

A Theory of Clausal Discovery*

Luc De Raedt and Maurice Bruynooghe

Department of Computing Science, Katholieke Universiteit Leuven

Celestijnenlaan 200A, B-3001 Heverlee, Belgium

email : Luc.DeRaedt@cs.kuleuven.ac.be

Abstract

A framework and algorithm for inducing full clausal theories from databases is presented. The algorithm is implemented in the system CLAUDIEN. The system combines clausal refinement operators, depth-first iterative deepening and language bias, to find clausal theories that satisfy a given database and that are expressible in the language. The clausal discovery algorithm computes a minimally complete theory expressible in the language. Such theories are most informative and logically non-redundant.

1 Introduction

A framework for discovering general first order clausal theories (sets of clauses of the form $p_1 \wedge \dots \wedge p_k \rightarrow q_1 \vee \dots \vee q_m$, where the p_i and q_j are first order atoms) from databases is presented. It is incorporated in the system CLAUDIEN (CLAUsal Discovery ENgine). CLAUDIEN combines ideas from the area of knowledge discovery (KDD), inductive logic programming (ILP) and deductive databases. From knowledge discovery in databases [Piatetsky-Shapiro and Frawley, 1991], CLAUDIEN inherits its aim: finding regularities from (potentially large) databases. However, in contrast to most of the current work in knowledge discovery, CLAUDIEN employs a first order logic framework. Current work in knowledge discovery has mainly used propositional and/or attribute-value representations, or has focussed on learning specific forms of first order sentences, such as for instance: determinations [Shen, 1992], and functional and inclusion dependencies [Kantola *et al.*, 1992, Flach, 1990]. CLAUDIEN shares its motivation for employing a first order logic framework with inductive logic programming [Muggleton, 1992, De Raedt and Lavrač, 1993]: logic facilitates the use of background knowledge and is much more expressive than propositional or attribute-value representations. Furthermore, there is a strong relation between logic programming and deductive databases [Minker, 1987], which provides

a solid theoretical basis for clausal discovery. Since clausal theories are often used as integrity theories in deductive databases, CLAUDIEN can be applied to automatically derive integrity theories/constraints from databases. Clausal theories are easy to understand. Therefore CLAUDIEN also contributes to the inductive engineering and knowledge synthesis paradigms proposed by Donald Michie and Ivan Bratko [Bratko, 1992].

CLAUDIEN employs an efficient iterative deepening search strategy, where clauses are expanded using a clausal refinement operator, and where pruning is achieved by checking whether a candidate clause is implied by the current clausal theory or not, and whether a clause contains redundant information or not. CLAUDIEN derives a minimally complete theory satisfying a given database. Roughly speaking, this means that the theory derived is most informative (i.e. logically maximally general) and logically non-redundant (i.e. deleting clauses from the theory results in losing information).

In Section 2, we present the theoretical framework of CLAUDIEN; in Section 3, the basic algorithm of CLAUDIEN is specified and illustrated; in Section 4, we report on some experiments with CLAUDIEN; and in Section 5, we conclude, review related work and indicate some directions for future work.

2 Problem specification

We briefly review some notions of logic programming (see [Lloyd, 1987] for more details):

Definition 1 A database D is a set of normal program clauses.

Definition 2 A normal program clause is a logical expression of the form $p \leftarrow l_1 \wedge \dots \wedge l_n$, where p is a positive atom and the l_i are positive or negative atoms.

Definition 3 A clause c is a logical expression of the form $p_1 \wedge \dots \wedge p_k \rightarrow q_1 \vee \dots \vee q_m$ where the p_i and q_j are positive atoms. $\text{body}(c)$ denotes $p_1 \wedge \dots \wedge p_k$ and $\text{head}(c)$ denotes $q_1 \vee \dots \vee q_m$.

Sometimes, we will use the equivalent disjunctive notation of clauses, i.e. $\neg p_1 \vee \dots \vee \neg p_k \vee q_1 \vee \dots \vee q_m$. As in most work on deductive database we assume that clauses are range-restricted:

Definition 4 A clause c is range-restricted if and only if all variables occurring in $\text{head}(c)$ also occur in $\text{body}(c)$.

*The authors are supported by the Belgian National Fund for Scientific Research and the ESPRIT III contract no. 6020 on Inductive Logic Programming.

There are many different semantics for normal programs. While they all agree for certain classes of programs, there can be subtle differences for programs involving recursion. Here, we adopt the completion semantics and the SLDNF proof procedure. Nevertheless, for function free databases one could argue in favor of a semantics selecting a particular model, such as the well-founded semantics for general logic programs [Van Gelder *et al.*, 1991], and a sound and complete proof procedure such as [Chen and Warren, 1992]. In what follows, completion semantics and SLDNF are replaceable by such a semantics and its corresponding proof procedure.

Definition 5 A clause c satisfies a database D if and only if $\text{Comp}(D) \models c$. A clause c violates a database D if and only if $\text{Comp}(D) \models \neg c$.

$\text{Comp}(D)$ denotes the completion of the database as proposed by Clark¹. A clause that neither satisfies nor violates the database is consistent, i.e. it is true in certain models and false in other models of the completion. The following result follows from the soundness of SLDNF resolution.

Lemma 1 If the query $\leftarrow \text{body}(c) \wedge \neg \text{head}(c)$ succeeds under SLDNF with substitution θ (i.e. $\text{Comp}(D) \vdash_{\text{SLDNF}} \text{body}(c) \wedge \neg \text{head}(c)\theta$) then c violates D ; if the query finitely fails under SLDNF then c satisfies D .

The result provides us with an incomplete method to verify whether a clause c satisfies a database or not. To check whether a clause c violates or satisfies a database we apply SLDNF-resolution ($\vdash_{\text{SLDNF}}$, cf. [Lloyd, 1987]) (i.e. the PROLOG execution mechanism) on the query. If the query succeeds, the clause violates the database. If the query finitely fails, the clause satisfies the database. Otherwise, the result is undefined: the query does not terminate.

Notice also that our semantics for ILP differs from the usual $B \wedge H \models E^+$ and $B \wedge H \models E^-$ semantics in ILP [Muggleton, 1992]; see [De Raedt and Lavrač, 1993, Helft, 1989] for a detailed discussion. The differences are related to the closed world assumption. In the usual ILP setting, the intended model is only partly specified by means of an incomplete set of positive and negative examples. In our framework, only the true facts (positive examples) are specified, and all facts not specified are considered false (by the completion and the closed world assumption). Therefore the intended model is much more constrained in our approach than usually. As a consequence fewer clauses will be acceptable in our semantics and there are no problems with dependencies when learning multiple predicates [De Raedt and Lavrač, 1993]. To illustrate this, assume $D = \{\text{flies}(\text{tweety}), \text{bird}(\text{tweety}), \text{bird}(\text{oliver})\}$. In the classical setting, $\text{bird}(B) \leftarrow \text{flies}(B)$ is acceptable as $\text{flies}(\text{oliver})$ is unknown, whereas in our approach this clause violates D as $\text{flies}(\text{oliver})$ is assumed to be false. Only if a complete example set is specified will clauses induced in the classical ILP setting also be solutions in our framework.

¹Throughout the paper we will assume that $\text{Comp}(D)$ is always logically consistent.

As in inductive machine learning, CLAUDIEN employs a declarative user-specified bias: the user has to specify the desired form that clauses discovered by CLAUDIEN should take. To make abstraction of the particular bias employed, we will employ the notion of a language $\mathcal{L}$ of clauses. For ease of presentation, we will consider $\mathcal{L}$ as a set of clauses. A partial order on clauses is induced on $\mathcal{L}$ by Plotkin's notion of θ -subsumption:

Definition 6 Clause c θ -subsumes clause c' if and only if there is a substitution θ such that $c\theta \subset c'$ and there is no substitution σ such that $c'\sigma \subset c$.

The problem addressed by CLAUDIEN, is that of discovering a minimally complete theory that satisfies a given database and language $\mathcal{L}$.

Definition 7 A theory is a set of range-restricted clauses.

Definition 8 A theory $T \subset \mathcal{L}$ is complete for a language $\mathcal{L}$ and a database D if and only if $\text{Comp}(D) \models T$ and $\forall T' \subset \mathcal{L} : (\text{Comp}(D) \models T') \rightarrow T \models T'$.

A theory is thus complete for a language if it is a maximally general subset of the language that satisfies the database (we employ the usual notion of logical generalisation here).

Lemma 2 For a given database D and language $\mathcal{L}$, all complete theories are logically equivalent.

Proofs of lemma's and theorems are contained in [De Raedt and Bruynooghe, 1993]. Complete theories capture all knowledge in the database that is expressible within the given language. Since all complete theories are logically equivalent, the most interesting ones are the minimal ones. Minimality requires that no clause in T is logically redundant.

Definition 9 A theory T is minimal with respect to a database D if and only if $\neg \exists c \in T : T - \{c\} \models c$.

Example 1: CLAUDIEN's problem specification:

$\mathcal{L} = \{c \mid \text{body}(c) \subset B, \text{head}(c) \subset B\}$
 where $B = \{\text{female}(X), \text{male}(X), \text{human}(X)\}$
 $D = \{\text{male}(\text{luc}) \leftarrow; \text{female}(\text{lieve}) \leftarrow; \text{human}(\text{lieve}) \leftarrow; \text{human}(\text{luc}) \leftarrow\}$
 $T = \{\text{female}(X) \wedge \text{male}(X) \rightarrow; \text{male}(X) \rightarrow \text{human}(X); \text{female}(X) \rightarrow \text{human}(X); \text{human}(X) \rightarrow \text{female}(X) \vee \text{male}(X)\}$

Theory T is minimally complete with respect to database D and language $\mathcal{L}$.

3 A clausal discovery algorithm

3.1 Refinement

The key to the clausal discovery algorithm is Lemma 1. It states that when a clause c violates a database D we must have that $\text{Comp}(D) \models \text{body}(c)\theta$ and $\text{Comp}(D) \models \neg \text{head}(c)\theta$ for a certain substitution θ . Since we assume the database to be correct, there are two ways of modifying the clause to make it satisfy the database²:

²The situation here is essentially dual to that considered by [De Raedt and Bruynooghe, 1992], where clauses are assumed correct and the database is modified.

- condition $body(c)$ can be strengthened so that $body(c)$ is true for less substitutions θ ;
- conclusion $head(c)$ can be weakened so that $head(c)$ is false for less substitutions θ .

Both operations can be achieved using a refinement operator based on Plotkin's θ -subsumption.

Definition 10 A refinement operator ρ (with transitive closure ρ^*) for a language $\mathcal{L}$ is a mapping from $\mathcal{L}$ to $2^{\mathcal{L}}$ such that

1. $\forall c \in \mathcal{L} : \rho(c) \subset \{c' \in \mathcal{L} \mid c' \text{ is a maximal specialisation of } c \text{ under } \theta\text{-subsumption}\}$, and
2. ρ is complete, i.e. $\rho^*(\top) = \mathcal{L}$ where $\top$ is the most general element in $\mathcal{L}$.

Completeness means that all elements of the language can be generated using ρ . In our framework, optimal refinement operators are the most desirable ones:

Definition 11 A refinement operator ρ (with transitive closure ρ^*) is optimal if and only if $\forall c, c_1, c_2 \in \mathcal{L} : c \in \rho^*(c_1)$ and $c \in \rho^*(c_2) \rightarrow c_1 \in \rho^*(c_2)$ or $c_2 \in \rho^*(c_1)$.

Optimal refinement operators are more efficient than classical refinement operators because they generate each candidate clause exactly once. A known problem with classical refinement operators is that they generate candidate clauses (and their refinements) more than once, making the search intractable. Optimality is thus desirable for efficiency reasons.

3.2 Pruning

Our clausal discovery algorithm employs a complete top-down search of the refinement graph. In this section, we present two lemmas that allow to prune a clause (and its refinements), without loss of completeness.

Lemma 3 For any refinement operator ρ , and clauses c, c' such that $c' \in \rho^*(c) : c \models c'$.

Clauses already implied by the current theory may be pruned without losing completeness. The lemma states that also the refinements of such clauses will be implied by the theory. Therefore, when a clause is implied, the clause together with all its refinements may be pruned.

When the considered language is fair, the search can also be pruned by Lemma 4.

Definition 12 A language $\mathcal{L}$ is fair if and only if $\forall c_1, c_2, c_3 \in \mathcal{L}$ and substitutions $\theta_1, \theta_2, c_1\theta_1 \subset c_2$ and $c_2\theta_2 \subset c_3 \rightarrow c_1\theta_1\theta_2 \cup (c_3 - c_2\theta_2) \in \mathcal{L}$.

Suppose $c_1 = A, c_2 = A\theta_1 \cup B, c_3 = A\theta_1\theta_2 \cup B\theta_2 \cup C$, where A, B, C represent disjunctions of literals (i.e. clauses). For an illustration of fairness and its use, see the example below Lemma 4.

Lemma 4 Given a database D , a fair language $\mathcal{L}$, and $\forall c_1 \in \mathcal{L}, c_2 \in \rho^*(c_1), c_1\theta_1 \subset c_2 : \text{Comp}(D) \models (c_2 - c_1\theta_1) \rightarrow c_1\theta_1$, then c_2 and its refinements may be pruned without losing completeness.

The lemma states that refinements $c_2 = c_1\theta_1 \cup d$ of c_1 contain redundant information if d implies $c_1\theta_1$ with regard to $\text{Comp}(D)$. Given a fair language $\mathcal{L}$ we may prune c_2 as well as its refinements. To illustrate the

lemma, reconsider Example 1 and assume $c_1 = \neg \text{male}(X)$ and $c_2 = \neg \text{male}(Y) \vee \neg \text{human}(Y)$. $c_2 - c_1\theta_1$ implies $c_1\theta_1$, i.e. $\neg \text{human}(Y)$ implies $\neg \text{male}(Y)$ in $\text{Comp}(D)$, therefore c_2 contains redundant information ($\text{human}(Y)$), and may be pruned. Fairness requires that if clause $c_3 = \neg \text{male}(Z) \vee \neg \text{human}(Z) \vee \neg \text{tall}(Z) \in \mathcal{L}$, also clause $c_4 = \neg \text{male}(Z) \vee \neg \text{tall}(Z) \in \mathcal{L}$. If the considered language is fair, pruning c_1 is allowed as c_4 , which is equivalent to c_3 with regard to $\text{Comp}(D)$ but θ -subsumes c_3 , will be considered.

3.3 A clausal discovery algorithm

```

T := ∅
Q := {⊤}
while Q ≠ ∅ do
  delete c = (h ← b) from Q
  if ← b ∧ ¬h finitely fails using SLDNF on D ▷ 0 ◁
  then if T ⊈ c ▷ 1 ◁
        then add c to T
  else for all c' ∈ ρ(c) where c' = cθ ∪ d do
        if ← d ∧ ¬cθ does not finitely fail
          under SLDNF on D ▷ 2 ◁
        then add c' to Q
for all clauses c in T do
  select next clause c from T
  if T - {c} ⊢ c ▷ 3 ◁
  then delete c from T

```

This algorithm defines CLAUDIEN: it starts from an empty theory T , and a queue Q containing only the most general element in the considered language $\mathcal{L}$. It then applies a search process where each element c is deleted from the queue Q , and tested for satisfaction with regard to the database D . If the clause satisfies the database, and is not yet implied by the current theory, it is added to the theory. If the clause does not satisfy the database, its refinements are considered. Because of Lemma 4, some refinements can be pruned if the language $\mathcal{L}$ is fair. If the language is not fair, the condition at $\triangleright 2 \triangleleft$ should be removed. Let us therefore refer to the algorithm, including the condition at point $\triangleright 2 \triangleleft$ as CLAUDIEN.FAIR, and to the algorithm without this condition as CLAUDIEN.SIMPLE. When the search process terminates, redundant clauses are removed from the theory. This is needed for finding a minimal theory. Indeed, CLAUDIEN may find clauses c_1 and c_2 in the specified order, and it may be that $c_1 \not\models c_2$ but $c_2 \models c_1$. The select function in the for-loop has to select the clauses in the reverse order they were learned (LIFO-strategy).

Notice that the algorithm does not specify the search strategy. To discover clausal theories, one can use essentially any search-strategy, which includes: depth-first, breadth-first, beam-search, best-first, iterative deepening, etc. Our current experiments (see Section 4) show that depth-first iterative deepening performs very well.

Throughout this paper, we will assume that Algorithm 1 always terminates, an assumption which does not hold in general. Indeed, CLAUDIEN may not terminate because of two reasons: the refinement graph may be infinite (which may lead to infinite paths through the search

space), or the tests whether $T \vdash c$ and/or tests involving SLDNF may not terminate. Using a bound on the depth of the tree, SLDNF can be forced to terminate. If SLDNF does neither succeed nor fail within the bounds, we assume there is no finite failure. For DATALOG, finite languages and a well-founded semantics, a variant of this algorithm can be designed that always terminates.

3.4 A detailed example with CLAUDIEN

Suppose we start CLAUDIEN from the database D and language $\mathcal{L}$ specified in Example 1. Notice that $\mathcal{L}$ is fair. A simple optimal refinement operator σ can be defined as follows:

1. $\sigma(T) = \{ \neg l \mid l \in B^+ \}$
where $B^+ = \{ \text{female}(X), \text{male}(X), \text{human}(X) \}$
2. $\sigma(l_1 \vee \dots \vee l_n) = \{ l_1 \vee \dots \vee l_n \vee l_{n+1} \mid l_{n+1} \in B$
and for all $i : l_i \neq l_{n+1}, \neg l_i \neq l_{n+1}$ and l_i comes
alphabetically before $l_{n+1} \}$,
where $B = B^+ \cup \{ \neg \text{female}(X), \neg \text{male}(X), \neg \text{human}(X) \}$

Employing breadth-first results in the following steps (the indices indicate the number of previous steps through the while-loop):

$T_0 = \emptyset; Q_0 = \{ T \};$
 $T_1 = \emptyset; Q_1 = \sigma(T) = \{ c_1, c_2, c_3 \}$
 $T_2 = \emptyset; Q_2 = \{ c_2, c_3, c_5, c_6 \}$
 $T_3 = \emptyset; Q_3 = \{ c_3, c_5, c_6, c_8, c_9, c_{10}, c_{11} \}$
 $T_4 = \emptyset; Q_4 = \{ c_5, c_6, c_8, c_9, c_{10}, c_{11}, c_{13} \}$
 $T_5 = \{ c_5 \}; Q_5 = \{ c_6, c_8, c_9, c_{10}, c_{11}, c_{13} \}$
 $T_6 = \{ c_5, c_6 \}; Q_6 = \{ c_8, c_9, c_{10}, c_{11}, c_{13} \}$
 $T_7 = \{ c_5, c_6 \}; Q_7 = \{ c_9, c_{10}, c_{11}, c_{13}, c_{14} \}$
 $T_8 = \{ c_5, c_6 \}; Q_8 = \{ c_{10}, c_{11}, c_{13}, c_{14} \}$
 $T_9 = \{ c_5, c_6 \}; Q_9 = \{ c_{11}, c_{13}, c_{14}, c_{17} \}$
 $T_{10} = \{ c_5, c_6 \}; Q_{10} = \{ c_{13}, c_{14}, c_{17} \}$
 $T_{11} = \{ c_5, c_6, c_{13} \}; Q_{11} = \{ c_{14}, c_{17} \}$
 $T_{12} = \{ c_5, c_6, c_{13} \}; Q_{12} = \{ c_{17} \}$
 $T_{13} = \{ c_5, c_6, c_{13}, c_{17} \}; Q_{13} = \emptyset$

$\sigma(T) = \{ c_1 = \text{female}(X) \rightarrow ; c_2 = \text{human}(X) \rightarrow ;$
 $c_3 = \text{male}(X) \rightarrow \}$
 $\sigma(c_1) = \{ c_4 = \text{female}(X) \wedge \text{human}(X) \rightarrow (\text{pruned } \triangleright 2 \triangleleft);$
 $c_5 = \text{female}(X) \wedge \text{male}(X) \rightarrow ; c_6 = \text{female}(X) \rightarrow \text{human}(X);$
 $c_7 = \text{female}(X) \rightarrow \text{male}(X) (\text{pruned } \triangleright 2 \triangleleft) \}$
 $\sigma(c_2) = \{ c_8 = \text{human}(X) \wedge \text{female}(X) \rightarrow ; c_9 = \text{human}(X) \wedge$
 $\text{male}(X) \rightarrow ; c_{10} = \text{human}(X) \rightarrow \text{female}(X); c_{11} = \text{human}(X)$
 $\rightarrow \text{male}(X) \}$
 $\sigma(c_3) = \{ c_{12} = \text{male}(X) \rightarrow \text{female}(X) (\text{pruned } \triangleright 2 \triangleleft);$
 $c_{13} = \text{male}(X) \rightarrow \text{human}(X) \}$
 $\sigma(c_8) = \{ c_{14} = \text{human}(X) \wedge \text{female}(X) \wedge \text{male}(X) \rightarrow$
 $(\text{pruned } \triangleright 1 \triangleleft); c_{15} = \text{human}(X) \wedge \text{female}(X) \rightarrow \text{male}(X)$
 $(\text{pruned } \triangleright 2 \triangleleft) \}$
 $\sigma(c_9) = \{ c_{16} = \text{human}(X) \wedge \text{male}(X) \rightarrow \text{female}(X) (\text{pruned}$
 $\triangleright 2 \triangleleft) \}$
 $\sigma(c_{10}) = \{ c_{17} = \text{human}(X) \rightarrow \text{female}(X) \vee \text{male}(X) \}$
 $\sigma(c_{11}) = \emptyset$

3.5 Properties of CLAUDIEN

In this section, we show that CLAUDIEN, when it terminates and SLDNF proofs always terminate with success or finite failure, discovers minimally complete theories satisfying a given database.

Theorem 1 *Given a database D , a language $\mathcal{L}$, and a refinement operator ρ for $\mathcal{L}$, CLAUDIEN.SIMPLE discovers a minimally complete theory satisfying D .*

Theorem 2 *Given a database D , a fair language $\mathcal{L}$, and a refinement operator ρ for $\mathcal{L}$, CLAUDIEN.FAIR discovers a minimally complete theory satisfying D .*

4 Practical aspects

In this section, we discuss the implementation of CLAUDIEN, its intended use and applications, as well as some experiments.

4.1 Implementation of CLAUDIEN

A prototype of Algorithm 1, named CLAUDIEN, has been implemented in Prolog by BIM. It uses depth-first iterative deepening (cf. [Korf, 1985]) together with a clever indexing scheme. Depth-first iterative deepening is the right strategy for CLAUDIEN, because

- It has essentially the same behaviour as breadth-first, which makes that simple constraints are discovered before more complex ones.
- It has the memory-efficiency of depth-first search but cannot get lost in infinite paths.
- To avoid the expensive (re)computation of the tests at points $\triangleright 0, 1, 2 \triangleleft$, we store the result of these tests using a clever indexing scheme (for each clause c we encode the result of the tests in an integer; in the next iterations the clauses c are recomputed but the corresponding tests are not re-evaluated as the result can be looked up in a table). This requires some additional memory, but saves a lot of (re)computation as all tests are done only once.³

We currently employ an optimal refinement operator for a fair language of range-restricted clauses containing neither functors nor constants. The refinement operator refines a clause by adding one literal at a time.

As T is not a program but a clausal theory, we employ Mark Stickel's fast theorem prover [Stickel, 1988] to check whether $T \vdash c$ at points $\triangleright 1, 3 \triangleleft$. This works very well in practice. Tests involving SLDNF resolution are currently realized using depth bounded SLDNF-resolution.

4.2 User involvement

When applying CLAUDIEN on a database containing a large number of predicates, CLAUDIEN has to search a vast search space. This is impractical for all but the simplest databases. Therefore, we believe that it is crucial that the user supplies additional constraints to CLAUDIEN. In the current implementation, the user has several options available:

- The user can ask CLAUDIEN to consider only clauses involving certain predicates in the head, and certain predicates in the body (this corresponds to stating a kind of clausal determination). Such

³ Alternatively, as in FOIL, to save computation, one could use the result of test $\triangleright 0 \triangleleft$ for clause c when evaluating $\triangleright 0 \triangleleft$ for a refinement $c' \in \rho(c)$.

knowledge is often available to domain experts, who have certain expectations about the relevant regularities in their domain. These expectations significantly reduce the search, and can be done incrementally: first look for relations between predicates p_i and then for relations between predicates q_j , etc. The user can also specify the maximum number of variables and literals in a clause.

- The user can impose certain criteria on constraints to be acceptable. For instance, to allow for noise handling, the user can relax the condition $Comp(D) \vdash_{SLDNF} c$ into $(p/t) > X$, where $p = |\{\theta \mid Comp(D) \vdash_{SLDNF} body(c)\theta \wedge head(c)\theta\}|$ and $t = |\{\theta \mid Comp(D) \vdash_{SLDNF} body(c)\theta\}|$, where X is a parameter indicating in how much per cent of the substitutions the clause is satisfied. $|S|$ is the cardinality of S . The user can require that clauses c with a non-empty head have a value for $t > Y$, reflecting a principle that clauses matching only few cases, are less reliable.
- The user can restrict the form of clauses such that only definite clauses are considered.

A planned extension of the current implementation, is to provide the user with a refinement operator based on schemata as in the MOBAL - BLIP system [Kietz and Wrobel, 1992]. A schema is a second-order expression involving predicate variables, such as $\exists P : P(X, Y) \wedge P(X, Z) \rightarrow Y = Z$. The notion of θ subsumption has been extended towards this second-order expressions by [Kietz and Wrobel, 1992]. Using second-order schemata, CLAUDIEN can derive specific forms of clauses. Given the above expression, CLAUDIEN would derive functional dependencies. In this context, CLAUDIEN can potentially and efficiently simulate some of the work in knowledge discovery in databases that focuses on discovering such specific dependencies (cf. [Kantola et al., 1992, Flach, 1990]).

4.3 Illustration

We ran CLAUDIEN on a database containing family relations father, mother, parent, male, female and human. The language restricted clauses to contain at most two variables and at most 3 literals. Any combination of predicates in clauses was allowed. CLAUDIEN derived the following theory:

```
female(V36) ∧ male(V36) → false
male(V33) → human(V33) (1)
female(V33) → human(V33) (2)
human(V33) → female(V33) ∨ male(V33) (3)
parent(V33, V33) → false
mother(V33, V34) → parent(V33, V34)
father(V33, V34) → parent(V33, V34)
parent(V36, V37) ∧ parent(V37, V36) → false
father(V36, V37) ∧ mother(V36, V37) → false
parent(V33, V34) → father(V33, V34) ∨ mother(V33, V34)
parent(V33, V34) → human(V34)
parent(V33, V34) → human(V33)
mother(V33, V34) → female(V33)
father(V33, V34) → male(V33)
```

The induced theory contains most of the relevant information about the involved predicates. The theory can be used as an integrity theory for a database about the predicates. The theory also contributes to understanding the domain of family relations as the regularities in the theory are easily understandable. Clausal theories are also very powerful for making predictions. Indeed, if $parent(luc, soetkin)$ and $male(luc)$ is known we can derive $\neg mother(luc, soetkin)$, $father(luc, soetkin)$, $female(soetkin) \vee male(soetkin)$, $human(soetkin)$, etc. On the other hand, if only $father(luc, soetkin)$ is known we can derive $human(luc)$, $human(soetkin)$, $female(soetkin) \vee male(soetkin)$, $male(luc)$, $parent(luc, soetkin)$, $\neg female(luc)$. It is easy to see that using the above theory, prediction can start from basically any fact(s) for any set of predicates. This contrasts significantly from the usual inductive logic programming setting, where the inferences only go in one direction. Given the usual approaches, the learner would induce only clauses for specific predicates. E.g. the clauses (1) and (2) could be inferred, or alternatively the two normal clauses corresponding to clause (3), i.e. $male(X) \leftarrow human(X) \wedge \neg female(X)$ (4) and $female(X) \leftarrow human(X) \wedge \neg male(X)$ (5). Most of the current inductive logic programming systems would not allow to induce both (1-2) and (4-5), because the resulting program loops. Because of this restriction, the induced theory can only be used in one direction. Given (4-5) one can derive facts for male and female from facts for human, male and female. Given (1-2), facts for human can be derived from facts for male or female. Using clausal theories, one does not suffer from this restriction.

5 Conclusions and Related Work

CLAUDIEN significantly extends the current work in KDD and ILP:

First, clausal theories are much more expressive than the representations currently being used in KDD and ILP. This has two effects: 1) clausal theories can express regularities that cannot be expressed in the other representations (e.g. $parent(X, X) \rightarrow$ cannot be expressed using definite clause logic), and 2) clausal theories can represent some regularities in a more compact way (e.g. $human(X) \rightarrow female(X) \vee male(X)$ could be expressed by two normal program clauses $female(X) \leftarrow human(X) \wedge \neg male(X)$ and $male(X) \leftarrow human(X) \wedge \neg female(X)$). Property 2) makes clausal induction potentially more efficient than classical ILP as one full clause corresponds to several normal program clauses. Property 1) does not necessarily imply that clausal induction is computationally more expensive than classical ILP as the approach can be tuned towards specific forms of clauses by using schemata.

Second, the use of a different semantics puts several issues in ILP in a new perspective (see also [De Raedt and Lavrač, 1993]): because the intended model is more constrained than usually, fewer clauses are acceptable in our framework and empirically learning multiple predicates is plausible. Empirically learning multiple predicates is one of the unsolved problems in current ILP.

Indeed, most ILP systems, such as GOLEM [Muggleton and Feng, 1990] and FOIL [Quinlan, 1990], focus on learning a single predicate and may run into problems when learning multiple predicates (cf. [De Raedt *et al.*, 1993, De Raedt and Lavrač, 1993]).

Third, we have shown that CLAUDIEN can also be used in deductive databases as an aid to semi-automate the formulation of an integrity theory. In deductive databases, one could start from a small database that is known to be integer. Running CLAUDIEN on such a database can help in formulating the integrity theory.

Finally, there are several remaining issues for future work. To evaluate the applicability of CLAUDIEN, real-life experiments should be performed. Furthermore, heuristics may be needed in such applications to cope with noise and to consider only the most interesting clauses. We already proposed two heuristics (the percentage of covered substitutions and the minimum coverage of the body for clauses with a non-empty head), but further experience is needed to find out whether these are satisfactory. Another important issue for applying clausal discovery in practice is to provide the user with tools to declaratively specify bias. Such bias is needed to restrict the search space and to make clausal discovery feasible on large databases containing many predicates and facts. Our work also leaves some open theoretical questions about clausal discovery. For what languages do optimal refinement operators exist? What is the computational complexity of the discovery algorithm? How to handle non-termination elegantly?

Acknowledgements

The authors are grateful to Ivan Bratko, Marc Denecker, Peter Flach, Nada Lavrac, Bern Martens, Stephen Muggleton, Gunther Sablon and Marc Van Barel for discussions, suggestions and encouragements.

References

- [Bratko, 1992] I. Bratko. Applications of machine learning : towards knowledge synthesis. In *Proceedings of Future Generation Computing Systems*, 1992.
- [Chen and Warren, 1992] W. Chen and D.S. Warren. A goal oriented approach to computing well-founded semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 589–603. The MIT press, 1992.
- [De Raedt and Bruynooghe, 1992] L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence*, 53:291–307, 1992.
- [De Raedt and Bruynooghe, 1993] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, 1993. to appear.
- [De Raedt and Lavrač, 1993] L. De Raedt and N. Lavrač. The many faces of inductive logic programming. In J. Komorowski, editor, *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1993. invited paper, to appear.
- [De Raedt *et al.*, 1993] L. De Raedt, N. Lavrač, and S. Džeroski. Multiple predicate learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1993.
- [Flach, 1990] P.A. Flach. Inductive characterisation of database relations. In Z.W. Ras, M. Zemankova, and M.L. Emrich, editors, *Methodologies for Intelligent Systems*, pages 371–378. North-Holland, 1990.
- [Helft, 1989] N. Helft. Induction as nonmonotonic inference. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pages 149–156. Morgan Kaufmann, 1989.
- [Kantola *et al.*, 1992] M. Kantola, H. Mannila, K.J. Raiha, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7(7), 1992.
- [Kietz and Wrobel, 1992] J-U. Kietz and S. Wrobel. Controlling the complexity of learning in logic through syntactic and task-oriented models. In S. Muggleton, editor, *Inductive Logic Programming*. Academic Press, 1992.
- [Korf, 1985] R. Korf. Depth-first iterative deepening : an optimal admissible search. *Artificial Intelligence*, 1985.
- [Lloyd, 1987] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 2nd edition, 1987.
- [Minker, 1987] J. Minker, editor. *Foundations of deductive databases and logic programming*. Morgan Kaufmann, 1987.
- [Muggleton and Feng, 1990] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st conference on algorithmic learning theory*. Ohmsma, Tokyo, Japan, 1990.
- [Muggleton, 1992] S. Muggleton, editor. *Inductive Logic Programming*. Academic Press, 1992.
- [Piatetsky-Shapiro and Frawley, 1991] G. Piatetsky-Shapiro and W. Frawley, editors. *Knowledge discovery in databases*. The MIT press, 1991.
- [Quinlan, 1990] J.R. Quinlan. Learning logical definition from relations. *Machine Learning*, 5:239–266, 1990.
- [Shen, 1992] W.M. Shen. Discovering regularities from knowledge bases. *International Journal of Intelligent Systems*, 7(7), 1992.
- [Stickel, 1988] M.E. Stickel. A prolog technology theorem prover: implementation by an extended prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.
- [Van Gelder *et al.*, 1991] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3), 1991.