Improving efficiency of recursive theory learning

Antonio Varlaro, Margherita Berardi, Donato Malerba

Dipartimento di Informatica – Università degli Studi di Bari
via Orabona 4 - 70126 Bari
{varlaro, berardi, malerba}@di.uniba.it

Abstract. Inductive learning of recursive logical theories from a set of examples is a complex task characterized by three important issues, namely the adoption of a generality order stronger than θ-subsumption, the non-monotonicity of the consistency property, and the automated discovery of dependencies between target predicates. Solutions implemented in the learning system ATRE are briefly reported in the paper. Moreover, efficiency problems of the learning strategy are illustrated and two caching strategies, one for the clause generation phase and one for the clause evaluation phase, are described. The effectiveness of the proposed caching strategies has been tested on the document processing domain. Experimental results are discussed and conclusions are drawn.

1. Introduction

Inductive Logic Programming (ILP) has evolved from previous research in Machine Learning, Logic Programming, and Inductive Program Synthesis. Like Machine Learning, it deals with the induction of concepts from observations (examples) and the synthesis of new knowledge from experience. Its peculiarity is the use of computational logic as the representation mechanism for concept definitions and observations. Typically, the output of an ILP system is a logical theory expressed as a set of definite clauses, which logically entail all positive examples and no negative example. Therefore, each concept definition corresponds to a predicate definition and a concept learning problem is reformulated as a predicate learning problem.

Learning a single predicate definition from a set of positive and negative examples is a classical problem in ILP. In this paper we are interested in the more complex case of learning multiple predicate definitions, provided that both positive and negative examples of each concept/predicate to be learned are available. Complexity stems from the fact that the learned predicates may also occur in the antecedents of the learned clauses, that is, the learned predicate definitions may be interrelated and depend on one another, either hierarchically or involving some kind of mutual recursion. For instance, to learn the definitions of odd and even numbers, a multiple predicate learning system will be provided with positive and negative examples of both odd and even numbers, and may generate the following recursive logical theory:

odd(X) ← succ(Y,X), even(Y)
even(X) ← succ(Y,X), odd(Y)

$$\text{even}(X) \leftarrow \text{zero}(X)$$

where the definitions of *odd* and *even* are interdependent. This example shows that the problem of learning multiple predicate definitions is equivalent, in its most general formulation, to the problem of learning recursive logical theories.

There has been considerable debate on the actual usefulness of learning recursive logical theories in knowledge acquisition and discovery applications. It is a common opinion that very few real-life concepts seem to have recursive definitions, rare examples being "ancestor" and natural language [2, 10]. Despite this scepticism, in the literature it is possible to find several ILP applications in which recursion has proved helpful [7]. Moreover, many ILP researchers have shown some interest in multiple predicate learning [6], which presents the same difficulty of recursive theory learning in its most general formulation.

To formulate the recursive theory learning problem and then to explain its main theoretical issues, some basic definitions are given below.

Generally, every logical theory T can be associated with a directed graph $\gamma(T)=<N,E>$, called the *dependency graph* of T, in which (i) each predicate of T is a node in N and (ii) there is an arc in E directed from a node a to a node b, iff there exists a clause C in T, such that a and b are the predicates of a literal occurring in the head and in the body of C, respectively.

A dependency graph allows representing the predicate dependencies of T, where a *predicate dependency* is defined as follows:

Definition 1 (predicate dependency). A predicate p depends on a predicate q in a theory T iff (i) there exists a clause C for p in T such that q occurs in the body of C; or (ii) there exists a clause C for p in T with some predicate r in the body of C that depends on q.

Definition 2 (recursive theory). A logical theory T is *recursive* if the dependency graph $\gamma(T)$ contains at least one cycle.

In *simple* recursive theories all cycles in the dependency graph go from a predicate p into p itself, that is, simple recursive theories may contain recursive clauses, but cannot express mutual recursion.

Definition 3 (predicate definition). Let T be a logical theory and p a predicate symbol. Then the *definition* of p in T is the set of clauses in T that have p in their head. Henceforth, $\delta(T)$ will denote the set of predicates defined in T and $\pi(T)$ will denote the set of predicates occurring in T, then $\delta(T) \subseteq \pi(T)$.

In a quite general formulation, the recursive theory learning task can be defined as follows:

Given

- A set of *target* predicates $p_1, p_2, \ldots, p_r$ to be learned
- A set of positive (negative) examples E_i^+ (E_i^-) for each predicate p_i, $1 \le i \le r$
- A background theory BK
- A language of hypotheses $\mathcal{L}_H$ that defines the space of hypotheses S_H

Find

a (possibly recursive) logical theory $T \in S_H$ defining the predicates $p_1, p_2, \ldots, p_r$ (that is, $\delta(T)=\{p_1, p_2, \ldots, p_r\}$) such that for each i, $1 \le i \le r$, $BK \cup T \models E_i^+$ (*completeness* property) and $BK \cup T \not\models E_i^-$ (*consistency* property).

Three important issues characterize recursive theory learning. First, the generality order typically used in ILP, namely θ-subsumption [13], is not sufficient to guarantee the completeness and consistency of learned definitions, with respect to logical entailment [12]. Therefore, it is necessary to consider a stronger generality order, which is consistent with the logical entailment for the class of recursive logical theories we take into account.

Second, whenever two individual clauses are consistent in the data, their conjunction need not be consistent in the same data [5]. This is called the non-monotonicity property of the normal ILP setting, since it states that adding new clauses to a theory T does not preserve consistency. Indeed, adding definite clauses to a definite program enlarges its least Herbrand model (LHM), which may then cover negative examples as well. Because of this non-monotonicity property, learning a recursive theory one clause at a time is not straightforward.

Third, when multiple predicate definitions have to be learned, it is crucial to discover dependencies between predicates. Therefore, the classical learning strategy that focuses on a predicate definition at a time is not appropriate.

To overcome these problems a new approach to the learning of multiple dependent concepts has been proposed in [8] and implemented in the learning system ATRE (www.di.uniba.it/~malerba/software/atre). This approach differs from related works for at least one of the following three aspects: the learning strategy, the generalization model, and the strategy to recover the consistency property of the learned theory when a new clause is added.

The paper synthesizes and extends the work presented in [8]. In particular, it presents a brief overview of solutions proposed and implemented in ATRE to the three main issues above. Evolutions of the search strategy are also reported. More precisely, two new issues regarding the search space exploration are faced, one concerning search bias definition in order to allow the user to guide the search space exploration according to his/her preference, and the other one concerning efficiency problems due to the computational complexity of the search space. Some solutions have been proposed and implemented in a new version of the system ATRE.

The paper is organized as follows. Section 2 illustrates issues and solutions related to the recursive theory learning. Section 3 introduces efficiency problems and presents optimization approaches adopted in ATRE. Section 4 illustrates the application of ATRE on real-world documents and presents results on efficiency gain. Finally, in Section 5 some conclusions are drawn.

2. Issues and solutions

2.1 The generality order

As explained above, in recursive theory learning it is necessary to consider a generality order that is consistent with the logical entailment for the class of recursive logical theories. A generality order (or *generalization model*) provides a basis for organizing the search space and is essential to understand how the search strategy proceeds. The main problem with the well-known θ-subsumption is that the objects

of comparison are two clauses, say C and D, and no additional source of knowledge (e.g., a theory T) is considered. For instance, with reference to the previous example on *odd* and *even* predicates, the clause:

C: *odd(X)* ← *succ(Y,X)*, *even(Y)*

logically entails, and hence can be correctly considered more general than

D: *odd(3)* ← *succ(0,1)*, *succ(1,2)*, *succ(2,3)*, *even(0)*

only if we take into account the theory

T: *even(A)* ← *succ(B,A)*, *odd(B)*

even(C) ← *zero(C)*

Therefore, we are only interested in those generality orders that compare two clauses relatively to a given theory T, such as Buntine's *generalized subsumption* [3] and Plotkin's notion of *relative generalization* [13, 14].

Informally, generalized subsumption ($\leq_T$) requires that the heads of C and D refer to the same predicate, and that the body of D can be used, together with the background theory T, to entail the body of C. Unfortunately, generalized subsumption is too weak for recursive theories, because in some cases, given two clauses C and D, it may happen that $T \cup \{C\} \models D$ holds but it can not be concluded that $C \leq_T D$.

Plotkin's notion of *relative generalization* [13, 14] was originally proposed for a theory T of unit clauses. Buntine [3] reports an extension of relative generalization to the case of a theory T composed of definite clauses (not necessarily of unit clauses)

Definition 4 (relative generalization). Let C and D be two definite clauses. C is *more general than D* under relative generalization, with respect to a theory T, if a substitution θ exists such that $T \models \forall(C\theta \Rightarrow D)$.

The following theorem holds for this extended notion of relative generalization:

Theorem 1. Let C and D be two definite clauses and T a logical theory. C is *more general than D* under relative generalization, with respect to a theory T, if and only if C occurs at most once in some refutation demonstrating $T \models \forall(C \Rightarrow D)$.

However, this extended notion of relative generalization is still inadequate. From one side, it is still weak. Indeed, if we consider the clauses and the theory reported in the example above, it is clear that a refutation demonstrating $T \models \forall(C \Rightarrow D)$ involves twice the clause C to prove both *odd(1)* and *odd(3)*.

Malerba [8] has defined the following generalization order, which proved suitable for recursive theories.

Definition 5 (generalized implication). Let C and D be two definite clauses. C is *more general than D* under generalized implication, with respect to a theory T, denoted as $C \leq_{T\Rightarrow} D$, if a substitution θ exists such that $head(C)\theta = head(D)$ and $T \models \forall(C \Rightarrow D)$.

Decidability of the generalized implication test is guaranteed in the case of Datalog clauses [4]. In fact, the restriction to function-free clauses is common in ILP systems, such as ATRE, which remove function symbols from clauses and put them in the background knowledge by techniques such as flattening [15].

2.2 The non-monotonicity property

It is noteworthy that generalized implication compares two definite clauses for generalization. This means that the search space structured by this generality order is the space of definite clauses. A recursive logical theory is generally composed of several clauses, therefore the learning strategy must search for one clause at a time. More precisely, a recursive theory T is built step by step, starting from an empty theory T_0, and adding a new clause at each step. In this way we get a sequence of theories

$$T_0 = \varnothing, T_1, \ldots, T_i, T_{i+1}, \ldots, T_n = T,$$

such that $T_{i+1} = T_i \cup \{C\}$ for some clause C. If we denote by LHM(T_i) the least Herbrand model of a theory T_i, the stepwise construction of theories entails that LHM(T_i) $\subseteq$ LHM(T_{i+1}), for each i$\in$ {0, 1, …, n-1}, since the addition of a clause to a theory can only augment the LHM. Henceforth, we will assume that both positive and negative examples of predicates to be learned are represented as *ground atoms* with a + or - label. Therefore, examples may or may not be elements of the models LHM(T_i). Let *pos(LHM(T_i))* and *neg(LHM(T_i))* be the number of positive and negative examples in LHM(T_i), respectively. If we guarantee the following two conditions:

1. *pos(LHM(T_i))* < *pos(LHM(T_{i+1}))* for each $i \in \{0, 1, \ldots, n\text{-}1\}$, and
2. *neg(LHM(T_i))* = 0 for each $i \in \{0, 1, \ldots, n\}$,

then after a finite number of steps a theory T, which is complete and consistent, is built. This learning strategy is known as *sequential covering* (or *separate-and-conquer*) [9].

In order to guarantee the first of the two conditions it is possible to proceed as follows. First, a positive example e^+ of a predicate p to be learned is selected, such that e^+ is not in LHM(T_i). The example e^+ is called *seed*. Then the space of definite clauses more general than e^+ is explored, looking for a clause C, if any, such that neg(LHM($T_i \cup \{C\}$)) = $\varnothing$. In this way we guarantee that the second condition above holds as well. When found, C is added to T_i giving T_{i+1}. If some positive examples are not included in LHM(T_{i+1}) then a new seed is selected and the process is repeated.

The second condition is more difficult to guarantee because of the second issue presented in the introduction, namely, the non-monotonicity property. Algorithmic implications of this property may be effectively illustrated by means of an example. Consider the problem of learning the definitions of ancestor and father from a complete set of positive and negative examples. Suppose that the following recursive theory T_2 has been learned at the second step:

C_1: *ancestor*(*X,Y*) ← *parent(X,Y)*

C_2: *father*(*Z,W*) ← *ancestor(Z,W), male(Z)*

Note that T_2 is consistent but still incomplete. Thus a new clause will be generated at the third step of the sequential-covering strategy. It may happen that the generated clause is the following:

C: *ancestor*(*A,B*) ← *parent(A,D),ancestor(D,B)*

which is consistent given T_2, but when added to the recursive theory, it makes clause C_2 inconsistent.

There are several ways to remove such inconsistency by revising the learned theory. Nienhuys-Cheng and de Wolf [11] describe a complete method of

specializing a logic theory with respect to sets of positive and negative examples. The method is based upon unfolding, clause deletion and subsumption. These operations are not applied to the last clause added to the theory, but may involve any clause of the inconsistent theory. As a result, clauses learned in the first inductive steps could be totally changed or even removed. This theory revision approach, however, is not coherent with the stepwise construction of the theory T presented above, since it re-opens the whole question of the validity of clauses added in the previous steps. An alternative approach consists of simple syntactic changes in the theory, which eventually creates new *layers* in a logical theory, just as the stratification of a normal program creates new strata [1].

More precisely, a layering of a theory T is a partition of the clauses in T into n disjoint sets of clauses or *layers* T^i such that $LHM(T)= LHM(LHM(\cup_{j=0,\ldots,n-2} T^j)\cup T^{n-1})$, that is, $LHM(T)$ can be computed by iteratively applying the immediate consequence operator to T^i, starting from the interpretation $LHM(\cup_{j=0,\ldots,i-1} T^j)$, for each $i\in\{1, \ldots,n\}$. In [8] an efficient method for the computation of a layering is reported. It is based on the concept of collapsed dependency graph and returns a unique layering for a given logical theory T. The layering of a theory provides a semi-naive way of computing the generalized implication test presented above and provides a solution to the problem of consistency recovering when the addition of a clause makes the theory inconsistent.

Theorem. Let $T=T^0 \cup \ldots \cup T^i \cup \ldots \cup T^{n-1}$ be a consistent theory partitioned into n layers, and C be a definite clause whose addition to the theory T makes a clause in layer T^i inconsistent. Let $p\in\{p_1, p_2, \ldots, p_r\}$ be the predicate in the head of C. Let T'' be a theory obtained from T by substituting all occurrences of p in T with a new predicate symbol, p', and $T'=T''\cup\{p(t_1, \ldots, t_n) \leftarrow p'(t_1, \ldots, t_n)\}\cup\{C\}$. Then T' is consistent and $LHM(T) \subseteq LHM(T') \setminus \{p(t_1, \ldots, t_n) \leftarrow p'(t_1, \ldots, t_n)\}$.

In short, the new theory T' obtained by renaming the predicate p with a new predicate name p' before adding C is consistent and keeps the original coverage of T. This introduces a first variation of the classical separate-and-conquer strategy sketched above, since the addition of a locally consistent clause C generated in the conquer stage is preceded by a global consistency check. If the result is negative, the partially learned theory is first restructured, and then two clauses, $p(t_1, \ldots, t_n) \leftarrow p'(t_1, \ldots, t_n)$ and C, are added. For instance, in the example above the result will be:

C_1': *ancestor'*$(X,Y) \leftarrow$ *parent*(X,Y)
C_2': *father*$(Z,W) \leftarrow$ *ancestor'*(Z,W), *male*(Z)
ancestor$(U,V) \leftarrow$ *ancestor'*(U,V)
C: *ancestor*$(A,B) \leftarrow$ *parent*(A,D),*ancestor*(D,B)

It is noteworthy that, in the proposed approach to consistency recovery, new predicates are invented, which aim to accommodate previously acquired knowledge (theory) with the currently generated hypothesis (clause).

2.3 Discovering dependencies between predicates

The third and last issue to deal with is the automated discovery of dependencies between target predicates $p_1, p_2, \ldots, p_r$. A solution to this problem is based on another variant of the separate-and-conquer learning strategy. Traditionally, this strategy is adopted by single predicate learning systems that generate clauses with the same predicate in the head at each step. In multiple predicate learning (or recursive theory learning) clauses generated at each step may have different predicates in their heads. In addition, the body of the clause generated at the *i*-th step may include all target predicates $p_1, p_2, \ldots, p_r$ for which at least a clause has been added to the partially learned theory in previous steps. In this way, dependencies between target predicates can be generated.

Obviously, the order in which clauses of distinct predicate definitions have to be generated is not known in advance. This means that it is necessary to generate clauses with different predicates in the head and then to pick one of them at the end of each step of the separate-and-conquer strategy. Since the generation of a clause depends on the chosen seed, several seeds have to be chosen such that at least one seed per incomplete predicate definition is kept. Therefore, the search space is actually a forest of as many search-trees (called *specialization hierarchies*) as the number of chosen seeds. A directed arc from a node C to a node C' exists if C' is obtained from C by a single refinement step. Operatively, the (downward) refinement operator considered in this work adds a new literal to a clause.

The forest can be processed in parallel by as many concurrent tasks as the number of search-trees. Each task traverses the specialization hierarchy top-down (or general-to-specific), but synchronizes traversal with the other tasks at each level. Initially, some clauses at depth one in the forest are examined concurrently. Each task is actually free to adopt its own search strategy, and to decide which clauses are worth to be tested. If none of the tested clauses is consistent, clauses at depth two are considered. Search proceeds towards deeper and deeper levels of the specialization hierarchies until at least a user-defined number of consistent clauses is found. Task synchronization is performed after that all "relevant" clauses at the same depth have been examined. A supervisor task decides whether the search should carry on or not on the basis of the results returned by the concurrent tasks. When the search is stopped, the supervisor selects the "best" consistent clause according to the user's preference criterion. This strategy has the advantage that simpler consistent clauses are found first, independently of the predicates to be learned.[1] Moreover, the synchronization allows tasks to save much computational effort when the distribution of consistent clauses in the levels of the different search-trees is uneven. The parallel exploration of the specialization hierarchies for *odd* and *even* is shown in Fig. 1.

[1] Apparently, some problems might occur for those recursive definitions where the recursive clause is syntactically simpler than the base clause. However, the proposed strategy does not allow the discovery of the recursive clause until the base clause has been found, whatever its complexity is.

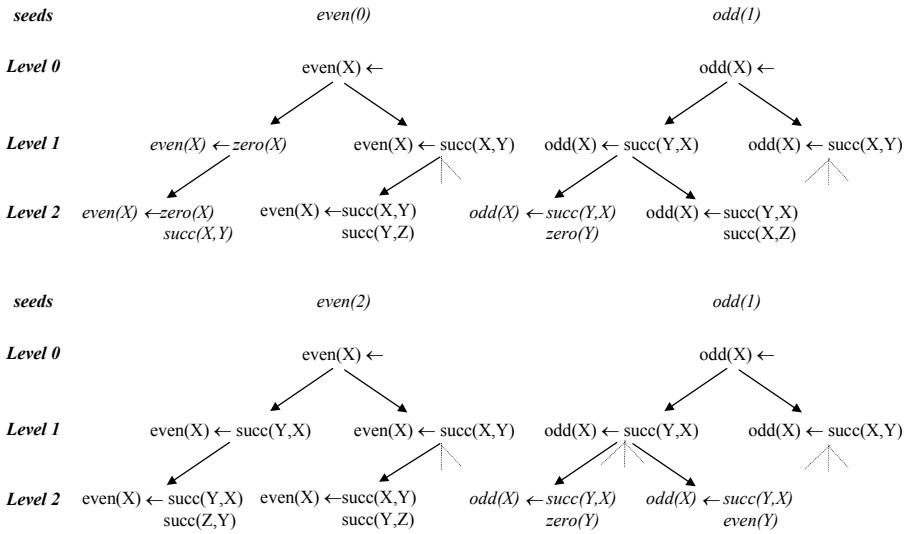

Fig. 1. Two steps (up and down) of the parallel search for the predicates *odd* and *even*. Consistent clauses are reported in italics.

2.4 Some refinements on the learning strategy

The learning strategy reported in previous section is quite general and there is room for several distinct implementations. In particular, the following three points have been left unspecified: 1) how seeds are selected; 2) what are the roots of specialization hierarchies; 3) what is the search strategy adopted by each task. In this section, solutions adopted in the last release of the learning system ATRE are illustrated.

Seed selection is a critical point. In the example of Fig. 1, if the search had started from *even(2)* and *odd(1)*, the first clause added to the theory would have been *odd(X)* ← *succ(Y,X), zero(Y)*, thus resulting in a less compact, though still correct, theory for odd and even numbers. Therefore, it is important to explore the specialization hierarchies of several seeds for each predicate. When training examples and background knowledge are represented either as sets of ground atoms (flattened representation) or as ground clauses, the number of candidate seeds can be very high, so the choice should be stochastic. The object-centered representation adopted by ATRE has the advantage of reducing the number of candidate seeds by partitioning the whole set of training examples *E* into *training objects*. The main assumption made in ATRE is that *each object contains examples explained by some base clauses of the underlying recursive theory*.[2] Therefore, by choosing as seeds *all* examples of different concepts represented in one training object, it is possible to induce some of the correct base clauses. Since in many learning problems the number of positive

[2] Problems caused by incomplete object descriptions violating the above assumption are not investigated in this work, since they require the application of *abductive* operators, which are not available in the current version of the system.

examples in an object is not very high, a parallel exploration of all candidate seeds is feasible. Mutually recursive concept definitions will be generated only after some base clauses have been added to the theory.

Seeds are chosen according to the textual order in which objects are input to ATRE. If a complete definition of the predicate p_j is not available yet at the i-th step of the separate-and-conquer search strategy, then there are still some uncovered positive examples of p_j. The first (seed) object O_k in the object list that contains uncovered examples of p_j is selected to generate seeds for p_j.

Generally, each specialization hierarchy is rooted in a unit clause, that is, a clause with an empty body. However, in some cases, the user has a clear idea of relevant properties that should appear in the body of the clauses and is even able to define the root of the specialization hierarchies. A *language bias* has been defined in ATRE to allow users to express constraints that should be satisfied by root clauses or by interesting clauses in the specialization hierarchy. In its current version, the language bias includes the following declarations:

starting_number_of_literals(p_i,N)

starting_clause(p_i,[L_1,L_2,...,L_N])

where p_i is a target predicate, N is a cardinal number, and [L_1,L_2,...,L_N] represents a list of literals. In particular, the *starting_number_of_literals* declaration specifies the initial length of the root clause (at least N literals in the body), while the *starting_clause* declaration specifies a conjunctive constraint on the body of a root clause: all literals in the list [L_1,L_2,...,L_N] must occur in the clause. Multiple *starting_clause* declarations for the same target predicate p_i specify alternative conjunctive constraints for the root clauses of specialization hierarchies associated to p_i. In addition, the following declaration:

starting_literal(p_i,[L_1,L_2,...,L_N])

specifies a disjunctive constraint at literal level for the body of root clauses. Literals are expressed as follows:

f(decl-arg$_1$, …, decl-arg$_n$) = Value

g(decl-arg$_1$, …, decl-arg$_n$) ∈ Range

where *decl-arg*'s are mode declarations for predicate arguments. Declarations are applicable only to variables and influence the way of generating variables. Two modes are available: *old* and *new*. The first mode means that the variable is an input variable, that is, it corresponds to a variable already occurring in the clause. The second mode means that the variable is a new one. Furthermore, values and ranges of predicates can be ground or not.

The third undefined point of the search strategy concerns the search strategy adopted by each task. ATRE applies a variant of the beam-search strategy. The system generates all candidate clauses at level $l+1$ starting from those filtered at level l in the specialization hierarchy. During task synchronization, which occurs level-by-level, the best m clauses are selected from those generated by all tasks. The user specifies the beam of the search, that is m, and a set of preference criteria for the selection of the best m clauses.

3. Improving efficiency in ATRE

In this section we present a novel caching strategy implemented in ATRE to overcome efficiency problems. Generally speaking, caching aims to save useful information that would be repeatedly recomputed otherwise, with a clear waste of time. In ATRE caching affects the two most computationally expensive phases of the learning process, namely the clause generation step and the clause evaluation step.

3.1 Caching for clause generation

The learning strategy sketched in Section 2.3 presents a large margin for optimization. One of the reasons is that every time a clause is added to the partially learned theory, the specialization hierarchies are reconstructed for a new set of seeds, which may intersect the set of seeds explored in the previous step. Therefore, it is possible that the system explores the same specialization hierarchies several times, since it has no memory of the work done in previous steps. This is particularly evident when concepts to learn are neither recursively definable nor mutually dependent. Caching the specialization hierarchies explored at the *i*-th step of the separate-and-conquer strategy and reusing part of them at the *(i+1)*-th step, seems to be a good strategy to decrease the learning time while keeping memory usage under acceptable limits.

First of all, we observe that a necessary condition for reusing a specialization hierarchy between two subsequent learning steps is that the associated seed remains the same. This means that if the seed of a specialization hierarchy is no longer considered at the *(i+1)*-th step, then the corresponding clauses cached at the *i*-th step can be discarded.

However, even in the case of same seed, not all the clauses of the specialization hierarchy will be actually useful. For instance, the cached copies of a clause *C* added to T_i can be removed from all specialization hierarchies including it. Moreover, all clauses that cover only positive examples already covered by *C* can be dropped, according to the separate-and-conquer learning strategy. These examples explain why a cached specialization hierarchy has to be pruned before considering it at the *(i+1)*-th step of the learning strategy.

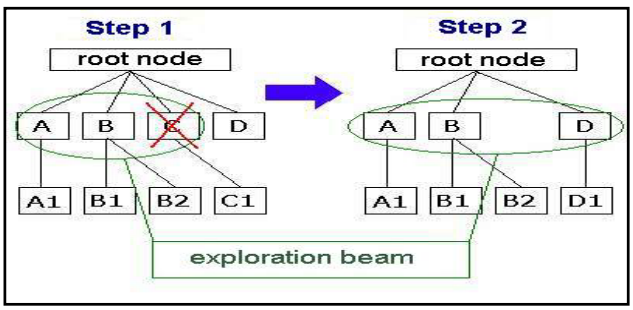

Fig.2. An example of search-tree pruning effect on beam-search width.

In order to maintain unchanged the width of the search beam, some grafting operations are necessary after pruning. Indeed, by removing the clauses that will be no more examined, the exploration beam decreases. Grafting operations aim to consider previous unspecialized clauses in order to restore the beam width, as shown in Fig.2.

Grafting operations are also necessary to preserve the generation of recursive clauses. For instance, by looking at the two specialization hierarchies of the predicate *odd* in Fig. 1, it is clear that once the clause *even(X)* ← *zero(X)* has been added to the empty theory (step 1), the consistent clause *odd(X)* ← *succ(Y,X), even(Y)* can be a proper node of the specialization hierarchy, since a base clause for the recursive definition of the predicate *even* is already available. Therefore, the grafting operations also aim to complete the pruned specialization hierarchy with new clauses that take predicate dependencies into account.

3.2 Caching for clause evaluation

Evaluating a clause corresponds to determining the lists of positive and negative examples covered by the clause itself. This requires a number of generalized implication tests, one for each positive or negative example. In ATRE the generalized implication test is optimized, however, if the number of tests to perform is high, the clause evaluation leads to efficiency problems anyway. To reduce the number of tests, we propose to cache the list of positive and negative examples of each clause, as well.

To clarify this caching technique, we distinguish between *dependent* clauses, that is, clauses with at least one literal in the body whose predicate symbol is a target predicate p_i, and independent clauses (all the others).

In independent clauses, the lists of negative examples remain unchanged between two subsequent learning steps. Indeed, the addition of a clause C to a partially learned theory T_i does not change the set of consequences of an independent clause, whose set of negative examples can neither increase nor decrease. Therefore, by caching the list of negative examples, the learning system can prevent its computation.

A different observation concerns the list of positive examples to be covered by the partially learned theory. For the same reason reported above it cannot increase, while it can decrease since some of the positive examples might have been covered by the added clause C. Actually, the set of positive examples of a clause C' generated at the $(i+1)$-th step can be calculated as intersection of the cached set computed at the i-th step of the learning strategy and the set of positive examples covered by the parent clause of C' in the specialization hierarchy computed at the $(i+1)$-th step (see Fig. 3). In the case of dependent clauses, both lists of the positive and negative examples can increase, decrease or remain unchanged, since the addition of a clause C to a partially learned theory T_i might change the set of consequences of a dependent clause. Therefore, caching the set of positive/negative examples covered by a dependent clause is useless.

It is noteworthy that, differently from the caching technique for clause generation, caching for clause evaluation does not require additional memory resources since all requested information are kept from the current learning step.

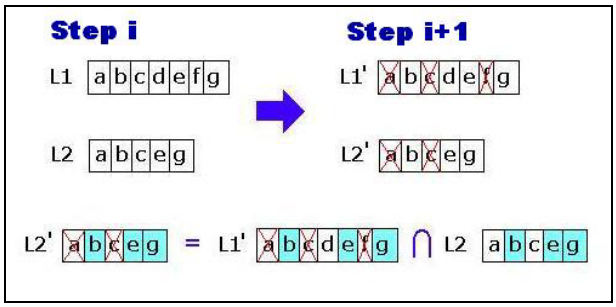

Fig.3. The positive examples list is calculated as intersection of the positive examples list of the same clause in previous learning step (i) and the positive examples list of the parent clause in current learning step (i+1).

4. Application to document understanding

The current release of ATRE is implemented in Sictus Prolog and C. It has also been integrated in WISDOM++(http://www.di.uniba.it/~malerba/wisdom++), an intelligent document processing system, that uses logical theories learned by ATRE to perform automatic classification and understanding of document images. In this section we show some experimental results for the document image understanding task alone.

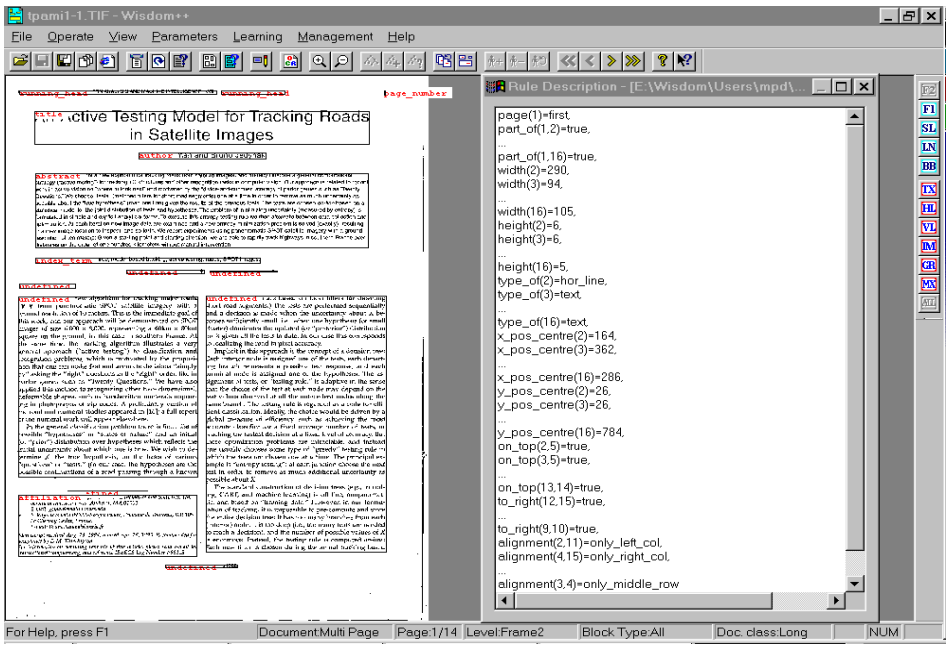

Fig. 4. Layout of a document image produced by WISDOM++ (left) and its partial description in the logical representation language adopted by ATRE (right).

A document is characterized by two different structures representing both its internal organization and its content: the layout structure and the logical structure. The former associates the content of a document with a hierarchy of layout components, while the latter associates the content of a document with a hierarchy of logical components. Here, the term document understanding denotes the process of mapping the layout structure of a document into the corresponding logical structure. The document understanding process is based on the assumption that documents can be understood by means of their layout structures alone. The mapping of the layout structure into the logical structure can be performed by means of a set of rules which can be generate automatically by learning from a set of training objects. Each training object describes the layout of a document image and the logical components associated to layout components (see Fig. 4).

To empirically investigate the effect of the proposed caching strategies, we selected twenty-one papers, published as either regular or short, in the IEEE Transactions on Pattern Analysis and Machine Intelligence, in the January and February issues of 1996. Each paper is a multi-page document; therefore, the dataset is composed by 197 document images in all. Since in the particular application domain, it generally happens that the presence of some logical components depends on the order page (e.g. *author* is in the first page), we have decomposed the document understanding problem into three learning subtasks, one for the first page of scientific papers, another for intermediate pages and the third for the last page. Target predicates are only unary and concern the following logical components of a typical scientific paper published in a journal: *abstract*, *affiliation*, *author*, *biography*, *caption*, *figure*, *formulae*, *index_term*, *page_number*, *references*, *running_head*, *table*, *title.* Some statistics on the dataset obtained from first page documents are reported in Table1.

Logical components	***Number of positive examples***
Abstract	21
Affiliation	22
Author	25
Index_term	11
Page_number	180
Running_head	203
Title	23
Total	**485**

Table 1. Distribution of logical components on first page documents.

By running ATRE on a document understanding dataset obtained from scientific papers, a set of theories is learned. Some examples of learned clauses follow:

```
author(X1) ← alignment(X1,X2)=only_middle_col, abstract(X2),
             height(X1)∈[7..13]
figure(X1) ← type_of(X1)= image, width(X1)∈[12..227],
             x_pos_centre(X1)∈[335..570]
references(X1) ← to_right(X1,X2), biografy(X2),
```

```
width(X2)∈[261..265]
```

They can be easily interpreted. For instance, the first clause states that if a quite short layout component (X1), whose height is between 7 and 13, is centrally aligned with another layout component (X2) labelled as the abstract of the scientific paper, then it can be classified as the author of the paper. These clauses show that ATRE can automatically discover meaningful dependencies between target predicates.

In the experiments the effect of the caching is investigated with respect to two system parameters, the minimum number of *consistent* clauses found at each learning step before selecting the best one and the beam of the search (*max_star* parameter). The former affects the depth of specialization hierarchies, in the sense that the higher the number of consistent clauses, the deeper the hierarchies. The latter affects the width of the search-tree.

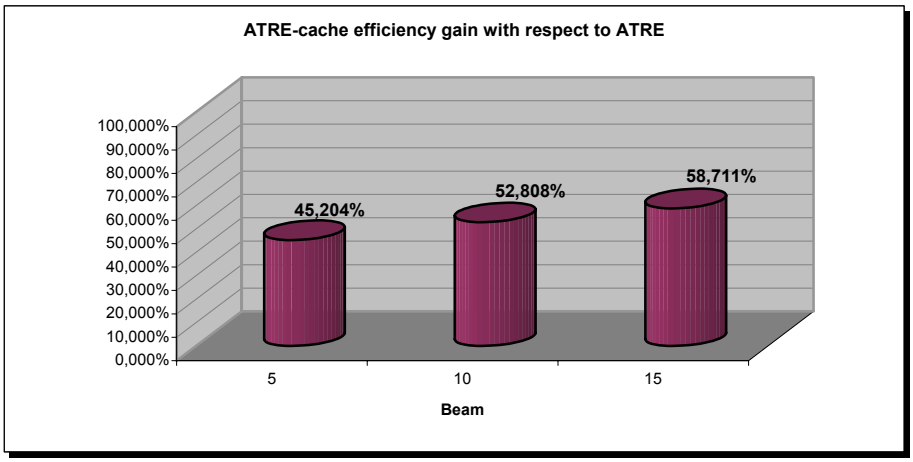

Fig. 5 Efficiency gain on "First-page" task on *max_star* variation.

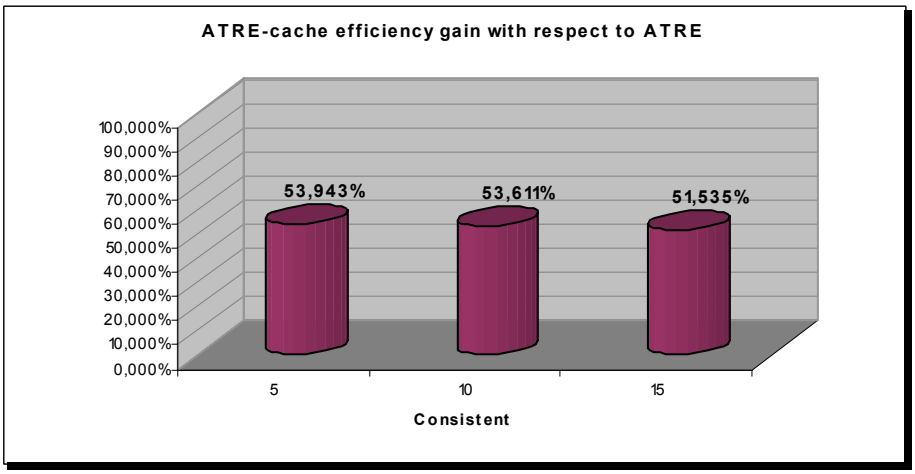

Fig. 6. Efficiency gain on "First-page" task on *consistent* variation.

Results for the first page learning task are shown in Figures 5 and 6. Percentages refer to reduction of the learning time required by ATRE with caching with respect to the original release of the system (without caching). Results show a positive dependence between the size of the beam and the reduction of the learning time. On the contrary, slight increases in the number of consistent clauses do not seem to significantly affect the efficiency gain due to caching.

5. Conclusions

In this paper issues and solutions of recursive theory learning are illustrated. Evolutions on the proposed search strategy to tackle efficiency problems are proposed. They have been implemented in ATRE and tested in the document understanding domain. Initial experimental results show that the learning task benefits from the caching strategy. As future work we plan to perform more extensive experiments to investigate the real efficiency gain in other real-world domains.

References

1. Apt, K.R.: Logic programming. In: van Leeuwen, J. (ed.): Handbook of Theoretical Computer Science, Vol. B. Elsevier, Amsterdam (1990) 493-574.
2. Boström, H.: Induction of Recursive Transfer Rules. In J. Cussens (ed.), Proceedings of the Language Logic and Learning Workshop (1999) 52-62.
3. Buntine, W.: Generalised subsumption and its applications to induction and redundancy. Artificial Intelligence, Vol. 36 (1988) 149-176.
4. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Trans. on Knowledge & Data Engineering 1(1) (1989) 146-166.
5. De Raedt, L., Dehaspe, L.: Clausal discovery. Machine Learning Journal, 26(2/3) (1997) 99-146.
6. De Raedt, L., Lavrac, N.: Multiple predicate learning in two Inductive Logic Programming settings. Journal on Pure and Applied Logic, 4(2) (1996) 227-254.
7. Khardon, R.: Learning to take Actions. Machine Learning, 35(1) (1999) 57-90.
8. Malerba, D.: Learning Recursive Theories in the Normal ILP Setting, Fundamenta Informaticae, 57(1) (2003) 39-77.
9. Mitchell, T.M.: Machine Learning. McGraw-Hill (1997).
10. Muggleton, S., Bryant, C.H.: Theory completion using inverse entailment. In: J. Cussens and A. Frisch (eds.): Inductive Logic Programming, Proceedings of the 10th International Conference ILP 2000, LNAI 1866, Springer, Berlin, Germany (2000) 130-146.
11. Nienhuys-Cheng, S.-W., de Wolf, R.: A complete method for program specialization based upon unfolding. Proc. 12th Europ. Conf. on Artificial Intelligence (1996) 438-442.
12. Nienhuys-Cheng, S.-W., de Wolf, R.: The Subsumption theorem in inductive logic programming: Facts and fallacies. In: De Raedt, L. (ed.): Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996) 265-276.
13. Plotkin, G.D.: A note on inductive generalization. In: Meltzer, B., Michie, D. (eds.): Machine Intelligence 5. Edinburgh University Press, Edinburgh (1970) 153-163.
14. Plotkin, G.D.: A further note on inductive generalization. In: Meltzer, B., Michie, D. (eds.): Machine Intelligence 6. Edinburgh University Press, Edinburgh (1971) 101-124.
15. Rouveirol, C.: Flattening and saturation: Two representation changes for generalization. Machine Learning Journal, 14(2) (1994) 219-232.