Covering vs. Divide-and-Conquer for Top-Down Induction of Logic Programs

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

Abstract

Covering and divide-and-conquer are two well-established search techniques for top-down induction of propositional theories. However, for top-down induction of logic programs, only covering has been formalized and used extensively. In this work, the divide-and-conquer technique is formalized as well and compared to the covering technique in a logic programming framework. Covering works by repeatedly specializing an overly general hypothesis, on each iteration focusing on finding a clause with a high coverage of positive examples. Divide-and-conquer works by specializing an overly general hypothesis once, focusing on discriminating positive from negative examples. Experimental results are presented demonstrating that there are cases when more accurate hypotheses can be found by divide-and-conquer than by covering. Moreover, since covering considers the same alternatives repeatedly it tends to be less efficient than divide-and-conquer, which never considers the same alternative twice. On the other hand, covering searches a larger hypothesis space, which may result in that more compact hypotheses are found by this technique than by divide-and-conquer. Furthermore, divide-and-conquer is, in contrast to covering, not applicable to learning recursive definitions.

1 Introduction

The search for an inductive hypothesis can be performed either bottom-up (i.e. from an overly specific hypothesis to a more general) or top-down (i.e. from an overly general hypothesis to a more specific). In propositional learning, two search techniques for top-down induction have been proposed: covering and divide-and-conquer. Covering, which has been used in e.g. the AQ family [Michalski,1980], constructs a hypothesis by repeatedly specializing an overly general hypothesis, on each iteration selecting a disjunct that satisfies a subset of the positive examples and no negative examples, until all positive examples are satisfied by the selected disjuncts. Divide-and-conquer, which has been used in e.g. ID3 [Quinlan,1986], constructs a hypothesis by dividing an overly general hypothesis into a set of hypotheses, which correspond to disjoint subsets of the examples. It then continues recursively with those hypotheses for which the corresponding subsets contain both positive and negative examples. The resulting hypothesis consists of all specialized hypotheses for which the corresponding sets contain positive examples only.

One of the main long term goals of inductive logic programming is to upgrade the techniques of the propositional learning paradigm to a logic programming framework. This in order to allow for the use of a more expressive formalism and to allow for the use of substantial background knowledge in the learning process. However, of the two search techniques used in propositional learning, only covering has been formalized and used extensively for top-down induction of logic programs (e.g. in MIS [Shapiro,1983], FOIL [Quinlan,1990], ANA-EBL [Cohen,1991], FOCL [Pazzani *et al*,1991], GRENDEL [Cohen,1992] and FOCL-FRONTIER [Pazzani and Brunk,1993]). This work contributes to the above long term goal by giving a formalization of divide-and-conquer in a logic programming framework. This formalization is in fact a reformulation of the algorithm SPECTRE [Boström and Idestam-Almquist,1994], and can also be viewed as a generalization of the technique used in ML-SMART [Bergadano and Giordana,1988].

In the next section, the two search techniques are formalized and analysed in a logic programming framework. In section three, an empirical evaluation of the two techniques is presented, in which the techniques are compared with respect to efficiency and to the accuracy of the resulting hypotheses. Finally, concluding remarks are given in section four. In the following, we assume the reader to be familiar with the standard terminology in logic programming [Lloyd,1987].

2 Top-Down Induction of Logic Programs

In this section, we first define the top-down induction problem in a logic programming framework. We then present two common specialization operators and show how covering and divide-and-conquer can be formalized using one of these. Finally, we analyse the tech-

niques with respect to the hypothesis spaces that are explored, the ability to produce recursive hypotheses and the amount of redundancy in the produced hypotheses.

2.1 The Top-Down Induction Problem

The top-down induction problem can be formulated as follows:

Given: a definite program O (overly general hypothesis), a definite program B (background predicates) and two 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 T, referred to as the *target predicate*, and that all clauses in O, and only those, define T. Furthermore, we assume the target predicate to be non-recursive (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 other recursive predicates from being used in the definition of the target predicate. The reason for this assumption is discussed in section 2.6.

2.2 Specialization Operators

Literal addition is a specialization operator that has been used in several approaches for top-down induction of logic programs (e.g. [Shapiro,1983; Quinlan,1990; Pazzani *et al*,1991]). By this operator, a clause is specialized 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 elsewhere in the clause [Quinlan,1990]). Goal reduction is another specialization operator that has been used in several approaches (e.g. [Cohen,1991; Pazzani *et al*,1991]). By this operator, a clause is specialized by resolving upon a literal in the body using one of the background clauses. It should be noted that any specialization obtained by goal reduction can also be obtained by literal addition (not necessarily in one step). On the other hand, it is also possible to define predicates that may introduce any literal (cf. [Cohen,1992]), which makes any clause obtainable by literal addition also obtainable by goal reduction. Since goal reduction allows for more explicit control of what specializations are allowed than literal addition, we have chosen the former operator when studying search techniques for top-down induction.

2.3 Covering

The covering principle can be applied in a logic programming framework in the following way. One of the clauses in the overly general hypothesis is selected and specialized until the selected clause does not cover[2] any negative examples. This process is iterated until all positive examples are covered by the selected clauses. This technique is formalized in Figure 1, using goal reduction as a specialization operator.

INPUT: two definite programs O and B, and two sets of ground atoms E^+ and E^-.
OUTPUT: a valid hypothesis H.

Let $H = \emptyset$.
WHILE $E^+ \neq \emptyset$ **DO**
  Let $C \in O$ be a clause covering an element in E^+
  WHILE C covers an element in E^- **DO**
    Let C be a resolvent of C and a clause in B, such that it covers an element in E^+
  Let $H = H \cup \{C\}$.
  Let $E^+ = E^+ \setminus \{e : e \text{ is covered by } C\}$.

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,j)`}

```
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(j). face(q). face(k).
```

Figure 2: Background predicates.

Since the clause in the overly general hypothesis covers negative examples, it is specialized. 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 specialized. Choosing the second literal to resolve upon using the first clause defining `rank(R)` results in the following clause:

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

[1] M_P denotes the least Herbrand model of P.

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

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. □

The algorithm produces a valid hypothesis in a finite number of steps under the assumptions that there are a finite number of SLD-derivations of the positive and negative examples w.r.t. the overly general hypothesis and background predicates and that no positive and negative example have the same sequence of input clauses in their SLD-refutations. It should be noted that this property is not dependent on how the nondeterministic 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 maximizes the information gain [Quinlan,1990; Pazzani *et al*,1991; Cohen,1992].

2.4 Divide-and-Conquer

The divide-and-conquer principle can be applied in a logic programming framework 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 taken together, these hypotheses 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 are equivalent to the clause that is split. This can be achieved by applying the transformation rule *unfolding*[3] [Tamaki and Sato,1984]. This technique is formalized in Figure 3.

Example

Consider again the overly general hypothesis, background predicates and examples in the previous example. Calling the divide-and-conquer algorithm with the clause in the overly general hypothesis together with the background predicates and all covered examples 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 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).
```

□

INPUT: a clause C, a definite program B and two sets of ground atoms E^+ and E^-.
OUTPUT: a valid hypothesis H.

IF $E^+ = \emptyset$ **THEN** let $H = \emptyset$.
IF $E^+ \neq \emptyset$ and $E^- = \emptyset$ **THEN** let $H = \{C\}$.
IF $E^+ \neq \emptyset$ and $E^- \neq \emptyset$ **THEN**
  Unfold upon a literal in C, giving $C_1, \ldots, C_n$.
  Let E_i^+ and E_i^-, $1 \leq i \leq n$, be all examples in E^+ and E^- respectively that are covered by C_i.
  FOR $i = 1$ **TO** n **DO**
    Call *Divide-and-Conquer* with C_i, B, E_i^+ and E_i^-, giving H_i.
  Let $H = H_1 \cup \ldots \cup H_n$.

Figure 3: The Divide-and-Conquer algorithm.

Divide-and-conquer produces a valid hypothesis in a finite number of steps under the same assumptions as for covering, i.e. that there are a finite number of SLD-derivations of the positive and negative examples and that no positive and negative example have the same sequence of input clauses in their SLD-refutations. 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. Again, the optimal choices can be approximated by selecting the specialization that maximizes the information gain, as is done in [Boström and Idestam-Almquist,1994] (cf. ID3 [Quinlan,1986]).

2.5 The Hypothesis Spaces

Let O be an overly general hypothesis and B be background predicates. The hypothesis space for covering is $H_{cov} = \{H : H \subseteq \{C_0 \times \ldots \times C_n : C_0 \in O$ and $C_1, \ldots, C_n \in B$, where $n \geq 0\}\}$.[4] The hypothesis space for divide-and-conquer is $H_{dac} = \{H : H \subseteq H' \setminus B$, where H' is obtained from $O \cup B$ by applying unfolding upon clauses that are not in $B\}$.

Note that $H_{dac} \subset H_{cov}$, which follows from that each set of clauses obtained by unfolding can be obtained by

[3] When unfolding upon a literal L in the body of a clause C in a definite program P, C is replaced with the resolvents of C and each clause in P whose head unifies with L.

[4] $C \times D$ denotes a resolvent of C and D upon a literal in the body of C.

goal reduction and that there are programs that can be produced by covering that can not be produced by divide-and-conquer. 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 H_{cov}, but not in H_{dac}:

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

2.6 Recursive Hypotheses

As was stated in section 2.1, the target predicate is assumed to be non-recursive. The reason for this is that divide-and-conquer is not applicable when specializing clauses that define recursive predicates, since decisions regarding one subset of examples may then affect other subsets of examples. For example, consider the overly general hypothesis:

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

together with the positive and negative examples:

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

Then the first clause covers negative examples only, and should therefore not be included in the resulting hypothesis. However, this can not be achieved without obtaining an overly specific hypothesis, since then the positive examples would no longer be covered. Extending the definition of cover to include all examples that use a clause, and not only use it as the first clause in their refutations, does not solve the problem. Then in the example above, all examples would be covered by the first clause, and all except one by the second clause. However, since the first clause can not be removed or specialized, it means this sub-part of the problem can not be treated separately according to the divide-and-conquer principle. One solution to this problem is to transform the overly general hypothesis into an equivalent non-recursive hypothesis, as proposed in [Boström and Idestam-Almquist,1994]. However, although this transformation allows divide-and-conquer to be applied, it prevents recursive hypotheses from being found. It should be noted that although divide-and-conquer is not applicable to recursive predicates, it does not mean that recursive definitions can not be found by applying unfolding and clause removal. On the contrary, a technique for achieving this is presented in [Boström, 1995].

Covering, on the other hand, can be easily extended to deal with recursive predicates. 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 can cover 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.

2.7 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 [Kanamori and Kawamura,1988]). In order to reduce 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.

3 Empirical Evaluation

In this section we empirically evaluate the performance of covering and divide-and-conquer. We first present three domains that are used in the experiments and then present the experimental results.

3.1 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 considers natural language parsing using a definite clause grammar and is taken from [Bratko,1990].

The example sets in the UCI repository are represented by attribute 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. [Cohen,1991]). 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 [Bratko,1990, 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).
...
```

where the definition of the 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 2.6.

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:

```
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.

3.2 Experimental Results

The performance of covering and divide-and-conquer in the three domains is evaluated using the information gain heuristics that were mentioned in section 2. 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 both algorithms. Each experiment is iterated 50 times and the mean accuracy on the test examples is presented below. In addition, the amount of work performed by the two algorithms is presented, measured as the number of times it is checked whether a clause covers an example or not[5].

In Figure 4 and Figure 5, the results from the King-Rook versus King-Pawn domain are presented. It can be seen that divide-and-conquer produces more accurate 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 3.3 times as many as the number of checks made by divide-and-conquer. The mean cpu time for divide-and-conquer at that point is 566.9 s, and for covering 2075.3 s.

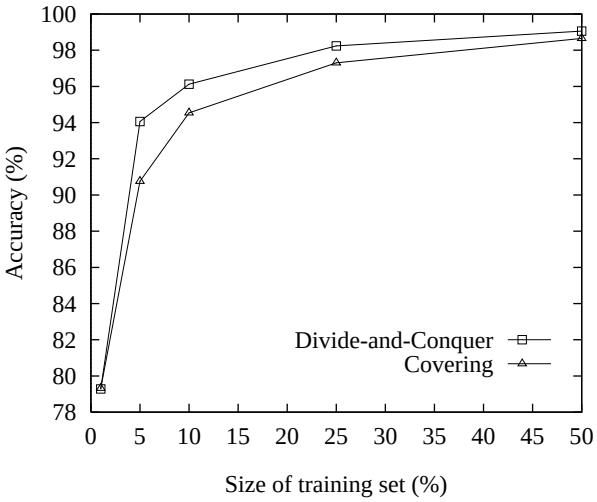

Figure 4: Accuracy for the KR vs. KP domain.

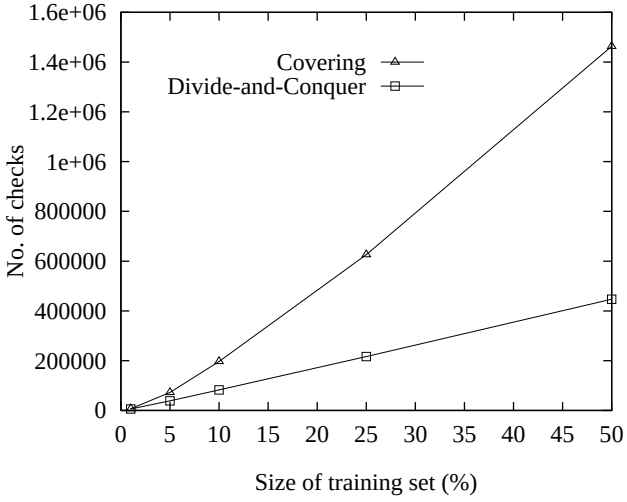

Figure 5: No. of checks for the KR vs. KP domain.

[5]The reason for using this measure of efficiency and not e.g. cpu seconds, is that this measure is not implementation dependent. Nevertheless, for some cases we also present the learning time measured in cpu seconds. The algorithms were implemented in SICStus Prolog 2.1 and executed on a SUN SparcStation 5.

In Figure 6 and Figure 7, the results from the DCG domain are presented. The two upper curves in Figure 6 denote the accuracy of the hypotheses produced by divide-and-conquer and covering using intermediate predicates. Interestingly, when no intermediate predicates are used, covering and divide-and-conquer produce identical hypotheses, which is shown by the lowest curve. This experiment also illustrates that the amount of background knowledge can be far more important than what search technique is used.

In Figure 7, 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 intermediate predicates is about 30.3 times as many as the number of checks made by divide-and-conquer. The mean cpu time for divide-and-conquer at that point is 5.4 s, and for covering 127.3 s.

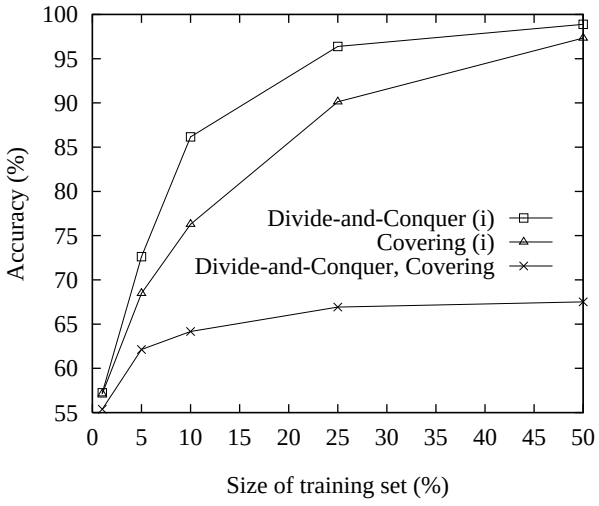

Figure 6: Accuracy for the DCG domain.

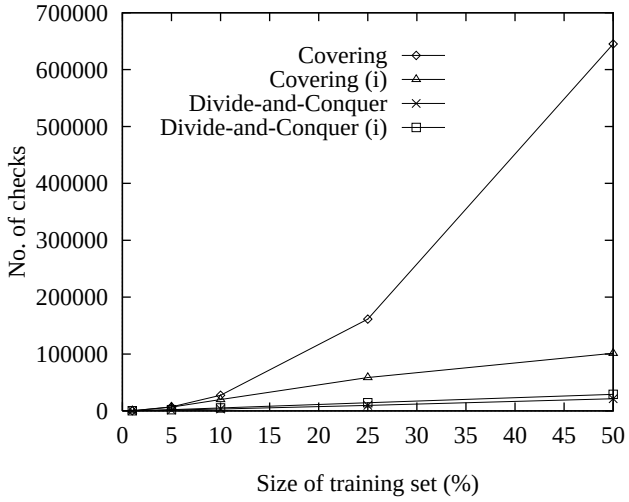

Figure 7: No. of checks for the DCG domain.

In Figure 8 and Figure 9, the results from the Tic-Tac-Toe domain are presented. The curves labeled *Covering(i)* and *Divide-and-Conquer (i)* in Figure 8 represent the accuracy of the hypotheses produced by divide-and-conquer and covering with intermediate predicates. The two other curves represent the accuracy of the programs produced by divide-and-conquer and covering without intermediate predicates. It can be seen that covering performs better 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 9. When the size of the training set is 50%, the number of checks made by covering without intermediate predicates is about 3.7 times as many as the number of checks made by divide-and-conquer. The mean cpu time for divide-and-conquer at that point is 14.0 s, and for covering 49.6 s.

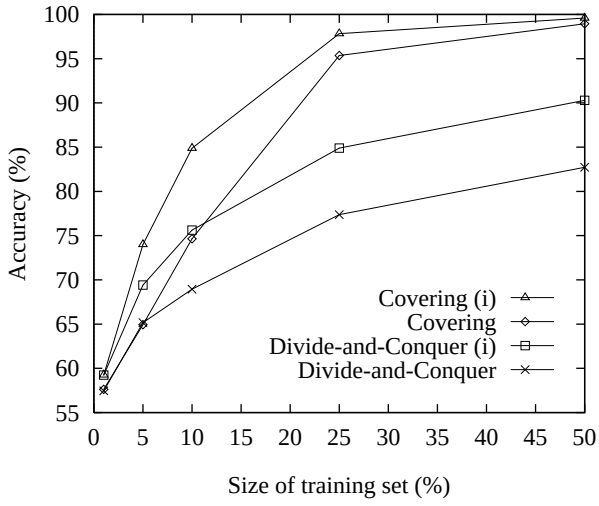

Figure 8: Accuracy for the Tic-Tac-Toe domain.

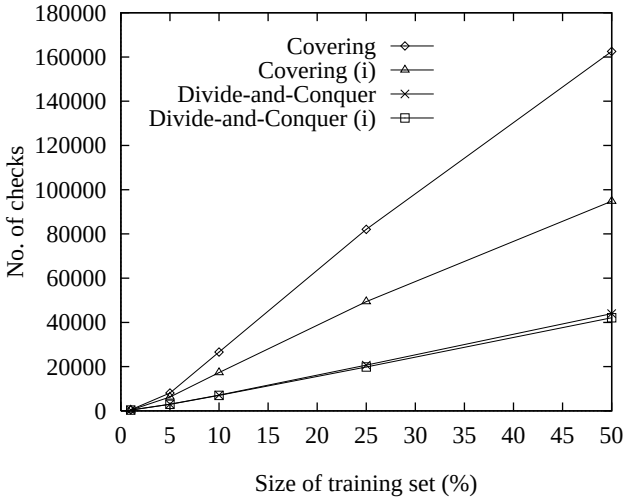

Figure 9: No. of checks for the Tic-Tac-Toe domain.

4 Concluding Remarks

We have formalized the covering and divide-and-conquer techniques for top-down induction of logic programs. The main difference between the techniques is that covering specializes an overly general hypothesis repeatedly, while divide-and-conquer only specializes it once. This has the consequence that a large amount of the work performed at each iteration in covering might be repeated, while this is not the case for divide-and-conquer. This was 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 experiments also demonstrated that the accuracy of the resulting hypothesis can be higher when focusing on discriminating positive from negative examples, which was done by divide-and-conquer, rather than focusing on a high coverage of positive examples, which was done by covering. On the other hand, the hypothesis space is larger for covering, and thus more compact hypotheses may be found by this technique than by divide-and-conquer. Moreover, 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 alterations would lead to that some valid hypotheses can be found that are not found by the algorithms in their current formulations.

Instead of using goal reduction as a specialization operator, literal addition could have been used in the formalizations and the experiments. In the covering algorithm, a clause would then be specialized by adding a literal rather than resolving upon a literal in the body. In the divide-and-conquer algorithm, a clause would then be replaced by all clauses obtainable by adding a literal, rather than all resolvents upon a literal. However, as was pointed out earlier, by using goal reduction instead of literal addition, explicit control of the possible specializations 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 European Community ESPRIT BRA 6020 ILP (Inductive Logic Programming). This work benefitted a lot from discussions with Peter Idestam-Almquist, who originally made the observation that the algorithm SPECTRE resembles ID3 and suggested the use of an impurity measure.

References

[Bergadano and Giordana,1988] 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

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

[Boström and Idestam-Almquist,1994] 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

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

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

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

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

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

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

[Pazzani and Brunk,1993] 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

[Pazzani *et al*,1991] 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

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

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

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

[Tamaki and Sato,1984] 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