On the Effect of Caching in Recursive Theory Learning

Margherita Berardi, Antonio Varlaro, and Donato Malerba

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

Abstract. This paper focuses on inductive learning of recursive logical theories from a set of examples. This is a complex task where the learning of one predicate definition should be interleaved with the learning of the other ones in order to discover predicate dependencies. To overcome this problem we propose a variant of the separate-and-conquer strategy based on parallel learning of different predicate definitions. In order to improve its efficiency, optimization techniques are investigated and adopted solutions are described. In particular, two caching strategies have been implemented and tested on document processing datasets. Experimental results are discussed and conclusions are drawn.

1 Introduction

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 into 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:

$$\begin{gathered} odd(X) \leftarrow succ(Y,X), even(Y) \\ even(X) \leftarrow succ(Y,X), odd(Y) \\ even(X) \leftarrow zero(X) \end{gathered}$$

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 [4, 14]. Despite this scepticism, in the literature it is possible to find several ILP applications in which recursion has proved helpful [10]. Moreover, many ILP researchers have shown some interest in multiple predicate learning [9], 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, $1\leq i\leq r$
- A background theory BK
- A language of hypotheses 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, $1\leq i\leq r$, $BK\cup T \models E_i^+$ (*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 [17], is not sufficient to guarantee the completeness and consistency of learned definitions, with respect to logical entailment [16]. 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 [8]. 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 some solutions have been proposed in [12] 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.

In this paper we focus on the main problem of the interleaving of the learning of one (possible recursive) predicate definition with the learning of the other ones. In particular, different aspects of the adopted strategy for the automated discovery of predicate dependencies, namely the separate-and-parallel-conquer strategy, are presented. Efficiency problems due to the computational complexity of the search space are also discussed and some solutions implemented in a new version of the system ATRE are described.

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

2 The Learning Strategy

2.1 The Separate-and-Parallel-Conquer Search

The high-level learning algorithm in ATRE belongs to the family of *sequential covering* (or *separate-and-conquer*) algorithms [13] since it is based on the strategy of learning one clause at a time, removing the covered examples and iterating the process on the remaining examples. Indeed, 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

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

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 $i \in \{0, 1, \ldots, n-1\}$, 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.

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($T_i \cup \{C\}$)) = $\emptyset$. 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. The approach followed in ATRE to remove inconsistency due to the addition of a clause to the theory consists of simple syntactic changes in the theory, which eventually creates new *layers*, just as the stratification of a normal program creates new strata [1]. Details on the layering approach and on the computation method are reported in [12]. The layering of a theory introduces a first variation of the classical separate-and-conquer strategy sketched above, since the addition of a locally consistent clause generated in the conquer stage is preceded by a global consistency check.

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. The main problem with the well-known θ-subsumption is that the objects of comparison are two clauses and no additional source of knowledge (e.g., a theory T) is considered. Instead, we are only interested in those relative generality orders that compare two clauses relatively to a given theory T. In ATRE, a new generalization order named *generalized implication* is adopted [12], since both Buntine's *generalized subsumption* [5] and Plotkin's [17,18] notion of *relative generalization* are not appropriate (they are either too strong or too weak).

A solution to the problem of automated discovery of dependencies between target predicates $p_1, p_2, \ldots, p_r$ 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/recursive predicate 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.[1]

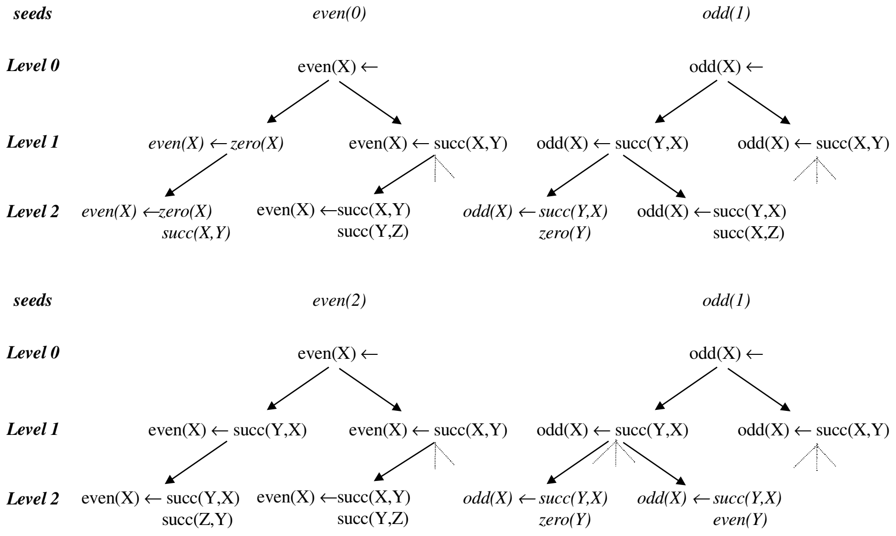

Fig. 1. Two steps (up and down) of the parallel search for the predicates *odd* and *even*. Consistent clauses are reported in italics.

The forest can be processed in parallel by as many concurrent tasks as the number of search-trees (hence the name of separate-and-parallel-conquer for this search strategy). 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

[1] A discussion on properties of this operator is beyond the scope of this paper. A thorough description of upward and downward refinement operators can be found in [16].

are found first, independently of the predicates to be learned.[2] 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.

2.2 Some Refinements

The learning strategy reported in the previous section is quite general and there is room for several distinct implementations. In particular, the following some points have been left unspecified: 1) how seeds are selected; 2) 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. According to the classical ILP learning setting, the set of training examples is a set of ground atoms. In this case, the choice of seeds should be stochastic because of the large number of candidate seeds. However, a random choice does not guarantee that the right seeds are chosen for the generation of the base clauses of the recursive definition. For this reason ATRE adopts a variant of the learning interpretation setting, where training examples of target predicates p_1, p_2, …, p_r are partitioned into training objects, each of which also includes a set of ground facts from the extensional BK. As observed by [3] within the setting of learning from interpretations, it is possible to develop more efficient learning algorithms than the classical ILP setting. This is especially true for recursive theory learning. Indeed, the object-centered representation adopted by ATRE has the advantage of reducing the number of candidate seeds. The main assumption made in this approach is that *each object contains examples explained by some base clauses of the underlying recursive theory*.[3] 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 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

[2] 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.

[3] 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.

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.

The second 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

Considering the separate-and-parallel-conquer search sketched in Section 2.1, it presents some efficiency problems and leaves 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. Intuitively, caching the specialization hierarchies explored at a certain step of the separate-and-conquer strategy and reusing part of them at the following step, seems to be a good strategy to decrease the learning time while keeping memory usage under acceptable limits. Furthermore, clause evaluation requires a number of generalized implication tests, one for each positive or negative example. Although the generalized implication test is optimized in ATRE, when the number of tests to perform is high, the clause evaluation leads to efficiency problems anyway. To reduce the number of tests, a caching method on the list of positive and negative examples of each clause has been investigated.

In this section we present the novel caching strategy implemented in ATRE to solve efficiency problems above. Generally speaking, caching aims to save useful information that would be repeatedly recomputed otherwise, with a clear waste of time. In particular, the proposed strategy 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

To prevent the exploration of the same specialization hierarchies several times, we propose a caching mechanism that aims to save the specialization hierarchies explored at the *i*-th step of the separate-and-conquer strategy so to reuse part of them at the $(i+1)$-th step.

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 *Ti* 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.

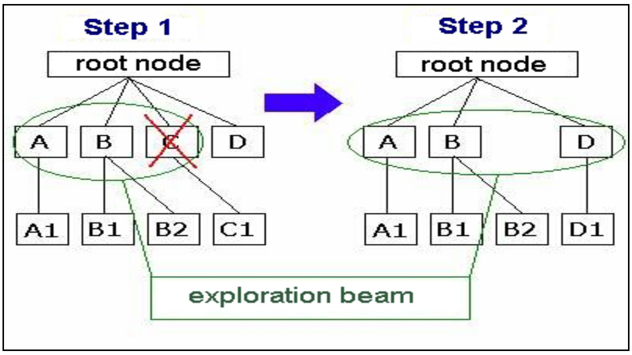

Fig. 2. An example of search-tree pruning effect on beam search width.

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

To clarify the caching technique proposed for the clause evaluation phase, we need to 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.

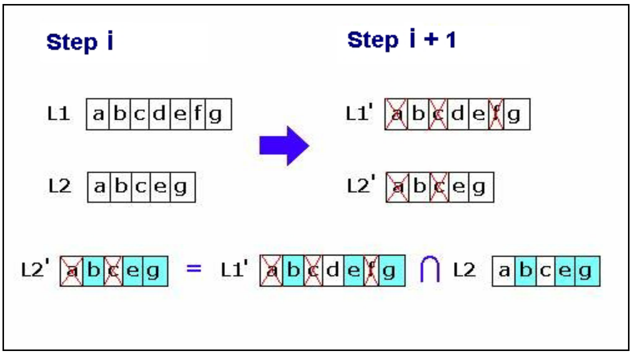

Fig. 3. An example of the positive examples list computation as intersection of the positive examples list of the same clause in the previous learning, step (i), and the positive examples list of the parent clause in the current learning, step (i+1).

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 (see Section 5.1.3).

4 Related Work

Efficiency of ILP systems is strongly dependent on the strategy adopted while searching the space of candidate hypotheses. Different techniques have been investigated to improve efficiency. A common approach is to reduce the number of hypotheses to evaluate during the search. Language bias specification to constraint the number of hypotheses [15] and branch-and-bound as well as heuristic searches have been widely employed in this perspective.

The caching of information on clauses coverage seems instead to be a convenient solution when exhaustive search is carried out. Indeed, [7] implements in Progol the caching of positive and negative cover of clauses during the search. Results show an improvement expressed by a 15.75 speed up factor. This approach is very similar to the caching strategy adopted in the clause evaluation phase of ATRE. Nevertheless, it presents some differences due to the additional difficulties raised by recursive theory learning. Indeed, as explained in Section 3.2, ATRE has to take into account the distinction between dependent/independent clauses in order to decide whether caching strategy should be applied or not. In addition, Progol implements a *prune cache* method that aims to cache clauses representing points of the hypothesis space where the search should be pruned.

Other approaches focusing on clause evaluation phase have been investigated by [6] whose work has been later extended by [2] in the particular ILP setting that considers examples of predicate definition to be learned as composing a database to repeatedly query. In this context, a query (clause) is a conjunction of goals (literals) to refine in order to generate a predicate definition. In particular, [6] propose a method to reduce the number of literals in clause bodies in order to optimize the number of predicate calls. The method is based on redundant literal reduction by means of a subsumption relation test. They also propose an optimisation of the number of clause coverage tests by grouping dependent body literals in equivalence classes and adding a cut predicate between literals of different equivalence classes. This method based on the cut introduction has been generalized by [2], which also implement a strategy to remove literals whose success is known and that will not influence the success of the examined clause.

[19] propose a method to minimize clause evaluation costs that is based on the reordering of dependent literals. In particular, they look for clause transformations with the shortest execution time. The idea is inspired by relational database management systems area and it is based on the observation that first order clauses are more efficient if “selective” literals are placed first.

All these last approaches address the goal of minimizing the number of theorem-proving for clauses to evaluate. In ATRE, the problem of managing redundancy or ordering of literals has never been tackled but it suggests interesting directions for further improvement.

5 Experimental Results

To evaluate the efficacy of the implemented caching strategies we performed extensive comparisons on the document understanding domain which is a source of interesting data sets for multiple predicate learning [11]. For this purpose, release of ATRE with caching has been also integrated in WISDOM++

(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. We show experimental results for the document image understanding task alone.

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 automatically learned from a set of training objects. Each training object describes the layout of a document image and the logical components associated to layout components.

In our empirical study on 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 (*tpami*), 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 pages of scientific papers, another for intermediate pages and the third for the last pages. 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.*

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), biography(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.

5.1 Settings and Results

Experiments have been conducted in order to compare running time of the standard version of the system (*ATRE*) against the running time of ATRE with caching (*ATRE-cache*). Three different experiments have been performed to investigate different factors affecting the use of caching varying the dataset setting. In particular, we evaluate the caching effect in the following settings: the set of 21 documents are divided into 5 folds according to a 5-fold cross-validation; the training set is incrementally built by adding 3 documents each time until all the 21 documents are taken into account; the learning parameters that affect the size of the search space are set to different values.

5.1.1 Experiment 1

In the first experiment, the 21 *tpami* documents have been divided in five folds removing some documents in turn (Table 1).

Table 1. Distribution of pages and examples per document grouped by 5 folds.

Fold number	List of documents	Number of pages	Number of positive examples	
			First pages	Last pages
1	Tpami1	13	27	13
	Tpami13	3	12	5
	Tpami14	10	26	14
	Tpami 16	14	34	19
2	Tpami8	5	16	6
	Tpami15	15	35	20
	Tpami18	10	24	12
	Tpami24	6	16	7
3	Tpami3	15	34	17
	Tpami7	6	16	6
	Tpami12	6	14	6
	Tpami20	14	34	16
4	Tpami9	5	15	7
	Tpami11	6	15	7
	Tpami19	20	45	24
	Tpami21	11	25	16
5	Tpami4	14	31	15
	Tpami6	1	6	2
	Tpami10	3	11	3
	Tpami17	13	31	17
	Tpami23	7	18	9

For each fold, the two versions of ATRE have been run on the four remaining folds. Besides, only the two learning tasks of first and last document pages have been examined. Execution time of each learning task, efficiency gain rates and caching rates are reported in Table 2 and Table 3. The caching rate is computed as the average on the percentages of cached clauses over the total number of clauses at each learning step. In particular, the caching rate estimates the caching effort in the clause generation phase. Running times refer to executions performed on a 1.4 Ghz IBM Centrino notebook equipped with 512 Mb of RAM.

Table 2. Running times, efficiency gain rates and caching rates for the first page learning task.

First pages	*ATRE*	*ATRE-cache*		
Fold No	**Execution time**	**Execution time**	**Caching rate**	**Time gain rate**
1	3095,470	1203,706	65,240%	61,114%
2	3096,785	1282,777	68,888%	58,577%
3	2545,410	1156,807	70,292%	54,553%
4	2790,321	1206,513	73,645%	56,761%
5	2851,612	1238,228	68,286%	56,578%
Mean values	2875,920	1217,606	69,270%	57,662%

Table 3. Running times, efficiency gain rates and caching rates for the last page learning task.

Last pages	*ATRE*	*ATRE-cache*		
Fold No	**Execution time**	**Execution time**	**Caching rate**	**Time gain rate**
1	2199,972	1381,098	31,726%	37,222%
2	1681,480	1317,043	34,102%	21,674%
3	1817,711	1254,341	35,878%	30,993%
4	1641,837	1117,979	44,403%	31,907%
5	2016,882	1299,936	35,286%	35,547%
Mean values	1871,576	1274,079	36,279%	31,925%

Results reported in the Table 2 and 3 show that the running time gain as well as the caching rates vary with respect to the training fold because it may happen that some folds are composed by more complex training objects than others. Moreover, the above tables show that the two rates are not so proportional as one can expect. This is because the caching rate is only related to the clause generation phase, while the time gain rate is related to both the clause generation and evaluation phases. Interesting observations can arise relating trends on caching and time gain rates.

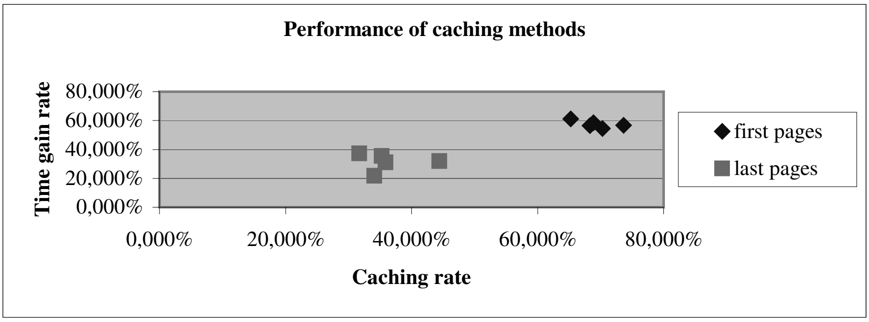

Fig. 4. Time gain rates on the varying of the caching rate.

Fig. 4 shows that the two tasks generate two distinct clusters. No definite relationship seems to characterize the use of caching with respect to the efficiency gain inside each cluster. In addition, we observe that the cluster generated for the last pages task is further down positioned with respect to the cluster generated for the first pages task. This means that the employment of caching strongly improves the running time in first pages task while does not significantly affect the last pages task. This is because in the case of first page documents, the system has more target predicate definitions to learn than in the case of last pages documents. Consequently, the search space in last pages task is composed by few specialization hierarchies. Hence, every seed change (i.e. a search-tree to remove) more heavily affects the search space portion to be re-explored. Obviously, this is because the caching rate is computed as the average of rates at each learning step. A more significant information could be provided by also evaluating the caching rate standard deviation. Indeed, for the last pages task, the standard deviation varies between 27 and 44, while for the first pages task it varies between 68 and 73. This suggests that on the first pages task, the effect of caching is more uniform than in the other task. The higher efficiency gain as well as the more uniform variation of caching rate for the first pages task are due to a more homogeneous distribution of examples in the training set. From a more general point of view, the dataset peculiarities affect both the caching rate and the time gain rate since whenever one of the two measures is high, the other one is high too.

5.1.2 Experiment 2

In this experiment we progressively increase the size of the dataset. The two versions of the system have been run on a set of 3, 6, 9, 12, 15, 18 and 21 *tpami* documents. For each dataset trends of the efficiency gain rate and the caching rate for first and last pages tasks are shown in Fig. 5 and Fig. 6, respectively.

Graphs in Fig. 5 and 6 show that both the measures tend to approximately follow the same trend. In the first pages task, both the caching rate and the time gain rate are quite constant, that is the effect of caching has no influence on performances while the dataset grows. In the last pages task, both the rates decrease starting from the dataset with 12 documents.

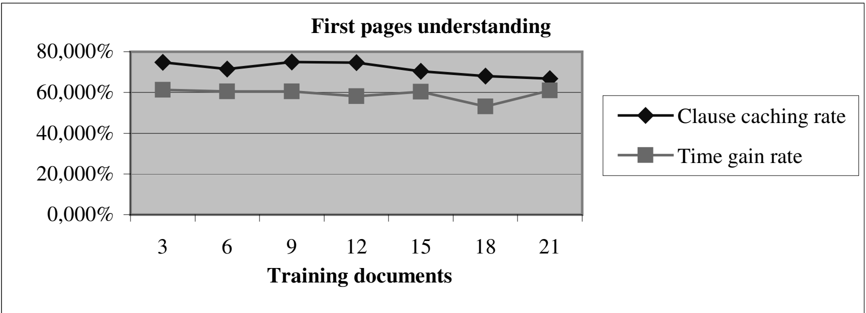

Fig. 5. Time gain rate and caching rate in first page document task varying the complexity of datasets in terms of examples number.

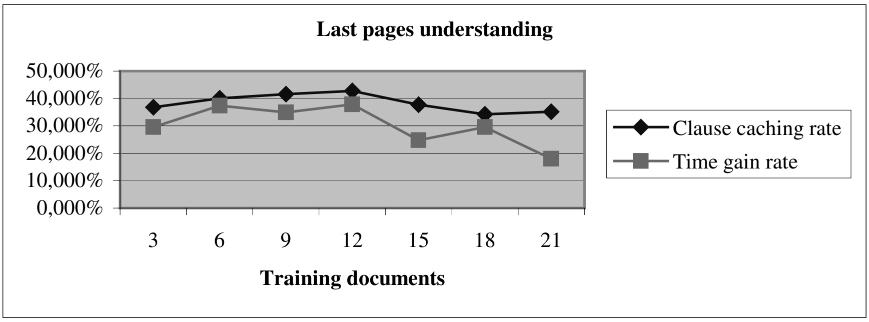

Fig. 6. Time gain rate and caching rate in last page document task varying the complexity of datasets in terms of examples number.

When we relate the two measures (Fig. 7), we can observe that in the case of last pages task there is a fairly good dependence between the caching and the time gain rates because when the caching rate raises, the time gain tends to raise too. Moreover, both the measures are affected by peculiarities of each task, because as in the first experiment, when one of the two rates is high, the other one tends to be high too, and vice versa.

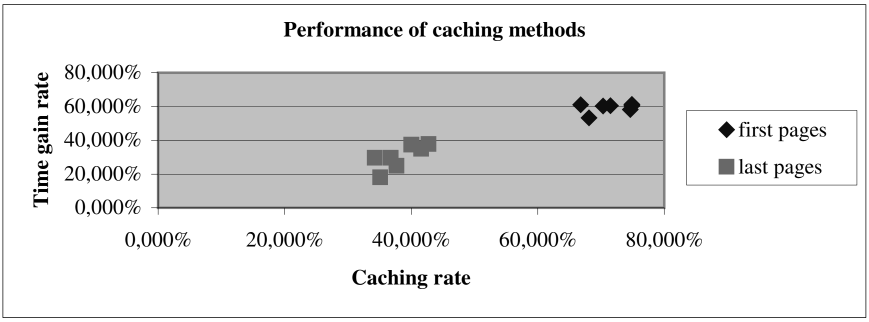

Fig. 7. Distribution of time gain and caching rates for first and last pages tasks varying the complexity of datasets in terms of examples number.

5.1.3 Experiment 3

In this experiment 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. The former affects the depth of specialization hierarchies, because the higher the number of consistent clauses is, the deeper the hierarchies are. The latter affects the width of the search-tree.

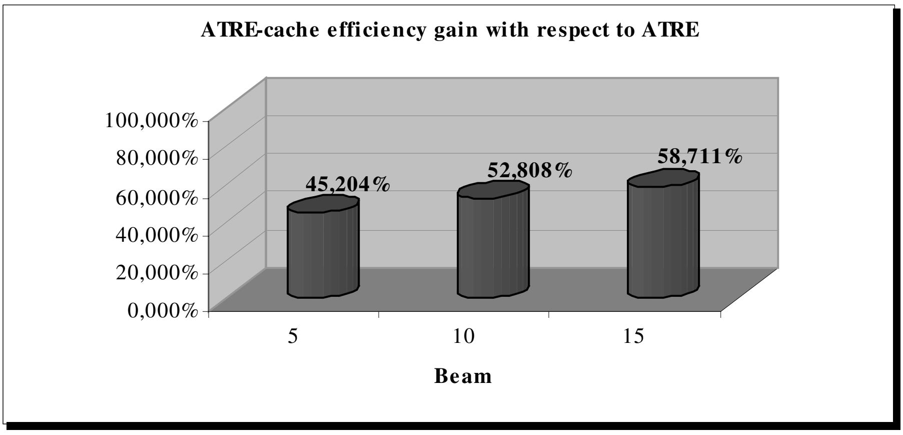

Fig. 8. Efficiency gain on first pages task on beam value variation.

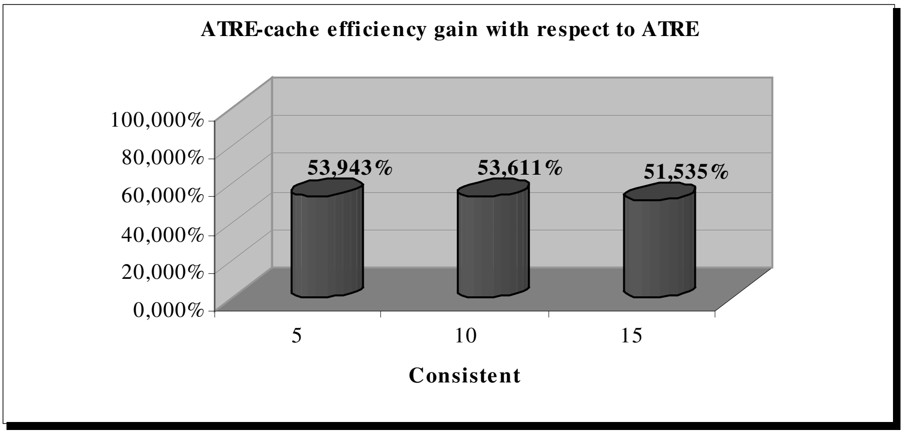

Fig. 9. Efficiency gain on first pages task on *consistent* variation.

Results for the first page learning task are shown in Fig. 8 and 9. Percentages refer to efficiency gain in time of *ATRE-cache* with respect to *ATRE*. Results show a positive dependence between the size of the beam and the efficiency gain rate. On the contrary, slight increases in the number of consistent clauses do not seem to significantly affect the efficiency gain due to caching.

From the memory use point of view, the application of caching leads to additional memory requirements. As an example, the memory use of *ATRE-cache* with respect to *ATRE* for the last pages learning task is reported in Fig. 10. Results show that the additional memory need is directly affected by the size of the beam. Results computed with respect to *consistent* variation do not point out any dependency between further memory requirements and the parameter variation.

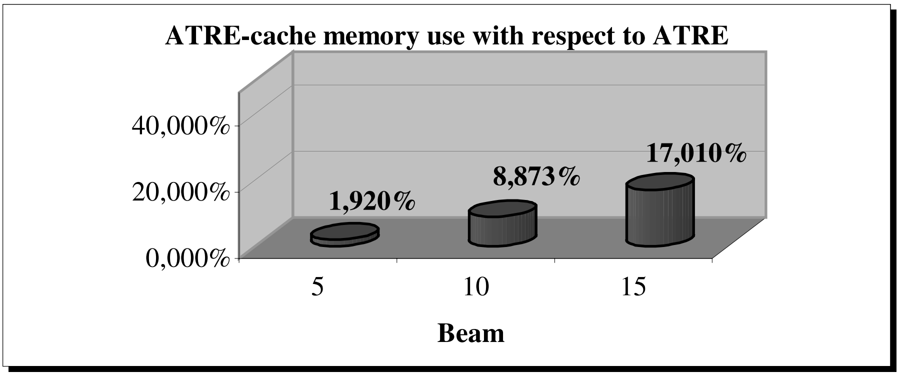

Fig. 10. Memory use on last pages task on beam value variation.

5.1.4 Accuracy and Structure of Induced Theories

Concerning the induced theories, we can observe that sometimes slight differences occur both in the number of learned clauses and in the order in which they have been learned. Considering that the clause generation caching implies the storage of the sequence of clauses to be explored, this can influence the order that the learning strategy "naturally" follows. Indeed, generally it may happen that going down from a learning step to the following step, the learning strategy generates the same clauses by following a different sequence. To evaluate the effect of caching on the accuracy of the learning problem, we compare results of a 5-fold cross-validation test performed on both the versions of the system. In particular, the set of 21 documents is firstly divided as reported in Table 1, and then, for every fold, *ATRE* and *ATRE-cache* are trained on the remaining folds and tested on the hold-out fold. For each learning problem, the number of omission/commission errors is recorded. O*mission* errors occur when logical labelling of layout components are missed, while *commission* errors occur when wrong logical labelling are "recommended" by a rule. Experimental results are reported in Table 4 for each trial, and the average number of omission and commission errors is also given. We can conclude that the caching does not significantly affect the accuracy.

Table 4. Accuracy of induced theories in the first page documents task.

Fold	***ATRE***		***ATRE-cache***	
	Omissions	**Commissions**	**Omissions**	**Commissions**
1	21/99	15/4080	20/99	12/4080
2	13/91	16/4697	12/91	11/4697
3	17/98	15/4781	18/98	17/4781
4	19/100	14/5318	19/100	14/5318
5	31/97	8/3746	30/97	9/3746
Average %	20,76	0,30	20,34	0,28
Std Dev. %	6,75	0,06	6,50	0,05

6 Conclusions

In this paper evolutions on a search strategy for recursive theory learning 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-word domains. As future work, further improvements concerning the quality of induced theories are worth to be investigated. The complexity and comprehensibility of learned theories are affected by the application of the layering technique and an optimisation on the number of layering operations can be profitably feasible.

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. Blockeel H., Demoen B., Jansseens G., Vandecasteele H., Van Laer W.: Two Advanced Transformations for Improving the Efficiency of an ILP System, In J. Cussens and A. Frisch (ed.), Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming (2000) 43-59.
3. Blockeel H., De Raedt L., Jacobs N., and Demoen B.: Scaling up inductive logic programming by learning from interpretations. Data Mining and Knowledge Discovery, 3(1) 59-93 (1999).
4. Boström H.: Induction of Recursive Transfer Rules. In J. Cussens (ed.), Proceedings of the Language Logic and Learning Workshop (1999) 52-62.
5. Buntine, W.: Generalised subsumption and its applications to induction and redundancy. Artificial Intelligence, Vol. 36 (1988) 149-176.
6. Costa V. S., Srinivasan A., Camacho R.: A note on two simple trasformations for improving the efficiency of an ILP system, Proceedings of the 10th International Conference on Inductive Logic Programming, Lecture Notes in Artificial Intelligence, Springer-Verlag (2000).
7. Cussens J.: Part-of-speech tagging using Progol. In Inductive Logic Programming: Proceedings of the 7th Int. Workshop. Lecture Notes in Artificial Intelligence, vol.1297 93–108 Springer-Verlag (1997).
8. De Raedt, L., Dehaspe, L.: Clausal discovery. Machine Learning Journal, 26(2/3) (1997) 99-146.
9. De Raedt, L., and N. Lavrac: Multiple predicate learning in two Inductive Logic Programming settings. Journal on Pure and Applied Logic, 4(2) (1996) 227-254.
10. Khardon, R.: Learning to take Actions. Machine Learning, 35(1) (1999) 57-90.
11. Malerba D., Esposito F., Lisi F.A. and Altamura O.: Automated Discovery of Dependencies Between Logical Components in Document Image Understanding. Proceedings of the 6th International Conference on Document Analysis and Recognition, Seattle (WA), (2001) 174-178.
12. Malerba D.: Learning Recursive Theories in the Normal ILP Setting, Fundamenta Informaticae, 57(1) (2003) 39-77.
13. Mitchell, T.M.: Machine Learning. McGraw-Hill (1997).

14. Muggleton, S. and C.H. Bryant: Theory completion using inverse entailment. In: J. Cussens and A. Frisch (eds.): Inductive Logic Programming, Proceedings of the 10th Int. Conference ILP 2000, LNAI 1866, Springer, Berlin, Germany (2000) 130-146.
15. Nedellec C., Ad H., Bergadano F., and Tausend B.: Declarative bias in ILP. In L. De Raedt (ed.), Advances in Inductive Logic Programming, volume 32 of Frontiers in Artificial Intelligence and Applications (1996) 82-103, IOS Press.
16. 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.
17. Plotkin, G.D.: A note on inductive generalization. In: Meltzer, B., Michie, D. (eds.): Machine Intelligence 5. Edinburgh University Press, Edinburgh (1970) 153-163.
18. 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.
19. Struyf J. and Blockeel H.: Query optimisation in Inductive Logic Programming by Reordering Literals. In T. Horváth and A. Yamamoto (ed.), Proceedings of the 13th International Conference on Inductive Logic Programming, Lecture Notes in Artificial Intelligence, vol. 2835 (329-346). Springer-Verlag, 2003.