Induction of Logic Programs by Example-Guided Unfolding

Henrik Boström and Peter Idestam-Almquist
Dept. of Computer and Systems Sciences
Stockholm University and Royal Institute of Technology
Electrum 230, 164 40 Kista, Sweden
{henke,pi}@dsv.su.se

Abstract

Resolution has been used as a specialisation operator in several approaches to top-down induction of logic programs. This operator allows the overly general hypothesis to be used as a declarative bias that restricts not only what predicate symbols can be used in produced hypotheses, but also how the predicates can be invoked. The two main strategies for top-down induction of logic programs, Covering and Divide-and-Conquer, are formalised using resolution as a specialisation operator, resulting in two strategies for performing example-guided unfolding. These strategies are compared both theoretically and experimentally. It is shown that the computational cost grows quadratically in the size of the example set for Covering, while it grows linearly for Divide-and-Conquer. This is also demonstrated by experiments, in which the amount of work performed by Covering is up to 30 times the amount of work performed by Divide-and-Conquer. The theoretical analysis shows that the hypothesis space is larger for Covering, and thus more compact hypotheses may be found by this technique than by Divide-and-Conquer. However, it is shown that for each non-recursive hypothesis that can be produced by Covering, there is an equivalent hypothesis (w.r.t. the background predicates) that can be produced by Divide-and-Conquer. A major draw-back of Divide-and-Conquer, in contrast to Covering, is that it is not applicable to learning recursive definitions.

1 Introduction

The search for a single clause in an inductive hypothesis can be performed either bottom-up (i.e. from an overly specific clause to a more general) or top-down (i.e. from an overly general clause to a more specific). In this work we study the induction of definite programs consisting of multiple clauses, where each clause is searched for top-down. This problem can be formulated in the following way:

Given: a definite program O (overly general hypothesis), a definite program B (background predicates) and two finite sets of ground atoms E^+ and E^- (positive and negative examples).
Find[1]: a definite program H, called a *valid hypothesis*, such that $M(H \cup B) \subseteq M(O \cup B)$, $E^+ \subseteq M(H \cup B)$ and $M(H \cup B) \cap E^- = \emptyset$.

In this work we assume that all positive and negative examples are ground instances of the same atom, whose predicate symbol is referred to as the *target predicate*, and that all clauses in O, and only those, define the target predicate. Furthermore, we assume the clauses in O and B to be non-recursive w.r.t. the target predicate (i.e. no instance of the target predicate is allowed in the body of a clause). It should be noted that this assumption does not prevent recursive predicates from being used in the definition of the target predicate. The motivation for assuming the target predicate to be non-recursive in the overly general hypothesis is given in section 3.3.

Two specialisation operators that are commonly used when searching top-down for a single clause are literal addition (used in e.g. [25, 24, 22]) and resolution (used in e.g. [1, 12, 22]). By literal addition, a clause is specialised by adding a literal to the body, where the literal usually is restricted to be an instance of a background predicate. Various restrictions are normally also put on the variables in the literals (e.g. at least one of the variables should appear previously in the clause [24]). By resolution, a clause is specialised by resolving upon a literal in the body using one of the background clauses. In the resolution-based approaches, the clauses in the overly general hypothesis 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. It should be noted that for each clause obtained by resolution there is an equivalent[2] clause (w.r.t. the background predicates) that can be obtained by literal addition[3] (not necessarily in one step). On the other hand, it is also possible to define predicates that may introduce any literal, such that for any clause obtainable by literal addition there is an equivalent clause (w.r.t. the background predicates)

[1] $M(P)$ denotes the least Herbrand model of P.

[2] We say that two definite clauses C_1 and C_2 (or hypotheses H_1 and H_2) are equivalent w.r.t. a definite program B if and only if $M(\{C_1\} \cup B) = M(\{C_2\} \cup B)$ (or $M(H_1 \cup B) = M(H_2 \cup B)$).

[3] It is assumed that `=(X,X)` is among the background predicates.

obtainable by resolution.[4]

The two main strategies for top-down induction of logic programs are Covering and Divide-and-Conquer. Covering, which has been used in e.g. MIS [25], FOIL [24], ANA-EBL [12], FOCL [22], GRENDEL [14], FOCL-FRONTIER [21] and PROGOL [18], constructs a hypothesis by repeatedly specialising an overly general clause, on each iteration selecting a specialised clause that covers a subset of the positive examples and no negative examples, until all positive examples are covered by the selected clauses. Divide-and-Conquer, which has been used in e.g. ML-SMART [1], STRUCT [27], IDEL [13] and SPECTRE [7], constructs a hypothesis by dividing an overly general clause into a set of clauses, which correspond to disjoint subsets of the examples. It then continues recursively with those clauses for which the corresponding subsets contain both positive and negative examples. The resulting hypothesis consists of all specialised clauses for which the corresponding sets contain positive examples only.

In the next section, we formalise the two main strategies for top-down induction of logic programs using resolution as a specialisation operator, resulting in two strategies for performing example-guided unfolding. In section three, we analyse these strategies theoretically and compare them with respect to their computational complexity, the size of their hypothesis spaces and their ability to produce recursive hypotheses. In section four, we compare the two strategies empirically w.r.t. efficiency and size and accuracy of the produced hypotheses. Finally, we give some concluding remarks in section five. The reader is assumed to be familiar with the standard terminology in logic programming [16].

2 Covering and Divide-and-Conquer

In this section we formalise Covering and Divide-and-Conquer using resolution as a specialisation operator and illustrate how the techniques work using an example. We also show under what conditions the techniques produce valid hypotheses. Finally, we show that one of these conditions can always be fulfilled.

[4] The following technique (suggested by the first author of this paper) has been proven to be complete [20]: Let D be a definite clause with variables $X1, ..., Xn$. Let D' be the clause obtained by adding to D the goal `g([X1, ..., Xn])`, where $g/1$ is defined in the following way:

```
g(L).
g(L):- g([X|L]).
```

for every n-ary predicate symbol p:

```
g(L):- p(X1, ..., Xn), member(X1,L), ..., member(Xn,L), g(L).
```

for every n-ary function symbol f:

```
g(L):- member(f(X1, ..., Xn),L), member(X1,L), ..., member(Xn,L), g(L).
```

Now any clause that is subsumed by D can be obtained by starting with D' and applying resolution upon the goals $g/1$ and $member/2$ (assuming the standard definition of $member/2$).

2.1 Covering

The Covering principle can be applied using resolution as a specialisation operator in the following way. One of the clauses in the overly general hypothesis is selected and specialised by resolving upon a literal in the clause until the selected clause does not cover[5] any negative examples. This process is iterated until all positive examples are covered by the selected clauses. This technique is formalised in Figure 1.

FUNCTION COVERING(O, B, E^+, E^-)
  $H := \emptyset$
  WHILE $E^+ \neq \emptyset$ **DO**
    $C :=$ a clause in O that covers some $e \in E^+$ w.r.t. B
    WHILE C covers an element in E^- w.r.t. B **DO**
      $C :=$ a resolvent R of C and a clause in B, such that R covers some $e \in E^+$ w.r.t. B
    $H := H \cup \{C\}$
    $E^+ := E^+ \setminus \{e \in E^+ : e$ is covered by C w.r.t. $B\}$
  RETURN H

Figure 1: The Covering algorithm.

Example Assume that we are given the overly general hypothesis:

```
reward(S,R):- suit(S), rank(R).
```

and the background predicates in Figure 2, together with the following sets of positive and negative examples:

$E^+ =$ { `reward(spades,7)`, `reward(clubs,3)`}
$E^- =$ { `reward(hearts,5)`, `reward(clubs,jack)`}

Calling Covering with this input results in the following. Since the clause in the overly general hypothesis covers negative examples, it is specialised. Choosing the first literal to resolve upon using the second clause defining `suit(S)` results in the following clause:

```
reward(S,R):- black(S), rank(R).
```

This clause still covers the second negative example, and is thus specialised. Choosing the second literal to resolve upon using the first clause defining `rank(R)` results in the following clause:

[5] A clause $A \leftarrow B_1, \ldots, B_n$ is said to cover an atom $A\theta$ w.r.t. a definite program P if and only if there is an SLD-refutation of $P \cup \{(\leftarrow B_1, \ldots, B_n)\theta\}$.

```
suit(S):- red(S).
suit(S):- black(S).
rank(R):- num(R).
rank(R):- face(R).
red(hearts).
red(diamonds).
black(spades).
black(clubs).
num(1). ... num(10).
face(jack). face(queen). face(king).
```

Figure 2: Background predicates.

```
reward(S,R):- black(S), num(R).
```

This clause does not cover any negative examples and is thus added to the resulting hypothesis. Since the hypothesis now covers all positive examples, the algorithm terminates. ■

Covering produces a valid hypothesis in a finite number of steps when i) all positive examples are covered by the overly general hypothesis w.r.t. the background predicates, ii) there is a finite number of SLD-derivations of positive and negative examples (i.e. the program terminates for all examples) and iii) there are no positive and negative examples that have the same sequence of input clauses in their SLD-refutations. This is shown in Appendix A. It should be noted that this property is not dependent on how the non-deterministic choices in the algorithm are made. However, these choices are crucial for the result. Normally, a few number of clauses of high generality are preferred to a large number of specific clauses, and making the wrong choices may result in a non-preferred, although valid, hypothesis. Since it is computationally expensive to find the optimal choices, these are often approximated. In several approaches this has been done by selecting the refinement that maximises the information gain [24, 22, 14]:

$$cov(R, E^+)\left(\log_2 \frac{cov(R, E^+)}{cov(R, E^+ \cup E^-)} - \log_2 \frac{cov(C, E^+)}{cov(C, E^+ \cup E^-)}\right)$$

where R is the resolvent of a clause C and a clause in B and $cov(D, E)$ denotes the number of elements in a set of examples E that are covered by a clause D.

2.2 Divide-and-Conquer

The Divide-and-Conquer principle can be applied in a logic programming framework using resolution as a specialisation operator in the following way. Each clause in the overly general hypothesis covers a subset of the positive and negative examples. If a clause covers positive examples only, then it should be included in the resulting hypothesis, and if it covers negative examples only then it should be excluded. If a clause covers both negative and positive examples, then it corresponds to a part of the hypothesis that needs to be further divided into sub-hypotheses. When dividing a hypothesis into a set of sub-hypotheses, these should be equivalent to the divided hypothesis. This means that a clause that covers both positive and negative examples should be split into a number of clauses, that taken together should be equivalent to the clause that is split. This can be achieved by applying the transformation rule *unfolding*[6] [26]. This technique is formalised in Figure 3.

FUNCTION DAC(C, B, E^+, E^-)
  IF $E^+ = \emptyset$ **THEN** $H := \emptyset$ **ELSE**
  IF $E^- = \emptyset$ **THEN** $H := \{C\}$ **ELSE**
    Unfold upon a literal in C, giving $C_1, \ldots, C_n$
    Let E_i^+ and E_i^- be the set of examples in E^+ and E^- respectively
    that are covered by C_i w.r.t. B, $1 \leq i \leq n$
    $H := DAC(C_1, B, E_1^+, E_1^-) \cup \ldots \cup DAC(C_n, B, E_n^+, E_n^-)$
  RETURN H

Figure 3: The Divide-and-Conquer algorithm.

Example Consider again the overly general hypothesis, background predicates and examples in the previous example. Calling Divide-and-Conquer with this input results in the following.

Since the clause covers both positive and negative examples, unfolding is applied. Unfolding upon `suit(S)` replaces the clause with the following two clauses:

```
reward(S,R):- red(S), rank(R).
reward(S,R):- black(S), rank(R).
```

The first clause covers one negative example only, while the second clause covers two positive examples and one negative example. The algorithm is then called once with each of these clauses. The empty hypothesis is returned by the first call since the first clause does not cover any positive examples. The clause used in the second call is unfolded since it covers both positive and negative

[6] Unfolding upon a literal L in the body of a clause C in a definite program P, means that C is replaced with the resolvents of C and each clause in P whose head unifies with L.

examples. Unfolding upon `rank(R)` replaces the clause with the following two clauses:

```
reward(S,R):- black(S), num(R).
reward(S,R):- black(S), face(R).
```

The first of these clauses covers two positive and no negative examples and is therefore included in the resulting hypothesis, while the second covers one negative example only, and is therefore not included. Hence, the resulting hypothesis is:

```
reward(S,R):- black(S), num(R).
```

■

Divide-and-Conquer produces a valid hypothesis in a finite number of steps when i) all positive examples are covered by the overly general hypothesis w.r.t. the background predicates, ii) there is a finite number of SLD-derivations of positive and negative examples (i.e. the program terminates for all examples) and iii) there are no positive and negative examples that have the same sequence of input clauses in their SLD-refutations. This is shown in Appendix B.

As for Covering, it should be noted that the non-deterministic choices (in this case of which literals to unfold upon) are crucial for the result when applying Divide-and-Conquer. Again, the optimal choices can be approximated by selecting the specialisation that maximises the information gain, as is done in [27, 12, 7] (cf. ID3 [23]). This is equivalent to minimising:

$$-c\sum_{i=1}^{n} cov(C_i, E^+)\log_2\frac{cov(C_i, E^+)}{cov(C_i, E^+ \cup E^-)} + cov(C_i, E^-)\log_2\frac{cov(C_i, E^-)}{cov(C_i, E^+ \cup E^-)}$$

where $C_1, \ldots, C_n$ are the resolvents upon one of the literals in the current clause C, $cov(C_i, E)$ denotes the number of elements in E that are covered by C_i and the constant c is $1/cov(C, E^+ \cup E^-)$.

Note that this heuristic credits a high coverage of either positive or negative examples, while the information gain for Covering credits a high coverage of positive examples only.

It should also be noted that in case of multiple SLD-refutations of some examples, unfolding is not guaranteed to partition the examples, which means that the sum of the number of examples covered by each resolvent may be larger than the number of examples covered by the current clause. In such cases, it seems more appropriate to count the number of SLD-refutations of positive and negative examples, rather than just counting the number of covered examples, as the number of SLD-refutations does not change when unfolding is applied (shown in [15]).

2.3 Guaranteeing Unique Sequences of Input Clauses

One of the conditions for Covering and Divide-and-Conquer to produce valid hypotheses is that no positive and negative examples have the same sequence of input clauses in their SLD-refutations. In this section, we present a transformation technique, which guarantees that this condition holds.

With no loss of generality, we can assume that all terms in the Herbrand universe are defined by a predicate `term(X)`.[7] Then we can rewrite the original program by adding to each clause defining the target predicate, a literal `term(X)` for each variable `X` in the head of the clause. Then each example will have a unique branch in the SLD-tree. For example, assume that the original program consists of one clause: `p(X)`, and that the Herbrand universe is $\{0, s(0), s(s(0)), ...\}$. Then the program can be written as:

```
p(X):- term(X).
term(0).
term(s(X)):- term(X).
```

Whereas in an SLD-tree of the original program the refutations of the two examples `p(0)` and `p(s(0))` follow identical branches, they follow different branches in an SLD-tree of the transformed program.

3 Theoretical Analysis

In this section we analyse Covering and Divide-and-Conquer with respect to the computational complexity, the hypothesis spaces that are explored, the ability to produce recursive hypotheses and the amount of redundancy in the produced hypotheses.

3.1 Computational Complexity

Assuming the cost of checking whether a clause covers an example or not to be constant, we can give upper bounds on the computational complexity of Covering and Divide-and-Conquer, as described below.

Let l be the maximum length of the SLD-refutations of the positive and negative examples, m be the maximum number of clauses defining a predicate, p be the number of positive examples, and n the number of negative examples. By *derivation branch*, we mean the sequence of derived clauses[8] from an overly general clause to a resolvent that is kept in the produced hypothesis.

The maximum length of a derivation branch is l, and for the ith derived clause ($1 \leq i \leq l$), there are in the worst case $m(l-i)$ alternative ways of applying resolution upon the clause, since there are at most $l-i$ literals in the

[7] One clause is needed for each constant and function symbol.

[8] *Derived clause* is defined in Appendix C.

body of the ith derived clause. Thus, for each derivation branch the number of derived clauses that need to be evaluated is bounded by $ml^2/2$.

For Covering, there are in the worst case p different derivation branches that need to be considered, and for the ith considered branch ($1 \leq i \leq p$), all negative examples and $p - i + 1$ positive examples have to be checked for each evaluated derived clause. Thus, for Covering, the number of times a clause is checked w.r.t. an example is bounded by $(ml^2/2)p(n + (p+1)/2 + 1)$.

In the case when no example has more than one SLD-refutation, each example needs at most to be checked w.r.t. one derivation branch in Divide-and-Conquer, since the sets of examples covered by a set of clauses obtained by unfolding are mutually exclusive. Since there are $ml^2/2$ derived clauses that need to be evaluated for each derivation branch, the number of times a clause is checked w.r.t. an example is bounded by $(ml^2/2)(n+p)$. In the case when there are more than one SLD-refutation for some examples, the number of times a clause is checked w.r.t. an example is bounded by the same number as for Covering.

In summary, the computational cost for Covering grows at most quadratically in the number of (positive) examples, while it grows linearly in the number of examples for Divide-and-Conquer when each example has at most one SLD-refutation, and quadratically otherwise.

The above analysis is consistent with the worst-case complexity analysis of Divide-and-Conquer and Covering in a propositional framework presented in [11], where the computational cost of Divide-and-Conquer (represented by Assistant) was shown to grow linearly with the number of examples, while the cost of Covering (represented by CN2) was shown to grow quadratically. In section 4.1, we show that any propositional learning problem can be transformed into a top-down ILP problem for which each example has exactly one SLD-refutation, thus allowing Divide-and-Conquer to run in linear time.

3.2 The Hypothesis Spaces

Let O be an overly general hypothesis and B be background predicates. The hypothesis space for Covering is:

$$\mathcal{H}_{cov} = \{H : H \subseteq \{C_0 \times \ldots \times C_n : C_0 \in O \text{ and } C_1, \ldots, C_n \in B, \text{ where } n \geq 0\}\}^9$$

The hypothesis space for Divide-and-Conquer is:

$$\mathcal{H}_{dac} = \{H : H \subseteq H' \setminus B, \text{ where } H' \text{ is obtainable from } O \cup B \text{ by a number of applications of unfolding upon clauses that are not in } B\}.$$

Note that $\mathcal{H}_{dac} \subset \mathcal{H}_{cov}$, which follows from the fact that each set of clauses obtained by unfolding can be obtained by resolution and that there are pro-

[9] $C \times D$ denotes a resolvent of C and D upon a literal in the body of C. Note that $C \times D$ is in general not unique.

grams that can be produced by adding resolvents that cannot be produced by unfolding. For example, consider the overly general hypothesis:

```
p(X):- q(X), r(X).
```

and the background predicates:

```
q(X):- s(X).
r(X):- t(X).
```

Then the following hypothesis is in $\mathcal{H}_{cov}$, but not in $\mathcal{H}_{dac}$:

```
p(X):- s(X), r(X).
p(X):- q(X), t(X).
```

However, for each (non-recursive) hypothesis that can be produced by Covering, Divide-and-Conquer can produce a hypothesis that is equivalent w.r.t. the background predicates. This is shown in Appendix C.

3.3 Recursive Hypotheses

As was stated in section 1, the target predicate is assumed to be non-recursive. The reason for this is that Divide-and-Conquer does not work properly when specialising clauses that define recursive predicates. This because decisions regarding one part of the hypothesis may affect the coverage of other parts, and thus such decisions cannot be made independently as is done in Divide-and-Conquer. In [27], this problem is approached by assuming that the definition of the target predicate will be equivalent (w.r.t. the background predicates) to the set of positive examples (as is done in FOIL [24]) and hence the coverage of different parts of the hypothesis can be determined independently. However, in many cases this assumption leads to non-valid hypotheses being produced.

Another approach to this problem is to transform the overly general hypothesis into a non-recursive definition, as proposed in [7]: Let T be the recursive target predicate and O be the clauses defining T. Then introduce a new predicate T' by adding a clause $T' \leftarrow T$, where the arguments of T' are all variables in T (i.e. definition [26]). Unfold upon T in the clause, and replace each instance $T\theta$ in the bodies of the clauses defining T and T' (directly or indirectly) with $T'\theta$ (i.e. folding [26]). Then a non-recursive definition of T has been obtained, such that $T\theta \in M(O \cup B)$ if and only if $T\theta \in M(O' \cup B)$. For example, let the recursive target predicate be `nat(X)`, which is defined by the two clauses:

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

Then the non-recursive definition will be:

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

However, although this transformation allows Divide-and-Conquer to be applied, it prevents recursive hypotheses from being found. For example, assume the following positive and negative examples to be given:

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

Then the hypothesis produced by Divide-and-Conquer, after having applied the above transformation, will exclude the negative examples only (i.e. a maximally general specialization is obtained):

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

It should be noted that although Divide-and-Conquer cannot be used to produce recursive hypotheses, it does not mean that such cannot be found by applying unfolding and clause removal. On the contrary, a technique for achieving this is presented in [4].

Covering, on the other hand, can be extended to deal with recursive predicates (c.f. [6]). Instead of searching for a clause that together with background predicates covers some positive examples and no negative examples, a clause can be searched for that together with the clauses found so far and the background predicates covers some not yet covered positive examples without covering any negative examples, and that allows for the remaining positive examples to be covered without covering any negative examples.

3.4 Redundancy

When using Covering, the number of SLD-refutations of the positive examples is not necessarily the same for the resulting hypothesis as for the overly general hypothesis, i.e. the amount of redundancy may increase or decrease. On the other hand, when using Divide-and-Conquer, the number of SLD-refutations of the positive examples is the same for both the overly general and the resulting hypothesis. This follows from the fact that the number of SLD-refutations does not increase when unfolding is applied (proven in [15]). In order to allow for reduction of the amount of redundancy when using Divide-and-Conquer, only a minor change to the algorithm is needed: instead of placing a positive example

in all subsets that correspond to clauses that cover the example, the example can be placed in one such subset.

4 Empirical Evaluation

In this section we empirically evaluate the performance of Covering and Divide-and-Conquer. We first present four domains that are used in the experiments and then present the experimental setting and results.

4.1 Domains

Two domains are taken from the UCI repository of machine learning databases and domain theories: King+Rook versus King+Pawn on a7 and Tic-Tac-Toe. The third domain, which considers natural language parsing using a definite clause grammar, is taken from [8], while the fourth domain King-Rook-King-Illegal [19] is one of the most frequently used ILP benchmark domains.

The example sets in the UCI repository are represented by attribute-value vectors, and have to be transformed into atoms in order to be used together with the algorithms. The number of examples is 3196 in the first domain (of which 52.2% are positive) and 958 in the second domain (of which 65.3% are positive).

Since the algorithms also require overly general hypotheses as input, such are constructed for the two first domains in the following way (cf. [12]). A new target predicate is defined with as many arguments as the number of attributes, and for each attribute a new background predicate is defined to determine the possible values of the attribute. This technique is illustrated by the following overly general hypothesis and background predicate for determining *win for x* in the Tic-Tac-Toe domain:

```
win_for_x(S1,S2,S3,S4,S5,S6,S7,S8,S9):-
     square(S1), square(S2), square(S3),
     square(S4), square(S5), square(S6),
     square(S7), square(S8), square(S9).
square(x).
square(o).
square(b).
```

An alternative formulation of the Tic-Tac-Toe domain is used as well, where a new intermediate background predicate is introduced. In the alternative formulation, the definition of the predicate `square(S)` is changed into the following:

```
square(x).
square(S):- o_or_b(S).
o_or_b(o).
o_or_b(b).
```

The hypothesis is that the new intermediate predicate will reduce the number of clauses in the resulting definition, and hence increase the accuracy. The reason for this is that it does not matter in the correct definition of the target predicate whether a square has the value `o` or `b`.

The set of positive examples in the third domain consists of all sentences of up to seven words that can be generated by the grammar in [8, p 455], i.e. 565 sentences. The set of negative examples is generated by randomly selecting one word in each correct sentence and replacing it by a randomly selected word that leads to an incorrect sentence. Thus the number of negative examples is also 565. Two versions of an overly general hypothesis are used for this domain. The first version is shown below:

```
sentence([every|X],Y):- s(X,Y).
sentence([a|X],Y):- s(X,Y).
...
s(X,X).
s([every|X],Y):- s(X,Y).
s([a|X],Y):- s(X,Y).
...
```

where the definition of the background predicate `s(X,Y)` is the same as for `sentence(X,Y)`, but with `s` substituted for `sentence` and with one extra clause: `s(X,X)`. By referring to the background predicate `s(X,Y)` instead of `sentence(X,Y)`, the problem with recursive overly general hypotheses is avoided, as discussed in section 3.3.

The second version introduces intermediate predicates in the definitions of `sentence(X,Y)` and `s(X,Y)`, that group words into classes in the following way[10]:

```
sentence(X,Y):- determiner(X,Z), s(Z,Y).
sentence(X,Y):- noun(X,Z), s(Z,Y).
...
determiner([every|X],X).
determiner([a|X],X).
...
```

The hypothesis is, like for the Tic-Tac-Toe domain, that the intermediate predicates will improve the accuracy of the resulting hypotheses.

The number of examples in the King-Rook-King-Illegal domain is 1000, of which 65.8% are negative. The following overly general hypothesis was used in the experiments:

[10] The word classes are taken from [8].

```
illegal(A,B,C,D,E,F):-
    comp(A,B), comp(A,C), ..., comp(E,F),
    adj_or_not(A,B), adj_or_not(A,C), ..., adj_or_not(E,F).

comp(A,A).
comp(A,B):- A < B.
comp(A,B):- A > B.

adj_or_not(A,B):- adj(A,B).
adj_or_not(A,B):- \+ adj(A,B).

adj(0,0).
...
adj(7,7).
adj(0,1).
...
adj(6,7).
adj(7,6).
...
adj(1,0).
```

4.2 Experimental Setting

Covering and Divide-and-Conquer are compared in the four domains using the information gain heuristics that were mentioned in section 2. In addition, we include results from using two state-of-the-art ILP systems: Progol 4.2 [18] and FOIL 6.4 [24].[11]

An experiment is performed with each domain, in which the entire example set is randomly split into two halves, where one half is used for training and the other for testing. The number of examples in the training sets that are given as input to the algorithms are varied, representing 1%, 5%, 10%, 25% and 50% of the entire example set, where the last subset corresponds to the entire set of training examples and a greater subset always includes a smaller. The same training and test sets are used for all algorithms. Each experiment is iterated 50 times and the mean accuracy on the test examples is presented below, as well as the mean number of clauses in the produced hypotheses. In addition, the amount of work performed by Divide-and-Conquer and Covering is presented, measured as the number of times it is checked whether a clause

[11] The default parameter settings were used in both systems except for that in FOIL the variable depth (d) was set to 7 (necessary in the DCG domain) and negated literals were disallowed (n), and in Progol, which was used in the last three experiments only, the variable depth parameter (i) was set to 7, 1 and 1 respectively and the maximum clause length (c) was set to 7, 9 and 2.

covers an example or not.[12]

4.3 Experimental Results

4.3.1 King-Rook vs. King-Pawn

The use of Progol in the King-Rook versus King-Pawn domain was prevented by the large number of identical arguments in the examples (in several cases more than twenty), leading to a combinatorial explosion when investigating all possible ways in which an example can be subsumed.[13] In Figure 4, 5 and 6, the results from this domain for the three other systems are presented. It can be seen that Divide-and-Conquer produces more accurate, but less compact, hypotheses than Covering for all sizes of the training set. Furthermore, Covering checks more examples than Divide-and-Conquer for all sizes of the training sets. When the size of the training set is 50%, the number of checks made by Covering is about 3.2 times as many as the number of checks made by Divide-and-Conquer. The mean learning time for Divide-and-Conquer at that point is 444.1 s, for Covering 1613.8 s and for FOIL only 26.4 s.

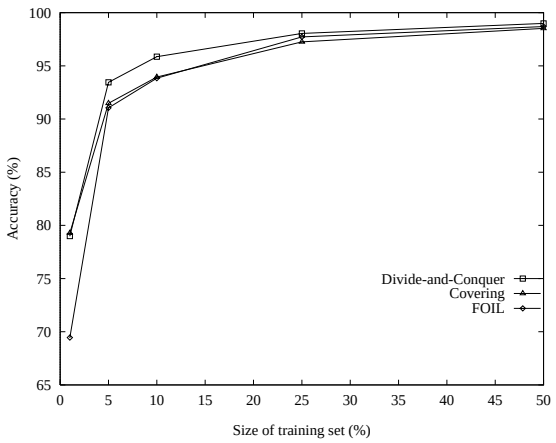

Figure 4: Accuracy for the KR vs. KP domain.

[12]The reason for using this measure of efficiency and not e.g. cpu seconds, is that this measure is implementation independent. Nevertheless, for some cases we also present the learning time for all four systems. Divide-and-Conquer and Covering were implemented in SICStus Prolog 3 #5 and all four systems executed on a SUN SparcStation 5.

[13]One way of avoiding this problem could be to unflatten each positive example by adding an equality to the body for each argument, but Progol 4.2 has problems detecting whether such an unflattened clause is covered or not.

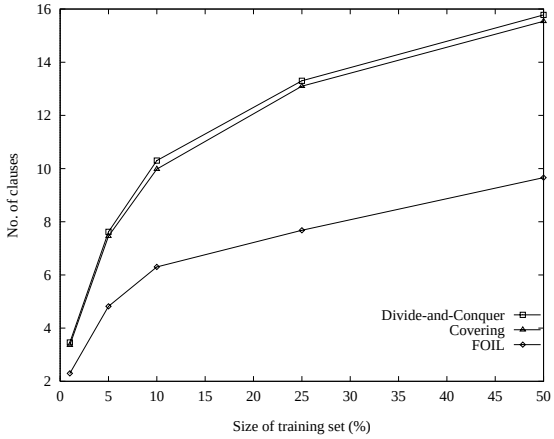

Figure 5: No. of clauses for the KR vs. KP domain.

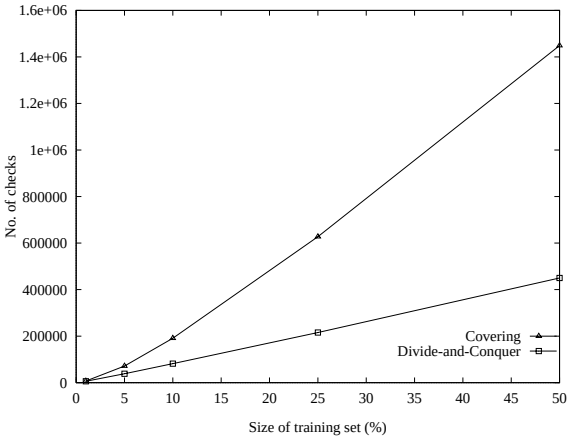

Figure 6: No. of checks for the KR vs. KP domain.

4.3.2 Definite Clause Grammar

FOIL requires that all background predicates are defined as ground unit clauses. In this domain, which considers sentences of up to seven words, and where there are 11 different words, a correct definition of the background predicate components/3, whose intensional definition is `components([X|L],X,L)`, would require more than 20 million facts, and the same number of facts would be needed to define the word classes, giving over 40 million facts in total. In order to make it possible to run FOIL in this domain, the above definitions were reduced to consider only lists of words that appear in the entire set of examples and suffixes of these lists (this resulted in a definition of components/3 with 2660 facts, and the same number of facts for the word classes). However, it should be noted that this solution is not possible in realistic situations as we will not know what sequences of words will appear in unseen examples.

In Figure 7 and 8 the predictive accuracy and size of the produced hypotheses is shown with and without word classes (the use of word classes is indicated by a w in the figures). When no word classes are used, Covering and Divide-and-Conquer produce identical hypotheses, which is shown by the dotted curves for these algorithms. The reason for this is that there is no choice of what literal to resolve upon (since there is only one body-literal in each overly general clause) and no example has more than one SLD-refutation. This experiment also shows that the amount of background knowledge can be far more important than what strategy is used.

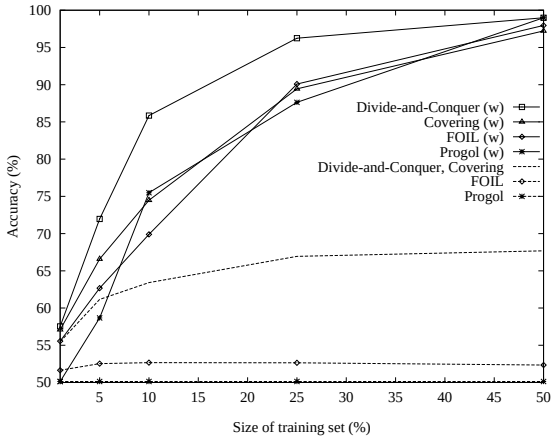

Figure 7: Accuracy for the DCG domain.

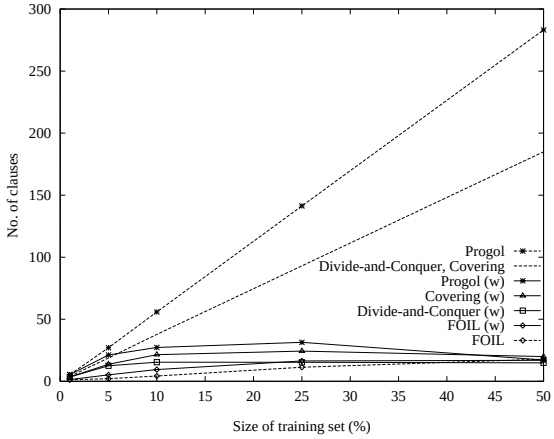

Figure 8: No. of clauses for the DCG domain.

In Figure 9, it can be seen that Covering checks more examples than Divide-and-Conquer for all sizes of the training sets and for both overly general hypotheses. When the size of the training set is 50%, the number of checks made by Covering without word classes is about 30 times as many as the number of checks made by Divide-and-Conquer. The mean learning time without word classes for Divide-and-Conquer at that point is 3.7 s, for Covering 86.8 s, for FOIL 19.7 s and for Progol 165.4 s. The mean learning time with word classes for Divide-and-Conquer at the same point is 14.5 s, for Covering 50.1 s, for FOIL 23.2 s and for Progol 1632.4 s.

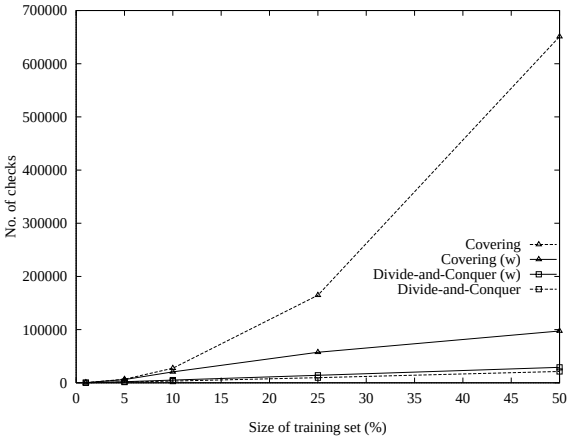

Figure 9: No. of checks for the DCG domain.

4.3.3 Tic-Tac-Toe

In Figure 10, 11 and 12, the results from the Tic-Tac-Toe domain are presented. The curves labeled Covering (i) and Divide-and-Conquer (i) represent the accuracy of the hypotheses produced by Divide-and-Conquer and Covering with the intermediate predicate, while the other curves are obtained whithout the intermediate predicate.[14] It can be seen that Covering produces more accurate and compact hypotheses than Divide-and-Conquer both with and without intermediate predicates.

The amount of work performed by Covering is more than what is performed by Divide-and-Conquer for all sizes of the training sets and for both overly general hypotheses, as shown in Figure 12. When the size of the training set is 50%, the mean learning time for Divide-and-Conquer with and without the intermediate predicate is 16.9 s and 9.2 s respectively, for Covering 42.1 s and 34.4 s, for FOIL (without the intermediate predicate) 1.9 s and for Progol 561.0 s.

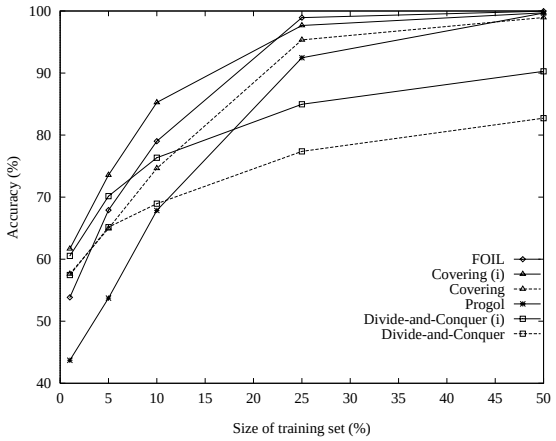

Figure 10: Accuracy for the Tic-Tac-Toe domain.

[14] The intermediate predicate was not given to FOIL and Progol as they could only perform worse by considering this predicate, since it will not help them in focusing on squares containing x (in contrast to the resolution-based approaches).

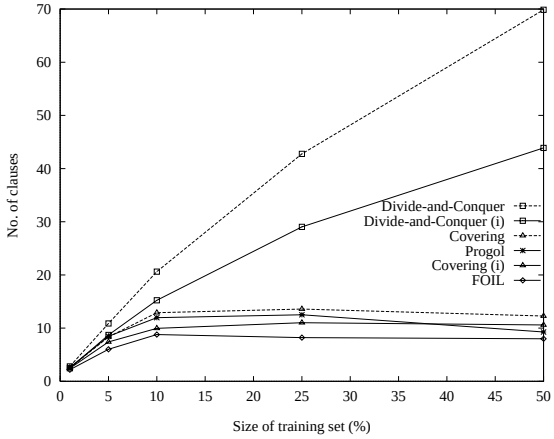

Figure 11: No. of clauses for the Tic-Tac-Toe domain.

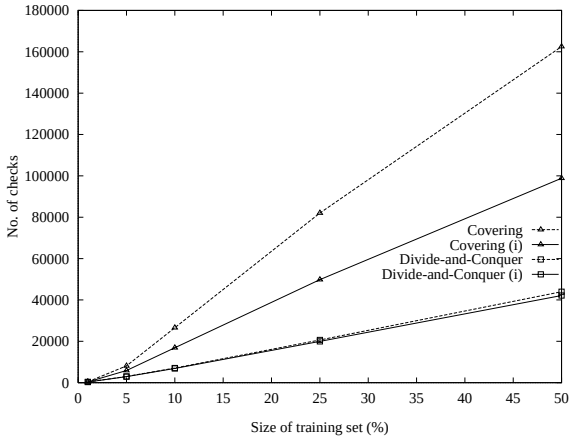

Figure 12: No. of checks for the Tic-Tac-Toe domain.

4.3.4 King-Rook-King-Illegal

In Figure 13, 14 and 15, the results from the King-Rook-King-Illegal domain are presented. It can be seen that w.r.t. accuracy and size, all three approaches based on covering (Progol, FOIL and the resolution-based approach Covering) outperform the divide-and-conquer approach.

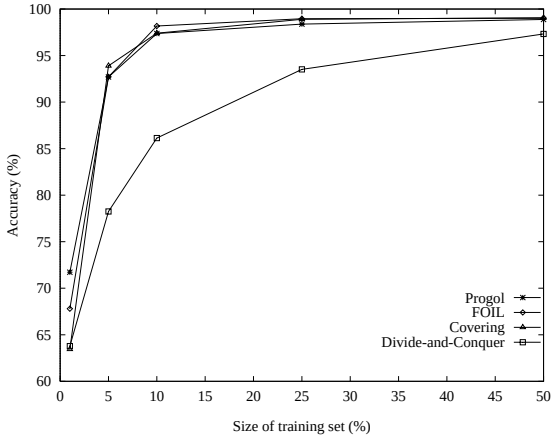

Figure 13: Accuracy for the King-Rook-King-Illegal domain.

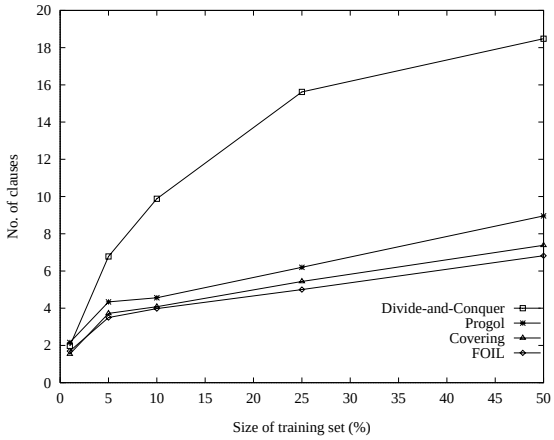

Figure 14: No. of clauses for the King-Rook-King-Illegal domain.

The amount of work performed by Covering is more than what is performed by Divide-and-Conquer for all sizes of the training sets, as shown in Figure 15. When the size of the training set is 50%, the number of checks made by Covering without intermediate predicates is about 1.85 times as many as the number of checks made by Divide-and-Conquer. The mean learning time for Divide-and-Conquer at that point is 234.8 s, for Covering 441.5 s, for FOIL 2.2 s and for Progol 242.3 s.

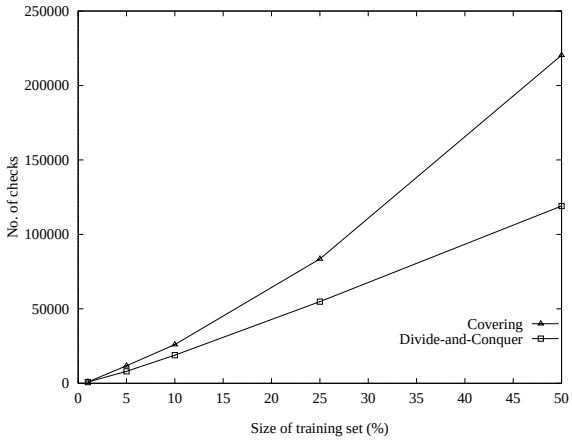

Figure 15: No. of checks for the King-Rook-King-Illegal domain.

4.3.5 Summary of experimental results

In summary, the hypotheses produced by Divide-and-Conquer were more accurate than the hypotheses produced by Covering in the two first domains, while they were less accurate in the two last domains. The results in the two first domains illustrate that it can be beneficial to focus on discriminating positive from negative examples, which is done by Divide-and-Conquer, rather than focusing on a high coverage of positive examples, which is done by Covering. The difference in accuracy in the two last domains can be explained by the fact that the number of clauses in the correct hypothesis within the hypothesis space for Covering is much less than the number of clauses in the correct hypothesis within the hypothesis space for Divide-and-Conquer (e.g. for the Tic-Tac-Toe domain these numbers are 8 and 126 respectively), and these numbers give lower-bounds for the number of positive examples needed for producing correct hypotheses.

In all domains, hypotheses were found with a smaller amount of work when using Divide-and-Conquer compared to when using Covering.

5 Concluding Remarks

We have formalised Covering and Divide-and-Conquer when applied to top-down induction of logic programs using resolution as a specialisation operator, resulting in two strategies for example-guided unfolding. It should be noted that in contrast to earlier approaches to example-guided unfolding (e.g. [9, 2, 10, 3]), the presented techniques only maintain partial correctness, as the purpose is to cover a set of positive examples and exclude a set of negative examples.

The main difference between the two strategies is that Covering applies

unfolding to the same overly general hypothesis repeatedly, while Divide-and-Conquer only uses the same hypothesis once. We have shown that the computational cost grows at most quadratically in the size of the example set for Covering, while it grows linearly for Divide-and-Conquer (when each example has at most one SLD-refutation). This was also demonstrated by the experiments, in which the amount of work performed by Covering was up to 30 times the amount of work performed by Divide-and-Conquer. The hypothesis space is larger for Covering, and thus more compact hypotheses may be found by this technique than by Divide-and-Conquer. However, we have shown that for each hypothesis that can be produced by Covering, there is an equivalent hypothesis (w.r.t. the background predicates) that can be produced by Divide-and-Conquer. A major draw-back of Divide-and-Conquer, in contrast to Covering, is that it is not applicable to learning recursive definitions.

The termination conditions for Covering and Divide-and-Conquer could be relaxed by slightly altering the algorithms. Instead of requiring that no positive and negative examples have the same sequence of input clauses in their SLD-refutations, it is enough to require that for each positive example there is one SLD-refutation with a unique sequence of input clauses. This alteration would lead to that some hypotheses can be found that are not found by the algorithms in their current formulations.

Instead of using resolution as a specialisation operator, literal addition could have been used in the formalisations and the experiments. In the Covering algorithm, a clause would then be specialised by adding a literal (as in [24]) rather than resolving upon a literal in the body. In the Divide-and-Conquer algorithm, there are two alternatives to replacing a clause by all resolvents upon a literal: either the clause is replaced by all clauses obtainable by adding a literal, or as in [27], by two clauses, where one is obtained by adding a new literal and the other is obtained by adding the negation of the literal (or a complementary literal). All results in the theoretical analysis would still be valid, since the former alternative corresponds to having a highly redundant overly general hypothesis, while the latter corresponds to having an overly general hypothesis for which each example has at most one SLD-refutation. In the light of the theoretical analysis, the second alternative seems to be superior. However, as was pointed out earlier, by using resolution instead of literal addition, explicit control of the possible specialisations is obtained, where the overly general hypothesis is used as a declarative bias that not only limits what predicate symbols are used, but also how they are invoked.

Acknowledgements

This work has been supported by the Swedish Research Council for Engineering Sciences (TFR) and the European Community ESPRIT IV LTR project no. 20237 (Inductive Logic Programming II).

Appendix A

In this section, it is shown that Covering produces a valid hypothesis in a finite number of steps when all positive examples are covered by the overly general hypothesis w.r.t. the background predicates, there is a finite number of SLD-derivations of positive and negative examples (i.e. the program terminates for all examples) and there are no positive and negative examples that have the same sequence of input clauses in their SLD-refutations.

Theorem 1 *Let E^+ and E^- be two finite sets of ground atoms and $P = O \cup B$ be a definite program (overly general program), such that the number of SLD-derivations of $P \cup \{\leftarrow e\}$ are finite for all $e \in E^+ \cup E^-$, and there is no $e^+ \in E^+$ and $e^- \in E^-$, such that the same sequence of input clauses is used both in an SLD-refutation of $P \cup \{\leftarrow e^+\}$ and in an SLD-refutation of $P \cup \{\leftarrow e^-\}$. Then after a finite number of steps, Covering outputs a definite program H, such that $M(H \cup B) \subseteq M(O \cup B)$, $E^+ \subseteq M(H \cup B)$ and $M(H \cup B) \cap E^- = \emptyset$.*

Proof Since the number of examples in E^+ is finite, it suffices to show that the inner while-loop in Covering finds a clause in a finite number of steps that covers at least one positive example and no negative examples. This can be shown by induction on the length l of the longest SLD-refutation of $\{C\} \cup B$ and $\{\leftarrow e^+\}$, for some covered $e^+ \in E^+$, where C is the clause selected before the inner while-loop is entered.

Base case: l=1. Then C is a clause such that $C \times \{\leftarrow e^+\} = \square$, for some $e^+ \in E^+$, and $C = C_1 \times \ldots \times C_n$, where each C_i is a variant of a clause in P $(1 \leq i \leq n)$. Then there is an SLD-refutation of $P \cup \{\leftarrow e^+\}$ with input clauses $C_1, \ldots, C_n$, since $C_1 \times \ldots \times C_n \times \{\leftarrow e^+\} = \square$. Assume that there is some $e^- \in E_i^-$, such that $C \times \{\leftarrow e^-\} = \square$. Then it follows that there is an SLD-refutation of $P \cup \{\leftarrow e^-\}$, with input clauses $C_1, \ldots, C_n$, since $C_1 \times \ldots \times C_n \times \{\leftarrow e^-\} = \square$. This contradicts the assumption that no $e^+ \in E^+$ and $e^- \in E^-$ have the same sequence of input clauses in their SLD-refutations. Thus $M(\{C\} \cup B) \cap E^- = \emptyset$.

Induction step: Assume the longest SLD-refutation of $\{C\} \cup B \cup \{\leftarrow e^+\}$ for some $e^+ \in E^+$ to be l+1. If $M(\{C\} \cup B) \cap E^- = \emptyset$ then the inner while-loop terminates and $H = H \cup \{C\}$. Otherwise, the inner while-loop is entered with a resolvent C' of C. Since the length of the longest SLD-refutation of $\{C'\} \cup B \cup \{\leftarrow e^+\}$, for some $e^+ \in E^+$, is l, the inner while-loop terminates, with a selected clause C such that $M(\{C\} \cup B) \cap E^+ \neq \emptyset$ and $M(\{C\} \cup B) \cap E^- = \emptyset$, according to the induction hypothesis.

Let $H = \{C_1, \ldots, C_n\}$. Since the target predicate is non-recursive, $M(H \cup B) = M(\{C_1\} \cup B) \cup \ldots \cup M(\{C_n\} \cup B)$. Hence, $E^+ \subseteq M(H \cup B)$ and

$M(H \cup B) \cap E^- = \emptyset$. Since the clauses in H are resolvents of clauses in $O \cup B$, it follows that $M(H \cup B) \subseteq M(O \cup B)$.

■

Appendix B

In this section, it is shown that Divide-and-Conquer produces a valid hypothesis in a finite number of steps when all positive examples are covered by the overly general hypothesis w.r.t. the background predicates, there is a finite number of SLD-derivations of positive and negative examples (i.e. the program terminates for all examples) and there is no positive and negative examples that have the same sequence of input clauses in their SLD-refutations.

Theorem 2 *Let E^+ and E^- be two finite sets of ground atoms and $P = O \cup B$ be a definite program (overly general program), such that the number of SLD-derivations of $P \cup \{\leftarrow e\}$ are finite for all $e \in E^+ \cup E^-$, and there is no $e^+ \in E^+$ and $e^- \in E^-$, such that the same sequence of input clauses is used both in an SLD-refutation of $P \cup \{\leftarrow e^+\}$ and in an SLD-refutation of $P \cup \{\leftarrow e^-\}$. Let $O = C_1, \ldots, C_n$ and E_i^+ and E_i^-, $1 \leq i \leq n$, be all examples in E^+ and E^- respectively that are covered by C_i. Then for each C_i, E_i^+ and E_i^-, $1 \leq i \leq n$, Divide-and-Conquer outputs after a finite number of steps a definite program H_i, such that $M(H_i \cup B) \subseteq M(O \cup B)$, $E_i^+ \subseteq M(H_i \cup B)$ and $M(H_i \cup B) \cap E_i^- = \emptyset$.*

Proof When $E_i^+ = \emptyset$, the theorem trivially holds. In the other case, the theorem can be proved by induction on the length l of the longest SLD-refutation of $\{C_i\} \cup B$ and $\{\leftarrow e^+\}$, for some $e^+ \in E_i^+$.

Base case: l=1. Then C_i is a clause such that $C_i \times \{\leftarrow e^+\} = \square$, for some $e^+ \in E^+$, and $C_i = D_1 \times \ldots \times D_m$, where each D_j is a variant of a clause in P $(1 \leq j \leq m)$. Then there is an SLD-refutation of $P \cup \{\leftarrow e^+\}$ with input clauses $D_1, \ldots, D_m$, since $D_1 \times \ldots \times D_m \times \{\leftarrow e^+\} = \square$. Assume that there is some $e^- \in E_i^-$, such that $C_i \times \{\leftarrow e^-\} = \square$. Then it follows that there is an SLD-refutation of $P \cup \{\leftarrow e^-\}$, with input clauses $D_1, \ldots, D_m$, since $D_1 \times \ldots \times D_m \times \{\leftarrow e^-\} = \square$. This contradicts the assumption that no $e^+ \in E^+$ and $e^- \in E^-$ have the same sequence of input clauses in their SLD-refutations. Thus $E_i^- = \emptyset$, and Divide-and-Conquer outputs $H_i = \{C_i\}$.

Induction step: Assume the longest SLD-refutation of $\{C_i\} \cup B$ and an example in E_i^+ to be $l + 1$. If $E_i^- = \emptyset$, Divide-and-Conquer terminates and outputs $H_i = \{C_i\}$. Otherwise, Divide-and-Conquer is called once for each resolvent D_j, $1 \leq j \leq m$, of C_i obtained by unfolding, with the sets of examples F_j^+ and F_j^-. Since the length of the longest SLD-refutation of $\{D_j\} \cup B \cup \{\leftarrow f^+\}$,

$1 \leq j \leq m$, where $f^+ \in F^+$, is l, the jth call to Divide-and-Conquer results in I_j, after a finite number of steps according to the induction hypothesis, where $F_j^+ \subseteq M(I_j \cup B)$ and $M(I_j \cup B) \cap F_j^- = \emptyset$. Then Divide-and-Conquer outputs $H_i = I_1 \cup \ldots \cup I_m$. Since the target predicate is non-recursive $M(\{H_i\} \cup B) = M(\{I_1\} \cup B) \cup \ldots \cup M(\{I_m\} \cup B)$. Hence $E_i^+ \subseteq M(H_i \cup B)$ and $M(H_i \cup B) \cap E_i^- = \emptyset$. Since the clauses in H_i are resolvents of clauses in $O \cup B$, it follows that $M(H_i \cup B) \subseteq M(O \cup B)$.

■

Appendix C

In this section, it is shown that for each non-recursive hypothesis that can be produced by Covering, there is an equivalent hypothesis (w.r.t. the background predicates) that can be produced by Divide-and-Conquer.

Let $C = (A_0 \leftarrow A_1, \ldots, A_m)$ and $D = (B_0 \leftarrow B_1, \ldots, B_p)$ be definite clauses. Then we write $C \times_n D$ to denote the resolvent $(A_0 \leftarrow A_1, \ldots, A_{n-1}, B_1, \ldots, B_p, A_{n+1}, A_m)\theta$ of C and D where θ is an mgu of $\{A_n, B_0\}$. The following lemma follows from the Switching Lemma in SLD-resolution [16, p 50].

Lemma 3 *Let $C = (L_0 \leftarrow L_1, \ldots, L_m, \ldots, L_n, \ldots, L_p)$, $D = (A_0 \leftarrow A_1, \ldots, A_q)$, and $E = (B_0 \leftarrow B_1, \ldots, B_r)$ be definite clauses. Then $(C \times_n D) \times_m E$ and $(C \times_m E) \times_{n+r-1} D$ are variants.*

Let C be a definite clause, and P a definite program. Then the *unfolding* of C w.r.t. P upon the nth body literal of C is the set of clauses $\{C \times_n D : D \in P\}$.

A proof of the following lemma can be found in [26, p 131].

Lemma 4 *Let C be a definite clause, P a definite program, and U an unfolding of C w.r.t. P. Then $M(P \cup \{C\}) = M(P \cup U)$.*

Before stating the theorem we need to introduce some terminology.

Let O be a set of definite clauses, and P a definite program. Then a *derived clause* w.r.t. O and P is recursively defined as follows:
a) if $C \in O$ then C is a derived clause w.r.t. O and P, and
b) if D is a derived clause w.r.t. O and P, and $E \in P$, then $D \times E$ is a derived clause w.r.t. O and P.

Let O be a set of clauses, and P a definite program. Then the *depth* of a literal in a derived clause w.r.t. O and P is recursively defined as follows:
a) if $C \in O$ then the depth of the literals in C is 0, and
b) if $D = (A_0 \leftarrow A_1, \ldots, A_n, \ldots, A_m)$ is a derived clause w.r.t. O and P, the depth of A_n is d, and $(A_0 \leftarrow A_1, \ldots, A_{n-1}, B_1, \ldots, B_p, A_{n+1}, \ldots, A_m)\theta$ is a resolvent of D and some clause in P, then the depth of the literals $B_1\theta, \ldots, B_p\theta$ is $d + 1$.

Let O be a set of definite clauses, and P a definite program. Then a derived clause $(L_0 \leftarrow L_1, \ldots, L_m)$ w.r.t. O and P is a *d-depth derived clause* w.r.t. O and P if and only if all the literals $L_1, \ldots, L_m$ are of depth d.

Let O be a set of clauses, P a definite program, R a set of derived clauses w.r.t. O and P, and the maximal depth of a literal in a clause in R is $\leq d$. Then a *d-depth unfolding set* of R w.r.t. P is a set of clauses obtained from R by repeatedly replacing each clause $C \in R$ with the unfolding of C w.r.t. P upon a body literal of C with a depth $< d$, until all clauses in R are d-depth derived clauses w.r.t. O and P.

Theorem 5 *Let O be a set of definite clauses, P a definite program, R a set of derived clauses w.r.t. O and P, and d the maximal depth of a literal in a clause in R. Then for every d-depth unfolding set U of O there exists an $S \subseteq U$ such that $M(P \cup S) = M(P \cup R)$.*

Proof By Lemma 3, the order of the applications of unfolding is insignificant. Thus, all d-depth unfolding sets of O w.r.t. P are equivalent (up to variable renaming). Let U_R be a d-depth unfolding set of R w.r.t. P. Then we have $U_R \subseteq U$. By Lemma 4, $M(P \cup O) = M(P \cup U)$ and $M(P \cup R) = M(P \cup U_R)$. Consequently, there exists a subset $S = U_R$ of U such that $M(P \cup S) = M(P \cup R)$. ■

References

[1] 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

[2] Boström H., "Eliminating Redundancy in Explanation-Based Learning", *Machine Learning: Proc. of the 9th International Conference*, Morgan Kaufmann, CA (1992) 37–42

[3] Boström H., "Improving Example-Guided Unfolding", *Proc. of the European Conference on Machine Learning*, Springer-Verlag (1993) 124–135

[4] Boström H., "Specialization of Recursive Predicates", *Proceedings of the Eighth European Conference on Machine Learning*, Springer-Verlag (1995) 92–106

[5] 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

[6] Boström H., "Theory-Guided Induction of Logic Programs by Inference of Regular Languages", *Proc. of the 13th International Conference on Machine Learning*, Morgan Kaufmann (1996) 46–53

[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] Bratko I., *Prolog Programming for Artificial Intelligence*, (2nd edition), Addison-Wesley (1990)

[9] Bruynooghe M., De Raedt L. and De Schreye D., "Explanation Based Program Transformation", *Proc. of the Eleventh International Joint Conference on Artificial Intelligence*, Morgan Kaufmann (1989) 407–412

[10] Clark P. and Holte R., "Lazy Partial Evaluation: An Integration of Explanation-Based Generalization and Partial Evaluation", *Machine Learning: Proc. of the 9th International Conference*, Morgan Kaufmann, CA (1992) 82–91

[11] Clark P. and Niblett T., "The CN2 Induction Algorithm", *Machine Learning* **3** (1989) 261–283

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

[13] Cohen W. W., "A Decision Tree Approach to Theory Specialization", unpublished manuscript (1991)

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

[15] Kanamori T. and Kawamura T., "Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation (II)", ICOT Technical Report TR-403, Japan (1988)

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

[17] Michalski R. S., "Pattern Recognition as Rule-Guided Inductive Inference", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **2** (1980) 349–361

[18] Muggleton S., "Inverse entailment and Progol", *New Generation Computing* **13** (1995) 245–286

[19] Muggleton S., Bain M., Hayes-Michie J. and Michie D., "An experimental comparison of human and machine learning formalisms". *Proceedings of the Sixth International Workshop on Machine Learning*, Morgan Kaufmann (1989) 113–118

[20] Nienhuys-Cheng S.-H. and de Wolf R., personal communication (1995)

[21] 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

[22] 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

[23] Quinlan J. R., "Induction of Decision Trees", *Machine Learning* **1**(1986) 81–106

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

[25] Shapiro E. Y., *Algorithmic Program Debugging*, MIT Press (1983)

[26] Tamaki H. and Sato T., "Unfold/Fold Transformations of Logic Programs", *Proceedings of the Second International Logic Programming Conference*, Uppsala University, Uppsala, Sweden (1984) 127–138

[27] Watanabe L. and Rendell L., "Learning Structural Decision Trees from Examples", *Proceedings of the Twelvth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann (1991) 770–776