*(*class*)))
modeb(∗, *navigates*(+*operation*, +*class*, +*class*))
modeb(∗, *isDerivedFragmented*(+*class*))
modeb(∗, *isNotDerivedFragmented*(+*class*))
modeb(∗, *isVerticallyFragmented*(+*class*))
modeb(∗, *isNotVerticallyFragmented*(+*class*))
modeb(∗, *cardinality*(+*class*, #*card*))
modeb(∗, *fragmentation*(+*class*, #*frag*))
modeb(∗, *relationshipType*(+*relationship*, #*rel*))
modeb(∗, *classification*(+*operation*, #*classif*))

References

Adé, H., Malfait, B., & Raedt, L. D. (1994). RUTH: an ILP theory revision system. In *LNCS. Proceedings of 8th international symposium of methodologies for intelligent systems (ISMIS-94)* (pp. 336–345). Berlin: Springer.

Badea, L. (2001). A refinement operator for theories. In *LNAI: Vol. 2157. Proceedings of the 11th international conference on ILP* (pp. 1–14). Berlin: Springer.

Baião, F., Mattoso, M., Shavlik, J., & Zaverucha, G. (2003). Applying theory revision to the design of distributed databases. In *LNAI: Vol. 2835. Proceedings of the 13th int. conference on inductive logic programming* (pp. 57–74). Berlin: Springer.

Baião, F. A., Mattoso, M., & Zaverucha, G. (2004). A distribution design methodology for object DBMS. *Distributed and Parallel Databases*, *16*(1), 45–90.

Bratko, I. (1999). Refining complete hypotheses in ILP. In *LNAI: Vol. 1634. Proceedings of the 9th international conference on inductive logic programming* (pp. 44–55). Berlin: Springer.

De Raedt, L. (2008). *Logical and relational learning*. Berlin: Springer.

Dietterich, T., Domingos, P., Getoor, L., Muggleton, S., & Tadepalli, P. (2008). Structured machine learning: the next ten years. *Machine Learning*, *73*, 3–23.

Duboc, A. L., Paes, A., & Zaverucha, G. (2008). Using the bottom clause and modes declarations on FOL theory revision from examples. In *LNAI: Vol. 5194. Proceedings of the 18th international conference on inductive logic programming* (pp. 91–106). Berlin: Springer.

Dzeroski, S., & Lavrac, N. (Eds.). (2001). *Relational data mining*. Berlin: Springer.

Fang, H., Tong, W., Shi, L. M., Blair, R., Perkins, R., Branham, W., Hass, B. S., Xie, Q., Dial, S. L., Moland, C. L., & Sheehan, D. M. (2001). Structure-activity relationships for a large diverse set of natural, synthetic, and environmental estrogens. *Chemical Research in Toxicology*, *3*(14), 280–294.

King, R. D., Sternberg, M. J. E., & Srinivasan, A. (1995). Relating chemical activity to structure: an examination of ilp successes. *New Generation Computing*, *13*(3–4), 411–433.

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the international joint conference on artificial intelligence (IJCAI)* (pp. 1137–1145).

Landwehr, N., Kersting, K., & De Raedt, L. (2007). Integrating naive bayes and foil. *Journal of Machine Learning Research*, *8*, 481–507.

Morik, K., Wrobel, S., Kietz, J.-U., & Emde, W. (1993). *Knowledge acquisition and machine learning: theory methods and applications*. San Diego: Academic Press.

Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing*, *13*, 245–286.

Muggleton, S. (2005). Machine learning for systems biology. In *LNCS: Vol. 3625. Proceedings of the 15th international conference on inductive logic programming* (pp. 416–423). Berlin: Springer.

Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: theory and methods. *Journal of Logic Programming*, *19*(20), 629–679.

Nadeau, C., & Bengio, Y. (2003). Inference for the generalization error. *Machine Learning*, *52*(3), 239–281.

Nienhuys-Cheng, Shan-Hwei, & de Wolf, R. (1997). *Foundations of inductive logic programming*. Berlin: Springer.

Ong, I. M., Dutra, I. C., Page, D., & Costa, V. C. (2005). Mode directed path finding. In *Proceedings of the 16th ECML* (vol. 3720, pp. 673–681).

Paes, A., Zaverucha, G., & Costa, V. S. (2008). Revising first-order logic theories from examples through stochastic local search. In *LNAI: Vol. 4894. Proceedings of the revised selected papers of the 17th international conference on inductive logic programming* (pp. 200–210). Berlin: Springer.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Richards, B. L., & Mooney, R. J. (1995). Automated refinement of first-order horn-clause domain theories. *Machine Learning*, *19*(2), 95–131.

Santos Costa, V. (2008). The life of a logic programming system. In *LNCS: Vol. 5366. Proceedings of the 24th international conference on logic programming (ICLP 2008)* (pp. 1–6). Berlin: Springer.

Shapiro, E. Y. (1983). *Algorithmic program debugging*. ACM Distinguished Doctoral Dissertations. New York: MIT Press.

Srinivasan, A. (2001). The Aleph manual.

Tamaddoni-Nezhad, A., & Muggleton, S. (2008). A note on refinement operators for IE-based ILP systems. In *LNAI: Vol. 5194. Proceedings of the 18th international conference on ILP* (pp. 297–314). Berlin: Springer.

Tang, L. R., Mooney, R. L., & Melville, P. (2003). Scaling up ILP to large examples: results on link discovery for counter-terrorism. In *Proceedings of the KDD-2003 workshop om multi-relational data mining*, Washington, DC (pp. 107–121).

Wrobel, S. (1996). First-order theory refinement. In L. D. Raedt (Ed.) *Advances in inductive logic programming* (pp. 14–33). Amsterdam: IOS Press.