Theory-Guided Induction of Logic Programs by Inference of Regular Languages

Henrik Boström
Dept. of Computer and Systems Sciences
Stockholm University
Electrum 230, 164 40 Kista, Sweden
henke@dsv.su.se

Abstract

Previous resolution-based approaches to theory-guided induction of logic programs produce hypotheses in the form of a set of resolvents of a theory, where the resolvents represent allowed sequences of resolution steps for the initial theory. There are, however, many characterizations of allowed sequences of resolution steps that cannot be expressed by a set of resolvents. One approach to this problem is presented, the system MERLIN, which is based on an earlier technique for learning finite-state automata that represent allowed sequences of resolution steps. MERLIN extends the previous technique in three ways: i) negative examples are considered in addition to positive examples, ii) a new strategy for performing generalization is used, and iii) a technique for converting the learned automaton to a logic program is included. Results from experiments are presented in which MERLIN outperforms both a system using the old strategy for performing generalization, and a traditional covering technique. The latter result can be explained by the limited expressiveness of hypotheses produced by covering and also by the fact that covering needs to produce the correct base clauses for a recursive definition before producing the recursive clauses. MERLIN on the other hand does not require that particular examples of the base cases are given, since both base clauses and recursive clauses can be inferred from a single example.

1 Introduction

The search for a clause in an inductive hypothesis can be performed either top-down (i.e. from an overly general clause to a more specific) or bottom-up (i.e. from a overly specific clause to a more general). Several approaches to induction of logic programs search top-down from clauses in a theory (i.e. definite program[1]) by specializing the clauses using literal addition (e.g. [14, 13, 18][2]) or resolution (e.g. [9, 10, 4, 7, 5, 6]) or both (e.g. [3, 16, 15]).

In the previous approaches that use resolution only, the theory can be viewed as a declarative bias that restricts not only what predicate symbols can be used in learned clauses, but also how the predicates can be invoked. In these approaches the theory is used to identify some sequences of resolution steps that discriminate positive examples from negative examples when proved by the theory. For example, assume that we are given the following theory T:

($c1$) `p([]).`
($c2$) `p([a|L]):- p(L).`
($c3$) `p([b|L]):- p(L).`

together with the positive examples $E^+ = \{$`p([a,b])`, `p([a,b,a,b,a,b])`$\}$ and the negative examples $E^- = \{$`p([a])`, `p([a,b,a])`$\}$. Then one characterization of the sequences of resolution steps that are used in the SLD-refutations of the positive examples that does not hold for the SLD-refutations of the negative examples is that the clauses $c2$ and $c3$ should be used in succession an arbitrary number of times followed by $c1$. This can be expressed by the following resolvents of the previous program:

($c1$) `p([]).`
($c4$) `p([a,b|L]):- p(L).`

All previous resolution-based approaches represent the identified sequences of resolution steps as a set of resolvents of a given theory. However, there are many char-

[1] Familiarity with the standard terminology in logic programming is assumed [12].

[2] These systems may perform bottom-up search as well.

acterizations of what sequences of resolution steps are allowed that cannot be expressed as sets of resolvents of the theory. For example, given the theory T above, there is no way of expressing by a set of resolvents of T that clause $c2$ should be applied an arbitrary number of times followed by an arbitrary number of applications of $c3$ followed by $c1$. The reason for this is that a new predicate symbol is needed in order to express this, and such are never introduced when applying resolution.

In this work, we present one approach to this problem, a system called MERLIN[3]. It is based upon the idea in [8] to learn finite-state automata that represent allowed sequences of resolution steps. In the next section, we show how the learning procedure presented in [8], which only handles positive examples[4], can be slightly modified to deal with negative examples as well. Another extension to the technique, which has been included in MERLIN, is a novel strategy for performing generalization when producing the automaton. We also show how MERLIN uses the learned automaton in order to produce a resulting inductive hypothesis in the form of a logic program, by calculating the intersection of the automaton and a context-free grammar that represents the original theory. In section three, we present experimental results from comparing MERLIN to a traditional covering technique and from comparing the new strategy to the old for performing generalization when producing the automaton. Finally, in section four we give some concluding remarks and point out directions for future research.

2 MERLIN

In this section, we first present the procedure in MERLIN for learning a finite state automaton from a theory and positive and negative examples. We then show how the learned automaton is used in MERLIN to specialize the theory.

2.1 Regular Language Inference

The basic idea in [8] is that sequences of input clauses in SLD-refutations can be viewed as strings in a formal language, which can be represented by a finite-state automaton. The learning procedure in [8] can be summarized[5]: i) construct an initial automaton from a given sequence, ii) repeat as long as possible: merge two states that have out-going arcs labeled with the same symbol, and make the resulting automaton deterministic. In [8] it is also shown how to handle multiple sequences: construct initial automata for the sequences, merge the start states, make the automaton deterministic and then apply step ii above.

We have made two alterations to this learning procedure: i) negative examples are considered in addition to positive examples, which may prevent overgeneralization, and ii) a strategy for choosing what states to merge is incorporated, which may result in simpler automata.

The procedure for producing a generalized automaton takes as input two sets of sequences: one representing SLD-refutations of positive examples and one representing SLD-refutations of negative examples. One initial automaton is constructed that represent all the sequences, where the final states are classified as either belonging to a positive example or to a negative example. Furthermore, the arcs that lie on paths from the start state to positive final states are classified as positive, and the others as negative. States that have positive out-going arcs labeled with the same symbol are then merged as long as possible, and after each merge, the automaton is made deterministic and it is checked that no final positive state is merged to a final negative state. Moreover, instead of just merging any of the mergeable states, as done in [8], the merge that leads to the fewest number of positive arcs in the resulting deterministic automaton is chosen.

Below, we first give a formal description of the learning procedure and then illustrate it with an example. An automaton is internally represented by a tuple $(q_0, A^+, A^-, F^+, F^-)$, where q_0 is the start state, A^+ and A^- are the positive and negative arcs, where an arc from q_a to q_b labeled with $Clause$ is represented by $(q_a, Clause, q_b)$, and where F^+ and F^- are positive and negative final states. The final automaton is, however, represented by the triple (q_0, A^+, F^+), since there is no need in the following to consider negative arcs and negative final states.

FUNCTION Produce-Automaton(S^+, S^-)
  M := Initial-Automaton(S^+, S^-)
  REPEAT
    Let $M = (q_0, A^+, A^-, F^+, F^-)$
    $M_{succ} := \{$Make-Deterministic(Merge(q_a, q_b, M)): $q_a \neq q_b$ and $\{(q_a, s, q'_a), (q_b, s, q'_b)\} \subseteq A^+$ for some s, q'_a and $q'_b\}$
    IF $M_{succ} \neq \emptyset$ **THEN** $M := M' \in M_{succ}$ with the fewest positive arcs
  UNTIL $M_{succ} = \emptyset$
  RETURN (q_0, A^+, F^+)

[3] Model Extraction by Regular Language INference

[4] The procedure was used to learn recursive concepts in explanation-based learning.

[5] We use ordinary automata [11] rather than the input-output automata used in [8].

FUNCTION Initial-Automaton(S^+, S^-)
  $M := (q_0, \emptyset, \emptyset, \emptyset, \emptyset)$
  FOR EACH $c_1 \ldots c_n \in S^+$ **DO**
    $A^+_{new} := \{(q'_0, c_1, q'_1), \ldots, (q'_{n-1}, c_n, q'_n)\}$, where $q'_0, \ldots, q'_n$ are new states
    Let $M = (q_0, A^+, A^-, F^+, F^-)$
    $M :=$ Make-Deterministic(Merge($q_0, q'_0, (q_0, A^+ \cup A^+_{new}, A^-, F^+ \cup \{q'_n\}, F^-)$))
  FOR EACH $c_1 \ldots c_n \in S^-$ **DO**
    $A^-_{new} := \{(q'_0, c_1, q'_1), \ldots, (q'_{n-1}, c_n, q'_n)\}$, where $q'_0, \ldots, q'_n$ are new states
    Let $M = (q_0, A^+, A^-, F^+, F^-)$
    $M :=$ Make-Deterministic(Merge($q_0, q'_0, (q_0, A^+, A^- \cup A^-_{new}, F^+ \cup \{q'_n\}, F^-)$))
  RETURN M

FUNCTION Merge($q_a, q_b, (q_0, A^+, A^-, F^+, F^-)$)
  IF ($q_a \in F^+$ and $q_b \in F^-$) or ($q_a \in F^-$ and $q_b \in F^+$)
  THEN fail
  $q' :=$ a new state
  Replace all occurrences of q_a and q_b with q' in A^+, A^-, F^+ and F^-
  $A^- := A^- \setminus A^+$
  IF $q_a = q_0$ or $q_b = q_0$ **THEN** $S := q'$ **ELSE** $S := q_0$
  RETURN (S, A^+, A^-, F^+, F^-)

FUNCTION Make-Deterministic($(q_0, A^+, A^-, F^+, F^-)$)
  $M := (q_0, A^+, A^-, F^+, F^-)$
  WHILE $\{(q, s, q_a), (q, s, q_b)\} \subseteq A^+ \cup A^-$ in M, where $q_a \neq q_b$ **DO**
    IF Merge(q_a, q_b, M) succeeds **THEN**
      $M :=$ Merge(q_a, q_b, M) **ELSE** fail
  RETURN M

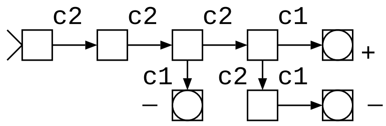
Example Given the following positive and negative examples:

$E^+ = \{$`nat(s(s(s(0))))`$\}$
$E^- = \{$`nat(s(s(0)))`, `nat(s(s(s(s(0)))))`$\}$

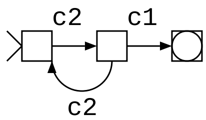
together with the following program:

($c1$) `nat(0).`
($c2$) `nat(s(X)):- nat(X).`

Then there is one SLD-refutation of the positive example, and hence one sequence in $S^+ = \{c2\ c2\ c2\ c1\}$. The set of sequences of input clauses in the SLD-refutations of the negative examples is $S^- = \{c2\ c2\ c1,\ c2\ c2\ c2\ c2\ c1\}$. The above sequences yield the following initial automaton, where the positive final state is marked with + and the negative final states are marked with $-$:



The only possible merge of two states with positive outgoing arcs labeled with the same symbols that results in a deterministic automaton is to merge the start state and the third state. Merging these, making the resulting automaton deterministic, and removing negative arcs gives the following final automaton:



It is easy to find cases where different decisions regarding what states to merge results in final automata accepting different languages. For example, if we in the previous case would have had an additional positive sequence $c2\ c2\ c2\ c2\ c2\ c2\ c2\ c1$, and chosen to merge the last two states that have outgoing arcs labeled with $c2$ on the path to the new positive final state, the resulting automaton would accept the first positive sequence as well as all sequences starting with five $c2$ or more followed by one $c1$. In section three, our strategy for choosing to merge states that result in the fewest positive arcs is empirically evaluated.

2.2 Specializing the Theory

Having learned an automaton that shows what sequences of input clauses are allowed in SLD-refutations of a given theory, MERLIN produces a new theory that allows only those sequences that are allowed by both the original theory and the learned automaton. This is done by first representing the set of sequences allowed by the given theory as a context-free grammar, and then deriving a new context-free grammar that represents the intersection of the first grammar and the learned automaton. The new grammar is then used to produce the final program.

2.2.1 Representing the Theory as a Context-Free Grammar

The set of possible sequences of input clauses in SLD-refutations of any instance of a goal G for a given program P can be represented by a context-free grammar, referred to as a *proof grammar*, (S, R), where S is the start symbol and R is a set of rules, where each rule is on the form $L \rightarrow CR_1 \ldots R_n$, where $n \geq 0$, $L, R_1 \ldots R_n$ are non-terminal symbols and C is a terminal symbol. Below, we show how to produce such a grammar followed by an example.

FUNCTION Proof-Grammar(G, P)
$S :=$ the predicate symbol of G
FOR EACH $C = (H : -B_1, \ldots, B_n) \in P$, where $n \geq 0$ **DO**
$R := R \cup \{L \rightarrow CR_1 \ldots R_n\}$, where L is the predicate symbol of H, and R_i is the predicate symbol of B_i, where $1 \leq i \leq n$.
RETURN (S, R)

Example Given the goal `{:- nat(N)}` together with the program in the previous examples. Then the above procedure produces the following proof grammar $(nat/1, R)$, where R is the following rules:

$nat/1 \rightarrow c1$
$nat/1 \rightarrow c2\ nat/1$

2.2.2 Deriving the Intersection

The intersection of a context-free language and a regular language is always a context-free language [2]. The following procedure for deriving a context-free grammar that represents the intersection of a proof grammar and a deterministic finite-state automaton is derived from the algorithm in [2].

FUNCTION Intersection$((S_{PG}, R_{PG}), (q_0, A, F))$
$NT := \{(L, S_1, S_2) : (L \rightarrow CR_1 \ldots R_n) \in R_{PG}, (S_1, C, S') \in A$, and S_2 is a state in A or 'empty'$\}$
$R := \emptyset$
FOR EACH pair $(L \rightarrow CR_1 \ldots R_n) \in R_{PG}$ and $(S_0, C, S_1) \in A$ **DO**
$R := R \cup \{(L, S_0, S_{n+1}) \rightarrow C(R_1, S_1, S_2) \ldots (R_n, S_n, S_{n+1}) : (R_i, S_i, S_{i+1}) \in NT$, for $1 \leq i \leq n\}$
IF $S_1 \in F$ and $n = 0$ **THEN**
$R := R \cup \{(L, S_0, empty) \rightarrow C\}$
RETURN $((S_{PG}, q_0, empty), R)$

Example Given the automaton and the proof grammar in the previous examples, the following rules is produced by the above procedure, together with the start symbol $(nat/1, q_0, empty)$[6]:

$(nat/1, q_0, empty) \rightarrow c2\ (nat/1, q_1, empty)$

$(nat/1, q_1, empty) \rightarrow c1$
$(nat/1, q_1, empty) \rightarrow c2\ (nat/1, q_0, empty)$

2.2.3 Producing the Final Program

Having derived the intersection of the learned automaton and the original proof grammar, MERLIN produces the final logic program, in which there is one clause for each rule in the intersection, and where each predicate symbol corresponds to a non-terminal symbol. The following procedure shows how this is achieved, given the intersection and the predicate symbol corresponding to the start symbol.

FUNCTION Make-Program$((S, R), T)$
$PS :=$ a bijection from the set of non-terminals in R to a set of predicate symbols, such that $PS(S) = T$
$P := \emptyset$
FOR EACH rule $(S_0 \rightarrow CS_1 \ldots S_n) \in R$, where $C = L_0 : -L_1, \ldots, L_n$ **DO**
$P := P \cup \{New_0 : -New_1, \ldots, New_n)\}$, where $New_i, 0 \leq i \leq n$, has the predicate symbol $PS(S_i)$ and the same arguments as L_i
RETURN P

Example Given the context-free grammar in the previous example, together with the predicate symbol $nat/1$, the above procedure produces the following program:

```
nat(s(X)):- nat2(X).
nat2(0).
nat2(s(X)):- nat(X).
```

3 Empirical Evaluation

In this section we present an empirical evaluation of the performance of MERLIN both with the new and the old strategy for selecting what merges to perform[7]. MERLIN was also compared to a traditional covering strategy, which is formalized in the appendix. We first present the three theories and four example sets that were used in the experiments and then the experimental results.

3.1 Domains

The first theory that was investigated is the same as presented in section 1:

$(c1)$ `p([]).`
$(c2)$ `p([a|L]):- p(L).`
$(c3)$ `p([b|L]):- p(L).`

Two sets of positive examples of `p(L)` were generated for this theory: one where `L` contained up to 20 elements representing the regular expression $(ab)^+$, and the other where `L` contained up to 7 elements representing the regular expression $a^*b^*a^*$. For each of these, a set of negative examples was generated by randomly replacing one of the elements in each positive

[6] *Dead rules* have been removed.

[7] The old strategy is simply to merge an arbitrary pair of states. The pairs are selected randomly in the experiments.

example with the other constant, such that the new instance did not belong to the set of positive examples[8]. The total number of examples in the first set was 20 (of which 50% were positive), and 104 in the second set (of which 60.6% were positive). It should be noted that a correct definition can, in principle, be produced by covering for the first example set, while this is not possible for the second set.

The second theory that was investigated is the same as presented in section 2:

($c1$) `nat(0).`
($c2$) `nat(s(X)):- nat(X).`

The entire set of examples consisted of the first 30 natural numbers, where the instances were classified as positive if they were divisible by two or three, and negative otherwise (this resulted in 63% positive examples). A correct definition of this concept can not be represented by a set of resolvents of the theory, and hence covering was deemed to fail in this domain.

The third theory that was investigated is shown below, where the second argument of `turing(D,M)` represents a sequence of movements of a Turing machine, where each move is on the form $(Read, Write, Move)$, and the first argument is a natural number representing the number of symbols to the left of the head. The set of positive examples consisted of sequences of movements of a Turing machine that starts at the leftmost position on the tape, and performs addition of the two natural numbers on the right side, where each is represented by between zero and five consecutive occurrences of the symbol `i` and where the two numbers are separated by a blank (`#`). The set of negative examples was generated by randomly replacing one of the movements in each positive example with some other incorrect movement. The total number of examples for this domain was 72 (of which 50% were positive). Note that although the learning problem in this domain was to recognize when addition has been correctly performed, any correct hypothesis that is produced can be used to guide a Turing machine to actually perform addition.

```
turing(D,[]).
turing(s(D),[(#,#,left)|Moves]):-
        turing(D,Moves).
turing(s(D),[(#,i,left)|Moves]):-
        turing(D,Moves).
turing(s(D),[(i,#,left)|Moves]):-
        turing(D,Moves).
turing(s(D),[(i,i,left)|Moves]):-
        turing(D,Moves).
turing(D,[(#,#,right)|Moves]):-
        turing(s(D),Moves).
turing(D,[(#,i,right)|Moves]):-
        turing(s(D),Moves).
turing(D,[(i,#,right)|Moves]):-
        turing(s(D),Moves).
turing(D,[(i,i,right)|Moves]):-
        turing(s(D),Moves).
```

[8]This is always possible for any example in the first set, but not for all examples in the second set.

3.2 Experimental Results

Each example set was randomly split into two halves, where one half was used for training and the other for testing. The number of examples in the training sets that were given as input to the algorithms were varied, where a larger set always included a smaller. The same training and test sets were used for all three techniques. Each experiment was iterated 50 times and the mean accuracy on the test examples is presented below.

In Figure 1, the results from the $(ab)^+$ domain are presented. In this domain, the strategy for choosing what merges to perform was of no importance, and the hypotheses produced by MERLIN with the new strategy were identical to the ones produced with the old strategy. However, it can be seen that MERLIN produced more accurate hypotheses than covering for all sizes of the training set, although covering in principle was able to produce a correct hypothesis in this domain. This can be explained by the fact that covering needed to produce the correct base clause for the recursive definition before producing the recursive clauses, and often none of the examples could be used to produce the correct base clause.

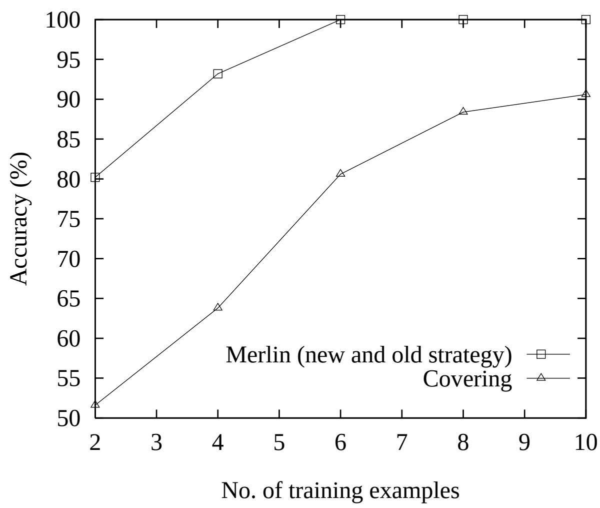


Figure 1: Accuracy for the $(ab)^+$ domain.

In Figure 2, Figure 3, and Figure 4, the results from the $a^*b^*a^*$ domain, divisible-by-2-or-3 domain, and the Turing machine domain are presented. As can been seen, the strategy for choosing what merges to perform was very important in these domains. MERLIN with the new strategy outperformed both MERLIN with the old strategy and covering in all three domains for almost all sizes of the training sets. That covering should not perform well in these domains was expected, since it was not even in theory able to produce correct hypotheses.

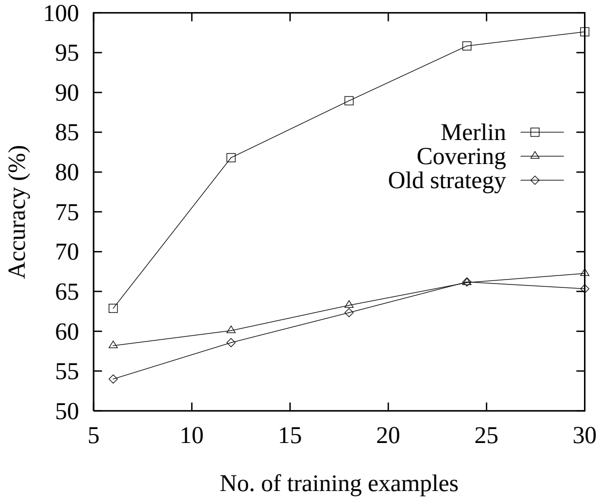


Figure 2: Accuracy for the $a^*b^*a^*$ domain.

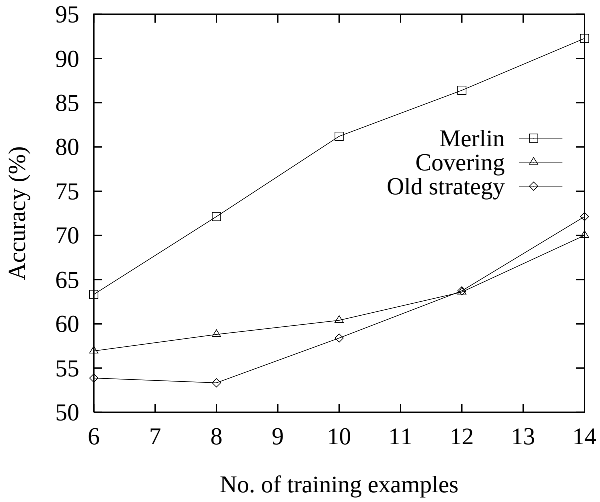


Figure 3: Accuracy for the divisible-by-2-or-3 domain.

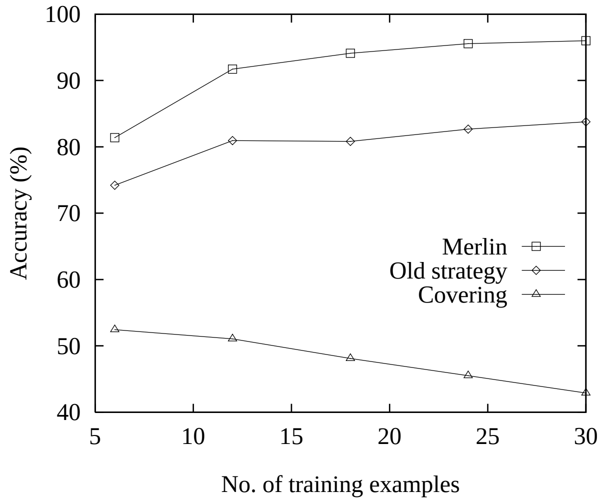


Figure 4: Accuracy for the Turing machine domain.

4 Concluding Remarks

We have presented a novel approach to theory-guided top-down induction of logic programs, the system MERLIN. The main motivation for this new approach is that previous resolution-based approaches represent learned definitions as sets of resolvents of a given theory, and many interesting concepts can not be represented in this way. The new approach is based on the technique in [8] to learn a finite-state automaton that represents sequences of allowed resolution steps. MERLIN extends the previous technique in three ways: i) negative examples are considered in addition to positive examples, ii) a new strategy for performing generalization is included, and iii) an inductive hypothesis is produced in the form of a logic program. Results from experiments have been presented in which the new strategy in MERLIN outperformed the old one, and also a traditional covering technique. The latter result can be explained by the limited expressiveness of the hypotheses produced by covering and also by the fact that covering needs to produce the correct base clauses for a recursive definition before producing the recursive clauses, and often none of the examples can be used to produce the correct base clauses. MERLIN on the other hand does not require that particular examples of the base cases are given, since the both the base clauses and recursive clauses can be inferred from a single example.

There are a number of possible directions for future research. One is to investigate other techniques for inferring finite-state automata (e.g. [1]) and to evaluate them using MERLIN. Another direction is to extend the expressiveness by using context-free grammars instead of finite-state automata, which also is investigated in [8]. A third direction for future research is to investigate how the computation rule affects the hypotheses produced by MERLIN. In all experiments, the SLD-refutations of the examples were produced using the standard Prolog computation rule (i.e. to select the leftmost goal to resolve upon). However, as demonstrated in [7], the result of the learning procedure can be highly dependent on how the SLD-refutations are derived.

Acknowledgements

This work has been supported by the European Community ESPRIT Long Term Research Project *Inductive Logic Programming II.* The author thanks Peter Idestam-Almquist for valuable discussions.

Appendix

In this appendix, we formalize the traditional covering strategy (i.e. until all positive examples are covered: select a clause that covers some positive examples, specialize it until it covers no negative examples, and then add it to the resulting hypothesis, which is initially empty). However, this technique can not be applied in a straight-forward manner when specializing recursive clauses. For example, consider the following program:

```
nat(0).
nat(s(X)):- nat(X).
```

together with the following positive and negative examples:

$E^+ = \{$`nat(s(0)), nat(s(s(s(0))))`$\}$
$E^- = \{$`nat(0), nat(s(s(0)))`$\}$

Since the coverage of each clause must be determined using the clauses found so far together with the background predicates, there is no clause in this case that covers any positive examples (we can not use the clauses in the overly general hypothesis as background predicates, since then we would never be able to produce a valid hypothesis). One solution to this problem is to introduce a new predicate, which is equivalent to the original predicate and to add a new clause with the new predicate in the head and the old predicate in the body. The above program would then be changed into the following:

```
nat(0).
nat(s(X)):- nat'(X).
nat'(0).
nat'(s(X)):- nat'(X).
nat'(X):- nat(X).
```

Now there is a clause defining the target predicate that together with the new background predicate covers some positive examples, and still all resolvents of the first program can be obtained from the second program. In the covering procedure below, it is assumed that the above transformation has been applied, and that the theory has been divided into two sets of clauses: O, which are clauses defining the target predicate, and B, which are clauses defining other predicates.

It should be noted that the decision of what resolvent to choose is left unspecified in the procedure, and in our experiments the resolvent that maximizes the information gain (c.f. [17]) was chosen.

FUNCTION Covering(O, B, E^+, E^-)
  $H := \emptyset$
  WHILE $E^+ \neq \emptyset$ **DO**
    $C :=$ a clause in O, which is used as the first input clause in an SLD-refutation of $\{\leftarrow e\} \cup \{C\} \cup H \cup B$ for some $e \in E^+$
    WHILE some $e \in E^-$ is covered by $\{C\} \cup H \cup B \cup E^+$ **DO**
      $C :=$ a resolvent R of C and a clause in B, for which there is a non-empty set $E^{\oplus}$, consisting of each $e \in E^+$, such that R is used as the first input clause in an SLD-refutation of $\{\leftarrow e\} \cup \{R\} \cup H \cup B$
    $E^+ := E^+ \setminus E^{\oplus}$
    $H := H \cup \{C\}$
  RETURN $H \cup B$

References

[1] Angluin D., “Learning Regular Sets from Queries and Counterexamples”, *Information and Computation*, **75**, (1987) 87–106

[2] Bar-Hillel Y., Perles M. and Shamir E., “On formal properties of simple phrase structure grammars”, *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, **14**, 1, Akademie Verlag, Berlin (1961) 143–172

[3] Bergadano F. and Giordana A., “A Knowledge Intensive Approach to Concept Induction”, *Proceedings of the Fifth International Conference on Machine Learning*, Morgan Kaufmann, CA (1988) 305–317

[4] Bergadano F. and Gunetti D. “Learning Clauses by Tracing Derivations”, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien, Gesellschaft für Mathematik und Datenverarbeitung MBH* (1994) 11–29

[5] Boström H., “Specialization of Recursive Predicates”, *Proceedings of the Eighth European Confer-*

ence on Machine Learning, Springer-Verlag (1995) 92–106

[6] Boström H., "Covering vs. Divide-and-Conquer for Top-Down Induction of Logic Programs", *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann (1995) 1194–1200

[7] Boström H. and Idestam-Almquist P., "Specialization of Logic Programs by Pruning SLD-Trees", *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien, Gesellschaft für Mathematik und Datenverarbeitung MBH* (1994) 31–48

[8] Cohen W. W., "Generalizing Number and Learning from Multiple Examples in Explanation Based Learning", *Proceedings of the Fifth International Conference on Machine Learning*, Morgan Kaufmann (1988) 256–269

[9] Cohen W. W., "The Generality of Overgenerality", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 490–494

[10] Cohen W. W., "Compiling Prior Knowledge Into an Explicit Bias", *Machine Learning: Proceedings of the Ninth International Workshop*, Morgan Kaufmann (1992) 102–110

[11] Lewis H. R. and Papadimitriou C. H., *Elements of the Theory of Computation*, Prentice-Hall (1981)

[12] Lloyd J. W., *Foundations of Logic Programming*, (2nd edition), Springer-Verlag (1987)

[13] Mooney R. J. and Ourston D., "Constructive Induction in Theory Refinement", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 178–182

[14] Ourston D. and Mooney R. J., "Changing the Rules: A Comprehensive Approach to Theory Refinement", *Proceedings of the Eighth National Conference on Artificial Intelligence*, MIT Press (1990) 815–820

[15] Pazzani M. and Brunk C., "Finding Accurate Frontiers: A Knowledge- Intensive Approach to Relational Learning", *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Morgan Kaufmann (1993) 328–334

[16] Pazzani M., Brunk C. and Silverstein G., "A Knowledge-Intensive Approach to Learning Relational Concepts", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 432–436

[17] Quinlan J. R., "Learning Logical Definitions from Relations", *Machine Learning* **5** (1990) 239–266

[18] Richards B. L. and Mooney R. J., "First-Order Theory Revision", *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann (1991) 447–451