

Clausal Discovery

Luc De Raedt

Luc Dehaspe

Report CW 238, September 3, 1996

Department of Computing Science, K.U.Leuven

Abstract

The clausal discovery engine *CLAUDIEN* is presented. *CLAUDIEN* is an inductive logic programming engine that fits in the knowledge discovery in databases and data mining paradigm as it discovers regularities that are valid in data. As such *CLAUDIEN* performs a novel induction task, which is called characteristic induction from closed observations, and which is related to existing formalizations of induction in logic. In characterising induction from closed observations, the regularities are represented by clausal theories, and the data using Herbrand interpretations. *CLAUDIEN* also employs a novel declarative bias mechanism to define the set of clauses that may appear in a hypothesis.

Keywords : Inductive Logic Programming, Knowledge Discovery in Databases, Data Mining, Learning, Induction, Semantics for Induction, Logic of Induction, Parallel Learning.

1 Introduction

Despite the fact that the areas of knowledge discovery in databases [Fayyad *et al.*, 1995] and inductive logic programming [Muggleton and De Raedt, 1994] have both enjoyed a lot of attention recently, the combination of the two areas has seldomly been studied [Džeroski, 1995]. Enhancing data mining tools with relational abilities as offered by inductive logic programming is of crucial importance for the practice of knowledge discovery due to the central role of relational databases in database technology. Yet, most data mining techniques focus on learning within a single relation. On the other hand, inductive logic programming has always focused on learning classification rules, i.e. on performing concept-learning from positive and negative examples of a concept. In contrast, data mining is often aimed at finding interesting regularities in unclassified data.

CLAUDIEN¹ combines data mining principles with inductive logic programming. It can be considered the first system that discovers clausal regularities from unclassified data. To this aim, a novel semantics for inductive logic programming has been developed, cf. [De Raedt and Džeroski, 1994], in which examples are represented by Herbrand interpretations and the aim is to discover a logically maximally general hypothesis that has all the examples as models. The novel semantics is said to define characteristic induction from closed observations. The special case, where the data consists of a single model corresponds to the typical data mining situation and was earlier proposed in a slightly different form by Nicolas Helft [Helft, 1989]. The semantics is compared and contrasted with other formalizations of inductive logic programming and its various properties are presented. One of the properties of the proposed semantics is monotonicity, meaning that whenever two individual clauses are valid on the data, their conjunction will also be valid. Monotonicity is not satisfied by the explanatory inductive logic programming semantics. Monotonicity makes it easy to implement a parallel clausal discovery engine. Algorithms that implement the proposed semantics are presented, shown to be correct and tested on a wide range of applications.

A key ingredient of the clausal discovery engine is the definition of the declarative bias, which determines the type of regularity searched for. Declarative bias is essential in data mining as data mining systems have a less operational criterion of success than concept-learning. In concept-learning, one typically searches for any hypothesis consistent with the data whereas data mining is looking for all interesting or valid regularities. The number of regularities satisfying the criterion can be very large as shown also in propositional approaches to data mining. As the search space of clausal logic is larger (and even infinite) than that of propositional logic, bias is of crucial importance in clausal discovery. To declaratively represent the bias of the clausal discovery engine, a new formalism, called *DLAB*, derived from the work of [Adé *et al.*, 1995; Emde *et al.*, 1983; Kietz and Wrobel, 1992; Bergadano and Gunetti, 1993; Cohen, 1994] is proposed. Moreover, it is shown how the specification of the syntax of the clauses allowed in the hypothesis can be automatically translated in a refinement operator for the considered language. *DLAB* should also be useful in other inductive logic programming systems.

The practice of the clausal discovery engine is demonstrated using a variety of exper-

¹Details on how to obtain *CLAUDIEN* can be found on the World-Wide-Web at URL <http://www.cs.kuleuven.ac.be/~cwis/research/ai/Research/clauidien-E.shtml>

iments. The first of the experiments demonstrates the generality of the clausal discovery engine in a data mining context by showing that the engine is able to emulate many of the data mining systems specifically designed for particular induction tasks such as finding functional or multi-valued dependencies and association rules. By tuning *CLAUDIEN*'s parameters, especially the declarative bias, *CLAUDIEN* is able to address these tasks. The second experiment, in finite element mesh-design [Dolšak and Muggleton, 1992; Lavrač and Džeroski, 1994], shows that – although *CLAUDIEN* is not intended to perform classification tasks – it can also be successfully applied in this context. Two further experiments (on mutagenesis [Srinivasan *et al.*, 1995b]) and water-quality ([Džeroski *et al.*, 1994]) show *CLAUDIEN*'s performance on particular data mining tasks.

This paper is organised as follows: In Section 2, we review the concepts from (inductive) logic programming used, in Section 3, we introduce the novel semantics for inductive logic programming and contrast it with existing ones, in Section 4, we present a sequential and parallel algorithm for performing clausal discovery, we introduce a novel mechanism to declaratively represent the bias of the discovery engine, and present heuristics and extensions of the proposed algorithm, in Section 5, we show the effectiveness of the engine on a wide range of applications. Finally, in Sections 6 and 7, we conclude and touch upon related work.

2 (Inductive) Logic Programming Concepts

We assume familiarity with first order logic and model theory (see [Lloyd, 1987; Genesereth and Nilsson, 1987] for an introduction).

A first order alphabet is a set of predicate symbols, constant symbols and functor symbols. A clause is a formula of the form $A_1, \dots, A_m \leftarrow B_1, \dots, B_n$ where the A_i and B_i are logical atoms. An atom $p(t_1, \dots, t_n)$ is a predicate symbol p followed by a bracketed n -tuple of terms t_i . A term t is a variable V or a function symbol $f(t_1, \dots, t_k)$ immediately followed by a bracketed n -tuple of terms t_i . Constants are function symbols of arity 0. *Functor-free* clauses are clauses that contain only variables as terms.

The above clause can be read as A_1 or ... or A_m if B_1 and ... and B_n . All variables in clauses are universally quantified, although this is not explicitly written. Extending the usual convention for *definite clauses* (where $m = 1$), we call A_1, \dots, A_m the *head* of the clause and B_1, \dots, B_n the *body* of the clause. A *fact* is a definite clause with an empty body, ($m = 1, n = 0$).

A *Herbrand interpretation* over a first order alphabet is a set of ground facts constructed with the predicate, constant and functor symbols in the alphabet. A Herbrand interpretation I is a model for a clause c if and only if for all grounding substitutions θ of c : $body(c)\theta \subset I \rightarrow head(c)\theta \cap I \neq \emptyset$. We also say c is true in I or c makes the interpretation I true. A Herbrand interpretation I is a model for a clausal theory T if and only if it is a model for all clauses in T . Roughly speaking, the truth of a clause c in an interpretation I can be determined by running the query $? - body(c), not head(c)$ on a database containing I using a theorem prover (such as PROLOG). If the query succeeds, the clause is false in I . If it finitely fails, the clause is true².

²This method is designed for range restricted clauses. A clause c is range restricted if and only if $vars(head(c)) \subset vars(body(c))$.

The least Herbrand interpretation of a definite clause theory is the set of all ground facts (using the predicates, functors and constants of the definite clause theory) that are logically entailed by the definite clause theory. We will use the notation $M(T)$ to denote the least Herbrand model of a definite clause theory T .

These notions are illustrated in Example 1.

Example 1 Consider the following definite clause theory:

$$\begin{aligned} \text{flies}(X) &\leftarrow \text{normal}(X), \text{bird}(X) \\ \text{normal}(\text{tweety}) &\leftarrow \\ \text{bird}(\text{tweety}) &\leftarrow \end{aligned}$$

Then the least Herbrand model of this theory is:

$$\{\text{bird}(\text{tweety}), \text{normal}(\text{tweety}), \text{flies}(\text{tweety})\}$$

This Herbrand interpretation is a model for the clause:

$$\text{flies}(X) \leftarrow \text{bird}(X)$$

The following clause is false in the Herbrand interpretation:

$$\leftarrow \text{bird}(X), \text{normal}(X)$$

□

We will employ two notions of generality in this paper. A clausal theory T_1 is *logically more general than* a clausal theory T_2 if and only if $T_1 \models T_2$, i.e. if T_1 logically entails T_2 . The other notion employed is that of θ -subsumption among clauses. A clause c_1 θ -subsumes clause c_2 if and only if there exists a substitution θ such that $c_1\theta \subseteq c_2$.

3 Logical Frameworks for Induction

At present, there exist several formalisations of induction in clausal logic. Firstly, there is the explanatory or normal setting introduced by Gordon Plotkin [Plotkin, 1970], which is employed by the large majority of inductive logic programming systems, cf. [Muggleton and De Raedt, 1994], which aims at discriminating positive observations from negative ones, and hence is classification oriented. Secondly, there is Nicolas Helft's non-monotonic setting [Helft, 1989], which aims at characterising one or more observations, and hence is oriented towards knowledge discovery. Thirdly, there is the confirmatory by Peter Flach [Flach, 1995]. Fourthly, there is Mannila's general framework for data mining (cf. [Mannila, 1995]). Fifth, there is the setting introduced by De Raedt and Džeroski [De Raedt and Džeroski, 1994], which we will employ for clausal discovery, and which we will call characterizing induction from closed observations³. In this section, we will introduce this induction setting and discuss its relation to the other ones.

³There is some historical confusion in terminology here. Helft [Helft, 1989] introduced the term non-monotonic induction, Flach first distinguished weak induction from strong or normal induction [Flach, 1992], but now uses confirmatory and explanatory induction [Flach, 1994; Flach, 1995]. Finally, though the setting by [De Raedt and Džeroski, 1994] is a generalization of Helft's setting, they also used the term non-monotonic.

3.1 Closed observations

3.1.1 Basic principles

Our setting for induction is derived from Nicolas Helft’s non-monotonic semantics for induction [Helft, 1989], cf. [De Raedt and Džeroski, 1994]. Although it differs from Helft’s setting in several respects, it is similar in spirit. The ideas are 1) that all observations are completely specified, and 2) that a hypothesis should reflect what is in the data. The first idea is implemented by representing the observations as Herbrand interpretations, with the consequence that all examples are assumed to be completely specified. Hence the term *closed*, in analogy with the closed world assumption. The second idea is enforced by requiring all hypotheses to be true in all of the observations. Since we are only working with one type of observation, the aim is to characterize them. Hence the term *characterizing induction*.

Characterizing induction from closed observations can be defined as follows.

Definition 1 (Characterizing induction from closed observations) *Let O be a set of observations, B a background theory, \mathcal{L} a set of clauses. $H \subset \mathcal{L}$ is a solution if and only if H is a logically maximally general valid hypothesis. A hypothesis H is valid if and only if for all $o_i \in O$, H is true in all⁴ minimal Herbrand models of $B \cup o_i$.*

In our framework for characterizing induction, we will impose some syntactic restrictions on the space of hypotheses. This is captured in the language assumption.

Definition 2 (Language assumption) *The language assumption states that the alphabet of the hypotheses language \mathcal{L} will only contain constant, functor or predicate symbols that occur in one of the observations or in the background theory.*

Let us first illustrate the definition.

Example 2 *Imagine we are observing different gorilla colonies, and our background theory B consists of*

$gorilla(X) \leftarrow female(X)$
 $gorilla(X) \leftarrow male(X)$

and we observe two different colonies

$o_1 = \{female(liz), male(richard)\}$ and
 $o_2 = \{female(ginger), male(fred), male(rudolph)\}$.

If \mathcal{L} is restricted to range-restricted, constant-free clauses a solution is:

(1) $gorilla(X) \leftarrow female(X)$
(2) $gorilla(X) \leftarrow male(X)$

⁴Helft distinguishes two different semantics: one where truth in *all* minimal models is required, and one where truth in a *single* minimal model is required. As for most practical situations, using definite clauses, the minimal Herbrand model will be unique, we will not further distinguish these two Helft settings.

(3) $male(X), female(X) \leftarrow gorilla(X)$

(4) $\leftarrow male(X), female(X)$

This is a solution because all clauses (1-4) are true in both minimal Herbrand models $M(B \cup o_1)$ and $M(B \cup o_2)$, i.e.

$M(B \cup o_1) = \{ female(liz), male(richard), gorilla(liz), gorilla(richard) \}$

$M(B \cup o_2) = \{ female(ginger), male(fred), male(rudolph),$
 $gorilla(ginger), gorilla(fred), gorilla(rudolph) \}.$

Furthermore, all other valid clauses over the same alphabet are logically entailed by this hypothesis. To see this, observe that as all predicates are unary and there are only three predicates, it suffices to restrict our attention to clauses with at most 3 literals in the head and at most 3 literals in the body as all clauses with more literals are equivalent to one of this form. The result then follows by enumerating the clauses, and removing logically redundant ones.

3.1.2 Properties

Let us now elaborate on the different choices and properties in characterizing induction from closed observations.

First, the models considered are minimal Herbrand models. The use of minimal Herbrand models restricts the domain of the model to the Herbrand universe, i.e. to the set of all terms constructable with the constant and functor-symbols appearing in the $B \cup o_i$. This is important because without the restriction to *Herbrand* models, as in Helft's framework, clause (4) would not be true in all minimal models. Also, employing minimal models is only justified when complete knowledge of all (relevant) aspects of the observation is available in $B \cup o_i$. To illustrate this, consider the following example in which this assumption does not hold. Suppose we have two birds, the first of which is known to be black, and the second having an unknown colour. Under these circumstances, it is not valid to say that all birds are black (as we do not know whether this statement holds for the second bird). Thus the use of minimal Herbrand models assumes complete knowledge of the elements in the domain relevant to the observation o_i . If such knowledge is not available one should be cautious when using the closed observations.

Second, we are interested in hypotheses that are valid in the models. Intuitively, validity means that the hypothesis holds on the data, i.e. that the induced hypothesis postulates true regularities present in the models. This is – as we shall see – a stronger requirement than those employed in the explanatory framework. Validity is a monotone property at the level of hypotheses:

Property 1 (Monotonicity) *If H_1 is valid and H_2 is valid with respect to a background theory B and a set of observations O , then $H_1 \cup H_2$ is valid.*

This property means that all well-formed clauses in \mathcal{L} can be considered completely independent of each other. It will turn out to be very important for efficiency reasons as it essentially allows for parallel search (cf. Section 4.3). Monotonicity does not hold when the

validity requirement is relaxed such that the hypothesis should be true only in *some* models, which has also been proposed by Helft.

Third, the condition of maximal generality. This condition appears in the definition because the most interesting hypotheses are the most informative and hence the most general. Without this condition, the empty hypothesis (which is always valid) would be a trivial solution and this is undesirable. The question then arises as to the circumstances under which a maximally general valid hypothesis exists. In general, for infinite hypotheses spaces, a maximally general hypothesis will not exist. This is demonstrated in Example 3.

Example 3 *Let B be \emptyset and O be $\{\text{parent}(\text{luc}, \text{soetkin}) \leftarrow\}$. Then the following clauses are all valid:*

- (1) $\leftarrow \text{parent}(X_1, X_1)$
- (2) $\leftarrow \text{parent}(X_1, X_2), \text{parent}(X_2, X_1)$
- (3) $\leftarrow \text{parent}(X_1, X_2), \text{parent}(X_2, X_3), \text{parent}(X_3, X_1)$
- ...

It is clear that there exists here a strictly ascending chain (according to generality) of clauses which are all valid. If we restrict \mathcal{L} to this set of clauses, the maximally general hypothesis should be an infinite clause.

However, in case a maximally general hypothesis exists, then all such hypotheses are logically equivalent. This can be formalised as follows:

Property 2 *If there exists a solution, then the solution is unique up to logical equivalence.*

Proof: suppose there are two maximally general solutions H_1 and H_2 and $\not\vdash H_1 \leftrightarrow H_2$. Because of monotonicity $H_1 \cup H_2$ must also be valid, and $H_1 \cup H_2$ is strictly more general than H_1 and than H_2 . This contradicts the fact that H_1 and H_2 are maximally general. \square

There are two possible ways to avoid the problems with infinite solutions. The first solution is to require that the set of well-formed clauses \mathcal{L} is finite. Although this solution may appear to be undesirable, it is made by the vast majority of current approaches to inductive logic programming. The second solution is due to Nicolas Helft (but generalized here) and works only when the Herbrand Universes are finite. It employs a notion of injectivity.

Definition 3 (Injectivity) *Let c be $p_1, \dots, p_m \leftarrow q_1, \dots, q_n$ and let $\text{vars}(c) = \{X_1, \dots, X_k\}$. The clause c is injective with regard to a background theory B and a set of observations O if and only if either, $m > 0$ and there exists an observation $o \in O$, and a substitution θ such that $(q_1 \wedge \dots \wedge q_n \wedge X_1 \neq X_2, \dots, X_i \neq X_j, \dots)\theta$ is true in all minimal Herbrand models of $B \cup o$ augmented with standard inequality, or, $m = 0$ and for all k , clause $\neg q_k \leftarrow q_1, \dots, q_{k-1}, q_{k+1}, \dots, q_n$ is injective.*

Definition 4 (Injectivity assumption) *The injectivity assumption requires that all clauses in a solution be injective.*

The problems with Example 3 disappear when the injectivity assumption is made. Indeed, the unique maximally general injective valid clause is clause (2). The intuition here is that one should not employ more variables than needed, and as the maximum chain of constants linked by the parent relation is 2, we should not introduce more variables. Analogously, if we would replace the observation O in Example 3 by $O = \{\text{parent}(\text{rose}, \text{luc}), \text{parent}(\text{luc}, \text{soetkin})\}$, clause (3) would be a solution.

Property 3 *If the Herbrand universes of the $B \cup o_i$ ($o_i \in O$) are finite and the injectivity assumption holds, then there exists a finite set of clauses that forms a solution.*

Proof: Let n be the maximum number of terms occurring in one of the Herbrand universes. Let X_1, \dots, X_n be n different variables. As each injective clause can contain at most n different variables, it suffices to consider clauses with as only variables the X_1, \dots, X_n . Therefore the only literals that need to be considered are those with the predicates of $B \cup o_i$, the terms in the Herbrand universes of the $B \cup o_i$, and the variables X_1, \dots, X_n . As there are only a finite number of such literals, the number of clauses to be considered is also finite. Let H contain all such clauses that are true in all the Herbrand models of all the $B \cup o_i$. H is finite and a injective solution. \square

The injectivity assumption, however, does not help when the Herbrand universe is infinite, see Example 4.

Example 4 *Let $B \cup o_i$ be $\{\text{parent}(X, p(X)), \text{human}(a)\}$. Then the problems outlined in Example 3 reappear.*

3.1.3 Additional options

A weaker but also useful condition than injectivity is that of non-triviality.

Definition 5 (Non-triviality) *Let c be $p_1, \dots, p_m \leftarrow q_1, \dots, q_n$. The clause c is non-trivial with regard to a background theory B and a set of observations O if and only if either $m > 0$ and there exists an observation $o \in O$ and a substitution θ such that $(q_1 \wedge \dots \wedge q_n)\theta$ is true in $M(B \cup o)$, or, $m = 0$ and for all k there exists a substitution θ and an observation o such that $(q_1 \wedge \dots \wedge q_{k-1} \wedge q_{k+1} \wedge q_n)\theta$ is true in $M(B \cup o)$.*

Non-triviality is used to exclude clauses that trivially hold from the hypotheses. Without non-triviality, one can always postulate implications, provided that the condition part never holds. This is illustrated in Example 5.

Example 5 *Consider as background theory:*

$\text{colour}(X) \leftarrow \text{black}(X)$
 $\text{colour}(X) \leftarrow \text{white}(X)$

and as observation $\{\text{swan}(s), \text{white}(s)\}$. Without requiring non-triviality the clause $\text{swan}(X) \leftarrow \text{black}(X)$ is valid. Clearly, this is undesirable.

An alternative to the non-triviality condition for denials would be to demand maximally general clauses.

Definition 6 (Maximally general clauses) *Under this assumption, it is required that all clauses c in a solution H , are maximally general and valid. This means that there is no clause c' that θ -subsumes c and is also valid on the observations⁵.*

The condition of maximally general clauses is however harder to enforce than non-triviality due to the possibility of strictly infinitely ascending chains of clauses under θ -subsumption, which may again lead to a need for adding infinite clauses to the hypotheses.

Another option relates to the issue of redundant hypotheses. Upon investigating Example 2, the reader may have noticed that the clauses that belong to the background theory reappear in the induced hypothesis. This is not always desirable. It can be avoided by the non-redundancy assumption.

Definition 7 (Non-redundancy) *Under the non-redundancy assumption, it is required that no clause $c \in H$ is logically entailed by B , i.e. for all $c \in H : B \not\models c$.*

A related requirement requires a minimal solution, i.e. a solution in which no clause is logically redundant with respect to the induced hypothesis.

Definition 8 (Compactness) *Under the compactness assumption, it is required that no clause $c \in H$ is logically entailed by $H - \{c\}$, i.e. for all $c \in H : H - \{c\} \not\models c$.*

3.2 Relation to other frameworks for induction

3.2.1 Helft's and Flach's notions

The key difference with Helft's notion of induction is that Helft assumes there is a single closed observation. Working with multiple observations is more natural as many well-known machine learning notions such as for instance incrementality have a clear meaning in our framework. Furthermore, by working with multiple observations, the boolean PAC-learning setting is generalized, cf. also [De Raedt and Džeroski, 1994]. Other differences with Helft's framework include the use of *Herbrand* models as well as that we allow for functors.

Flach's adequacy conditions for induction provide a framework for reasoning about the properties and semantics of induction. However, Flach's adequacy conditions allow for many instantiations. Our framework can be considered one such instantiation, which is close to Flach's *confirmatory setting*.

3.2.2 Explanatory Induction

Our setting for induction is specifically tailored towards the discovery of regularities that hold in a set of (unclassified) observations or that *characterize* the *closed* observations. Within inductive logic programming and other forms of machine learning, people have classically focused on learning rules that *discriminate* positive observations from negative ones. Within inductive logic programming this is captured in the following definition of explanatory induction, which is based on [Plotkin, 1970].

⁵One might as well use implication as a notion of generality, though this would be computationally harder.

Definition 9 (Explanatory induction) Let P be a set of true observations, N be a set of false observations, B a background theory. $H \subset \mathcal{L}$ is an explanatory solution if and only if H is complete with regard to the positive observations and consistent with regard to the negative observations. A hypothesis H is complete with regard to P and B if and only if $B \cup H \models P$; H is consistent with regard to N and B if and only if $B \cup H \cup N \not\models \square$.

This definition is illustrated in Example 6.

Example 6 Suppose $P = \{\text{flies}(\text{tweety}), \text{flies}(\text{woody})\}$, $N = \{\neg \text{flies}(\text{oliver})\}$, $B = \{\text{bird}(\text{tweety}), \text{bird}(\text{woody}), \text{bird}(\text{oliver}), \text{normal}(\text{tweety}), \text{normal}(\text{woody})\}$. Then an explanatory hypothesis would be $\text{flies}(X) \leftarrow \text{bird}(X), \text{normal}(X)$.

The aim of explanatory induction is to induce a hypothesis that logically entails all of the true observations and none of the false observations. An important property of explanatory induction is:

Property 4 If H_1 is consistent and H_2 is consistent with respect to a background theory B and a set of observations O , then $H_1 \cup H_2$ need not be consistent with O .

This property is the cause of some well-known problems when learning multiple predicates or recursive predicates in the explanatory induction setting, cf. [De Raedt *et al.*, 1993; Bergadano and Gunetti, 1993; Cameron-Jones and Quinlan, 1993]. The reason for this is that inconsistencies may arise when H_1 and H_2 can resolve together.

Flach's [Flach, 1992] definition of weak induction (from which his later notion of confirmatory induction is derived) is the special case of explanatory induction where only consistency with the negative examples is required. The reader may notice that also for Flach's setting, the above property holds.

The differences between our induction setting and explanatory induction are akin to the differences between knowledge discovery (or data mining) and concept-learning. The differences can be explained in terms of the two ideas underlying our induction setting, i.e. closed versus open observations and characterizing versus discriminant induction.

A first important difference is due to the completion of the observations into least Herbrand models in *closed observation* induction, which is not performed in explanatory induction. Using models to describe observations is the first order equivalent of what is done in attribute value learning. In attribute value learning each example is described by means of a complete vector of attribute value pairs. Completeness in this respect means that a value for each attribute is known. Working with models thus implicitly corresponds to assuming the closed world assumption on each observation: all examples are assumed to be completely described, and all facts not true in the least Herbrand model are regarded false. This contrasts with traditional inductive logic programming approaches where examples are definite clauses (possibly obtained after applying some form of saturation on a ground fact). Using definite clauses one can model incomplete information and induce hypotheses that realize an inductive leap on the examples. Let us illustrate this point using a variant of Example 6. The example can be straightforwardly transformed in a set of models, one model for each of the birds, i.e. *tweety*, *woody*, and *oliver*. In this case, complete knowledge of the birds is available. Now, both our setting and explanatory induction would consider

$flies(X) \leftarrow bird(X), normal(X)$ as (part of) a solution. However, let us assume that the fact $flies(tweety)$ is unknown. In explanatory induction the previous solution would still hold and the induction procedure would postulate that $flies(tweety)$ holds. Hence, an inductive leap would result. However, when working with closed observations it would no longer hold as there would be a normal bird of which it is not known whether it flies. This clearly shows that closed observation induction – in contrast to explanatory induction – assumes complete information about the examples and does not allow inductive leaps on the models, i.e. applying the induced hypotheses on the observations will not result in postulating new facts. Closed observation induction makes inductive leaps of a different kind, in the sense that it postulates that the induced hypotheses will be valid on unseen observations.

This is the theoretical point of view. In practise however, closed observation induction can still be applied in the presence of a limited form of incompleteness. The trick is to put the predicates that are known to be incomplete in the condition part of the rules. Thus, with $flies(tweety)$ unknown in Example 6, solutions in our setting would include $bird(X) \leftarrow flies(X)$ and $normal(X) \leftarrow flies(X)$. Notice we have then learned necessary conditions for $flies(X)$ instead of sufficient ones. From a theoretical perspective, one could handle incomplete information in closed observation induction by using incomplete models, which would list the known true, and the known false facts. A hypothesis H would then be considered valid with an observation o and a background theory B if and only if $B \wedge H \wedge o \not\models \square$, which again closely corresponds to Flach’s notion of weak induction. Some ideas along this line have also been investigated by [Fensel *et al.*, 1995]. From a practical perspective however, complete knowledge is often available (cf. attribute value learning where missing values arise only seldomly, or well-known inductive logic programming problems such as mutagenesis [Srinivasan *et al.*, 1995b]). Furthermore, it is the assumption of complete knowledge that makes the monotonicity property hold, which is crucial for efficiency reasons, cf. Section 4.3 on parallel search.

The second difference can be explained using the notions of characterizing induction versus discriminating induction. In discriminating induction, the aim is to find a hypothesis that discriminates observations belonging to two classes, i.e. the positive examples from the negative ones. In characterizing induction, the aim is to find a most informative hypothesis that explains all of the (unclassified) observations. A most informative hypothesis is one that covers the least number of examples. When working with closed observations, most informative means logically maximally general. The reason is that the logically more general hypotheses have the least number of models, hence, they cover the least number of observations (in this case a hypothesis covers an example if the example is valid in the hypothesis). In contrast, when working with explanatory induction, most informative means logically maximally specific, as these hypotheses cover the least observations (in this case a hypothesis covers an example if the hypothesis entails the example).

These two aspects of induction allow us also to describe two other problem settings that have been considered. First, there is the explanatory setting where the set of negative examples is empty. This setting can be described as *characterizing induction from open observations*, it corresponds to learning from positive data only, and has been considered by many researchers. Secondly, there is no reason why one cannot learn clauses that discriminate closed observations in several classes, e.g. interpretations that make a theory true versus interpretations that make a theory false. This alternative setting has been adopted in the

ICL system of [De Raedt and Van Laer, 1995].

The ICL setting, discriminant induction from closed observations, provides a clue as how problems and solutions along the different dimensions relate to each other. It should be clear that the set of clauses output by a characteristic induction (using the positive observations only) is typically a superset of that produced by a discriminant procedure (we are ignoring all non-logical aspects of induction engines, such as heuristics, now). For instance, when working with closed observations, characteristic induction will produce a large set of clauses valid on the positive observations, whereas discriminant induction will retain a minimal subset needed for discriminating the negative observations. Transforming closed observations into open observations is also possible. Given a closed observation $\{f_1, \dots, f_n\}$ that forms a positive example, the open observation could be represented by e.g. $\oplus \leftarrow f_1, \dots, f_n$.

3.2.3 Mannila’s data mining framework

Heikki Mannila [Mannila, 1995] recently introduced a general definition for data mining. He views data mining as the process of constructing a theory $Th(\mathcal{L}, r, q)$, where \mathcal{L} is a set of sentences to consider, r the data(base), and q the quality criterion. The aim then is to find all sentences ϕ in the language \mathcal{L} that satisfy the quality criterion w.r.t. the data r , i.e.

$$Th(\mathcal{L}, r, q) = \{\phi \in \mathcal{L} \mid q(r, \phi(r)) \text{ is true}\}$$

It should be clear that our formalization of induction is a special case of Mannila’s one, where \mathcal{L} contains the clauses to consider, and the quality criterion q is true whenever the clause ϕ is valid on the data in r . This clearly shows that *CLAUDIEN* addresses a real data mining task.

4 A clausal discovery engine

This section provides a detailed description of our clausal discovery engine.

4.1 A Clausal Discovery Algorithm

The key to arrive at a clausal discovery algorithm for characterizing induction from closed observations is the well-known property/definition of logical entailment.

Property 5 (Pruning.) *Let G be a logical generalisation of S , i.e. $G \models S$. If an interpretation M is a model for G then M will also be a model of S .*

The contraposition states that if M is not a model for S then M will not be a model for any logical generalisation G of S . This contraposition shows that large parts of the search space can be pruned. Indeed, given an observation o , background theory B and hypothesis H such that H is false in $M(B \cup o)$ ⁶, all logical generalisations of H will be false in $M(B \cup o)$ and can thus be pruned.

⁶From now on, for convenience, we will assume the minimal model is unique.

By now, we can apply classical machine learning principles to obtain an algorithm for characterizing induction from closed observations. First, machine learning principles state that induction is a search process through a partially ordered space induced by the generalisation relation, cf. [Mitchell, 1982]. Second, machine learning systems typically search the space specific-to-general or general-to-specific. The question then arises as to which of these strategies is the most feasible one. Theoretically, there may however be a problem when searching specific-to-general as one should then start from the most specific hypothesis which could be an infinite one. Furthermore, it is well-known in machine learning that pruning parts of the search space is more reliable when working general-to-specific. Therefore, we will only consider general-to-specific search. Third, as characterizing induction aims at a maximally general hypothesis, it should not use a covering approach but rather an exhaustive search of the relevant parts of the search space.

In order to arrive at a general algorithm in Figure 1, we only need to define the search space and the operator for traversing it. In the remainder of this paper, we will use the notation \mathcal{L} to denote the search space consisting of clauses, and a refinement operator ρ based on θ -subsumption [Plotkin, 1970] to traverse it.

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. $\rho^*(\square) = \mathcal{L}$ where \square 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.

The algorithm in Figure 1 starts with an empty hypothesis H , 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 validity on the observations O . If the clause is valid, and not to be *pruned1* (see below), it is added to the hypothesis. If c is invalid, its refinements generated and those refinements which are not to be *pruned2* (see below) are added to the queue. When the queue is empty, the algorithm halts and outputs the current hypothesis.

The ClausalDiscovery algorithm has a number of parameters, which are printed in *italics*. They can be used to specify the many options of the clausal discovery engine. The *delete* function determines the search-strategy. When delete is first in first out one realizes breadth-first search, when it is last in first out then depth-first, when it is according to some ranking of the clauses, it is best-first. Different heuristics for ranking clauses are discussed in Section 4.6. The function *valid* determines when a clause is accepted as (part of) a solution. When

function ClausalDiscovery

inputs : O : set of Closed Observations, B : background theory, ρ : refinement operator

outputs : Characterizing Hypothesis

$H := \emptyset$

$Q := \{\square\}$

while $Q \neq \emptyset$ **do**

delete c from Q

if c is *valid* on O

 and not *prune1*(c)

then add c to H

else for all $c' \in \rho(c)$ for which not *prune2*(c') **do**

 add c' to Q

endfor

endif

endwhile

reduce(H)

endfunction

Figure 1: A clausal discovery algorithm

coping with noisy data it is often useful to relax the validity requirements as detailed in Section 4.5. The functions *prune1*, *prune2* and *reduce* are meant to implement the options presented in Section 3.1. The implementation of these options is discussed in Appendix A. A special type of pruning is possible when the language is fair, which is discussed in Section 4.2. Most important is the language bias and corresponding refinement operator. The declarative language bias mechanism *DLAB* and the corresponding refinement operators are discussed in Section 4.4. Finally, a parallel version of this algorithm is shown in Section 4.3.

4.2 Properties and Extensions

We first prove that the ClausalDiscovery engine is correct, and then discuss two extensions. The first extension allows to deal with infinite models, the second one is an optimization for *fair* languages. Other extensions of the ClausalDiscovery engine corresponding to the semantic biases of Section 3.1 are discussed in Appendix A. These extensions, concerning injectivity, maximally general clauses, non-triviality, non-redundancy, and compactness, allow *CLAUDIEN* to effectively prune the search for a valid hypothesis.

4.2.1 Property

Ignoring for the moment the functions *prune1*, *prune2*, and *reduce*, which are used to implement the options (cf. Appendix A), it is easy to see that:

Property 6 *ClausalDiscovery* outputs a maximally general valid hypothesis within $2^{\mathcal{L}}$ if it terminates and ρ is complete with regard to \mathcal{L} .

Proof: If the algorithm would perform an exhaustive search of \mathcal{L} and would add all valid clauses to H , the result trivially holds. Now, a clause c is only pruned when it is θ -subsumed by a valid clause $c' \in H$. Because c' logically entails c , H is as general as $H \wedge c$, implying that c may be pruned without losing information. \square

4.2.2 Termination

The algorithm may not always terminate because of two reasons:

- the refinement graph searched may be infinite, which may lead the algorithm to exploring infinite paths through the search-space;
- testing whether a clause is valid on an observation is only semi-decidable in the general case.

As discussed above, the first problem can be avoided when working with finite Herbrand Universes and using the injectivity assumption, or when using only finite languages. The second problem only arises when the Herbrand Universe of an observation is infinite, which may in turn make its minimal model infinite. Two approaches can be taken in this case. First, one can use an h -easy notion of validity (by setting the function *valid* accordingly).

Definition 12 (h -easy validity) *A clause c is h -easy valid on an observation o if and only if an SLDNF-interpreter (with depth-bound h) fails when answering the query $?-body(c)$, not $head(c)$. on the knowledge base $B \cup o$.*

SLDNF-resolution is the basis of the logic programming language PROLOG, see [Lloyd, 1987] for more details. By employing a depth-bound on the depth of the proof tree, termination is guaranteed. However, soundness is lost in the following sense. If a clause is h -easy valid, it may be invalid in the logical sense. When employing h -easy validity, this may result in finding a logically inconsistent hypothesis $H \models \square$, so care should be taken with this approach.

Second, one can approximate the infinite models by finite subsets of them, and one can then use a flattening approach [Rouveirol, 1994; De Raedt and Džeroski, 1994] to allow for clauses that have only infinite models. Since this approach is detailed in [De Raedt and Džeroski, 1994], we do not further elaborate on this here.

4.2.3 Fairness

An important optimisation is possible in case the language considered is *fair* (cf. [De Raedt and Bruynooghe, 1993]).

Definition 13 *A language \mathcal{L} is fair if and only if $\forall c_1, c_2, c_3 \in \mathcal{L}$ and substitution θ_2 , $c_1 \subset c_2$ and $c_2\theta_2 \subset c_3 \rightarrow c_1\theta_2 \cup (c_3 - c_2\theta_2) \in \mathcal{L}$.*

```

function Fair
  inputs :  $c_1, c_2$ : clauses such that  $c_2 \in \rho(c_1)$ 
  outputs : true if  $c_2$  may be pruned

if possible
then let  $c_2 = c_1 \cup A$ .
      return  $\forall o \in O : c_1 \leftarrow A$  is true in  $M(B \cup o)$ .
else return false
endif

```

Figure 2: Pruning clauses in case \mathcal{L} is fair

Suppose $c_1 = A$, $c_2 = A \cup B$, $c_3 = A\theta_2 \cup B\theta_2 \cup C$, where A, B, C represent disjunctions of literals (i.e. clauses). Fairness then requires that $A\theta_2 \cup C$ also belongs to the language \mathcal{L} . When the language is fair, setting $\text{Prune2}(c') = \text{Fair}(c, c')$ (cf. Figure 2) can be used to optimise the search.

Property 7 (Fairness) *If \mathcal{L} is a fair language, $\text{ClausalDiscovery}(\text{Prune2}(c') = \text{Fair}(c, c'))$ will return a characterizing hypothesis provided that it terminates.*

Proof: We will use $c_1 = c$ and $c_2 = c'$.

We must prove that if Fair succeeds, then $c' = c_2$ may be pruned together with all its refinements.

We first prove that c_2 is valid if and only if c_1 is valid, i.e. for all observations o , $c_2 \leftrightarrow c_1$ is true in $M(B \cup o)$. $\models c_1 \rightarrow c_2$ follows from c_1 θ -subsuming c_2 . The truth of $c_2 \rightarrow c_1$ in $M(B \cup o)$ follows from $c_2 \rightarrow c_1 \cup d$ and the fact that $d \rightarrow c_1$ is true in all $M(B \cup o)$.

Therefore, as c_1 is invalid, c_2 must also be invalid.

Now, it remains to be shown that refinements of c_2 , i.e. clauses c_3 such that $c_2\theta_2 \subset c_3$, may be pruned. Because of the fairness of the language, $c_4 = c_1\theta_2 \vee (c_3 - c_2\theta_2)$ also belongs to the language and will be considered at another time. So, if c_4 is equivalent to c_3 in all $M(B \cup o)$ then c_3 can be safely discarded as well. Let us now prove that $c_4 \leftrightarrow c_3$ is true in all $M(B \cup o)$. Let $c_1 = A$, $c_2 = A \vee B$, $c_3 = A\theta_2 \vee B\theta_2 \vee C$. Then $c_4 = A\theta_2 \vee C$.

A. Because of θ -subsumption : $\models c_4 \rightarrow c_3$.

B. We now prove $c_3 \rightarrow c_4$ is true in all $M(B \cup o)$:

Given : $(c_2 - c_1) \rightarrow c_1$ is true in all $M(B \cup o)$, i.e. $B \rightarrow A$ is true in all $M(B \cup o)$, (1)

(2) $\neg B \vee A$ is true in all $M(B \cup o)$ because of (1)

We assume c_3 is valid, i.e. (3) $A\theta_2 \vee B\theta_2 \vee C$ is true in all $M(B \cup o)$.

We now assume $\neg c_4$ is true in all $M(B \cup o)$, and prove that this results in a contradiction (the result follows from this):

(4) $\neg(A\theta_2 \vee C)$ is true in all $M(B \cup o)$

(5) $\neg A\theta_2 \wedge \neg C$ is true in all $M(B \cup o)$ because of (4)

(6) $\neg A\theta_2$ is true in all $M(B \cup o)$ because of (5)

(7) $B\theta_2$ is true in all $M(B \cup o)$ because of (3) and (4)

(8) $\neg B\theta_2$ is true in all $M(B \cup o)$ because of (2) and (6)

(7) and (8) result in a contradiction. \square

The lemma states that refinements $c_2 = c_1 \cup d$ of c_1 contain redundant information if $d \rightarrow c_1$ is true in all $M(B \cup o)$. Given a fair language \mathcal{L} we may prune c_2 as well as its refinements. To illustrate the lemma, reconsider Example 2 and assume $c_1 = \neg \text{male}(X)$ and $c_2 = \neg \text{male}(X) \vee \neg \text{gorilla}(X)$. $c_2 - c_1 \rightarrow c_1$ in all $M(B \cup o)$, i.e. $\neg \text{gorilla}(X)$ implies $\neg \text{male}(X)$, therefore c_2 contains redundant information ($\text{gorilla}(X)$), and may be pruned. Fairness requires that if clause $c_3 = \neg \text{male}(Z) \vee \neg \text{gorilla}(Z) \vee \neg \text{tall}(Z) \in \mathcal{L}$, 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 (on the observations) but θ -subsumes c_3 , will be considered.

4.3 Parallellism

Due to the monotonicity property of our induction framework, it is relatively easy to parallelize the ClausalDiscovery engine. ClausalDiscovery essentially traverses the space of clauses exhaustively and general-to-specific. This yields a search-tree in which the nodes are clauses, and there is a subtree of a clause for each refinement (under the operator ρ) of the clause. Now, due to monotonicity all subtrees of the search-tree can be processed independently of each other and therefore in parallel. The resulting algorithm is shown in Figure 3. The rest of this section may be skipped by the casual reader.

ParallelClausalDiscovery is the main function of the parallel version of the algorithm. The input parameter n determines the degree of parallellism, i.e. the maximal number of processes that will be executing concurrently. Processes exchange information through the use of the shared variable Q ⁷. For each of the n processes, this variable contains a queue equivalent to queue Q in ClausalDiscovery. Initially, all queues in Q except the one of the first process are set to empty. The queue of the first process is initialized to the top node of the hypothesis space, i.e. \square . The UNIX⁸ inspired *fork* instruction creates a new (*child*) process that will execute the call given as the single argument of *fork* concurrently with the calling (*parent*) process. ParallelClausalDiscovery calls ParaCD, n times. The *fork* instruction causes $n - 1$ of these calls to be executed concurrently with the parent process in $n - 1$ newly created processes. All results are stored in $H_1 \dots H_n$ and combined to H , which is ultimately returned as the solution.

The single input parameter p of ParaCD ranges between 1 and n , and identifies the present process. Global variable $Q(p)$ contains a queue of clauses that represents the root of the subtree to be explored by p . The outmost loop terminates the moment this queue is empty for all processes. At that moment the local solution H_p is returned and ParaCD stops. There are two more nested loops. The first one terminates either if the same condition of the outer loop is fulfilled or if the current process has received a new subtree. The body of this loop is empty but for the do-nothing-instruction *skip*. After termination of this first inner loop, $Queue$ gets the value of $Q(p)$. The second inner loop is a near copy of ClausalDiscovery. The only difference is that at the beginning of each step Q is searched for empty queues. If such an empty queue is found on position i in Q , process p cedes part of its subtree to

⁷More sophisticated systems for interprocess communication exist, but for reasons of simplicity we will continue to use the most general and basic constructs throughout.

⁸UNIXTM Trademark of Bell Laboratories

```

function ParaClausalDiscovery
  inputs :  $O$ : set of Closed observations,  $B$ : background theory,
            $\rho$ : refinement operator,  $n$  : number of processors
  outputs : Characterizing Hypothesis

   $Q(1) := \{\square\}$ 
  for all  $i \in 2 \dots n$  do  $Q(i) := \emptyset$ 
   $H_2 := \text{fork}(\text{ParaCD}(2))$ 
  ...
   $H_n := \text{fork}(\text{ParaCD}(n))$ 
   $H_1 := \text{ParaCD}(1)$ 
   $H := \cup H_i$ 
  reduce( $H$ )
  return  $H$ 
endfunction

function ParaCD
  inputs :  $p$ : name of processor,
  outputs : Partial Confirmatory Hypothesis

   $H_p := \emptyset$ 
  while not ( $\forall i \in 1 \dots n : Q(i) = \emptyset$ ) do
    while not ( $\forall i \in 1 \dots n : Q(i) = \emptyset$ ) and ( $Q(p) = \emptyset$ ) do skip
     $Queue := Q(p)$ 
    while  $Queue \neq \emptyset$  do
      for all  $i \in 1 \dots n$  do if  $Q(i) = \emptyset$  then move part of Queue to  $Q(i)$ 
      delete  $c$  from  $Queue$ 
      if  $c$  is valid on  $O$  and not prune1( $c$ )
      then add  $c$  to  $H_p$ 
      else for all  $c' \in \rho(c)$  for which not prune2( $c'$ ) do add  $c'$  to  $Queue$ 
      endif
      endwhile
     $Q(p) := \emptyset$ 
    endwhile
  return  $H_p$ 
endfunction

```

Figure 3: A parallel clausal discovery algorithm

process i by moving part of *Queue* to $Q(i)$. Which part of *Queue* is moved will depend on the search strategy chosen by the user (cf. parameter *delete* in Figure 1). An important general restriction is that the *move* instruction should not be allowed to empty *Queue*, as this might result in a loop where the same subtask is passed round forever. From the moment *Queue* contains no further candidates for refinement, $Q(p)$ is set to empty in order to inform the other processes that process p is ready to receive a new subtask, i.e. a new subtree.

In case common variables such as Q are used for interprocess communication the synchronization problem of mutual exclusion occurs. *Mutual exclusion* is concerned with ensuring that a sequence of statements, called a *critical section*, is treated as an indivisible operation that can not be executed by more than one process at the same time. In ParaCD the boxes mark two critical sections. They should prevent that two processes are simultaneously writing to $Q(i)$ or that the incomplete $Q(p)$ is copied to *Queue* while it is being written by some other process.

It is easy to see that ParallelClausalDiscovery has the same behaviour as ClausalDiscovery.

4.4 Declarative language bias

Even if we choose the search space \mathcal{L} to be finite, it is in most cases impractical to define \mathcal{L} extensionally. We then need a formalism to formulate an intensional syntactic definition of language \mathcal{L} .

The problem of making this type of syntactic bias a parameter to the learning or discovering engine has been studied extensively, especially in frameworks that use first-order clausal logic (see [Muggleton and De Raedt, 1994; Adé *et al.*, 1995] for an overview). In CLAUDIEN we use *DLAB* (Declarative LAnguage Bias), which is now available as a general machine learning system component that allows for a straightforward specification of syntactic bias. *DLAB* extends the syntactic bias of [Adé *et al.*, 1995] which in turn integrates the schemata of [Emde *et al.*, 1983; Kietz and Wrobel, 1992], and the predicate sets of [Bergadano and Gunetti, 1993; Bergadano, 1993]. When compared to Cohen’s antecedent description grammars [Cohen, 1994], *DLAB* is a special case where the definite clause grammar is fixed and hidden. This grammar takes the *DLAB* formula as its single argument. In that sense *DLAB* is a higher order formalism based on the lower order antecedent description grammar, and from a practical view point both formalisms compare as do higher and lower order programming languages.

We now present an overview of *DLAB* in two stages. First, we discuss syntax, semantics and a refinement operator for DLAB^\ominus , a subset of *DLAB*. We then extend DLAB^\ominus to full *DLAB*. A simplified implementation of *DLAB* can be found in the text itself. A more sophisticated *DLAB* PROLOG library is available by anonymous ftp access⁹.

4.4.1 DLAB^\ominus syntax

A DLAB^\ominus grammar is a set of templates to which the clauses in search space \mathcal{L} conform. We first define DLAB^\ominus grammars syntactically.

⁹This library is compatible at least with ProLog by BIM 4.0.5 and SICStus Prolog 2.1. It is available by ftp access to *ftp.cs.kuleuven.ac.be*. The relevant directory is *pub/logic – prgm/ilp/dlab*.

Definition 14 (\mathcal{DLAB}^\ominus grammar) *DGRAM* is a \mathcal{DLAB}^\ominus grammar if and only if *DGRAM* is a set of \mathcal{DLAB}^\ominus templates.

Definition 15 (\mathcal{DLAB}^\ominus template) *DTEMP* is a \mathcal{DLAB}^\ominus template if and only if *DTEMP* = $HA \leftarrow BA$, where *HA* and *BA* are \mathcal{DLAB}^\ominus atoms.

Definition 16 (\mathcal{DLAB}^\ominus atom) *DATOM* is a \mathcal{DLAB}^\ominus atom if and only if *DATOM* is an atomic formula, or *DATOM* = $Min \cdot \cdot Max : L$, with *Min* and *Max* integers such that $0 \leq Min \leq Max \leq \text{length}(L)$, and *L* is a list of \mathcal{DLAB}^\ominus atoms.

The following are a few examples of syntactically well-formed \mathcal{DLAB}^\ominus grammars:

- $\{say(Hello) \leftarrow to_world\}$
- $\{false \leftarrow 0 \cdot \cdot 2 : [male(X), female(X)]\}$
- $\{2 \cdot \cdot 2 : [a(X), b(Y)] \leftarrow 1 \cdot \cdot 2 : [c(X), 0 \cdot \cdot 1 : [d(Y)]] ,$
 $0 \cdot \cdot 1 : [n, 1 \cdot \cdot 2 : [o, 1 \cdot \cdot 1 : [p, q], r], s] \leftarrow true\}$

4.4.2 \mathcal{DLAB}^\ominus semantics

The generation of a language \mathcal{L} given a \mathcal{DLAB}^\ominus grammar then basically consists of the (recursive) selection of all subsets of *L* with length within range *Min*...*Max* from each \mathcal{DLAB}^\ominus atom $Min \cdot \cdot Max : L$ in the grammar. To simplify our definition of a generation function we here introduce (and will continue to use throughout this section) a special list notation for clauses, such that $h_1, \dots, h_n \leftarrow b_1, \dots, b_m$ will be written as $[h_1, \dots, h_n] \leftarrow [b_1, \dots, b_m]$.

The following definitions provide a generator for \mathcal{DLAB}^\ominus grammars.

Definition 17 (*dlab_generate(DGRAM)*) Let *DGRAM* be a \mathcal{DLAB}^\ominus grammar.

$$dlab_generate(DGRAM) = \{dlab1(HT) \leftarrow dlab1(BT) \mid (HT \leftarrow BT) \in DGRAM\}$$

where *dlab1(DATOM)* is a list of literals generated by the definite clause grammar [Clocksin and Mellish, 1981; Sterling and Shapiro, 1986] *dlab1*:

$$dlab1(A) \longrightarrow [A], \{A \neq Min \cdot \cdot Max : L\}. \quad (1)$$

$$dlab1(Min \cdot \cdot Max : []) \longrightarrow \{Min \leq 0\}, []. \quad (2)$$

$$dlab1(Min \cdot \cdot Max : [-|L]) \longrightarrow dlab1(Min \cdot \cdot Max : L). \quad (3)$$

$$dlab1(Min \cdot \cdot Max : [A|L]) \longrightarrow \{Max > 0\}, dlab1(A),$$

$$dlab1((Min - 1) \cdot \cdot (Max - 1) : L). \quad (4)$$

From the previous definition we can derive a formula for calculating the number of clauses in *dlab_generate(DGRAM)*, which corresponds to the size of the search space defined by a \mathcal{DLAB}^\ominus grammar *DGRAM*.

Definition 18 ($dlab_size(DGRAM)$) Let $DGRAM = \{H_1 \leftarrow B_1, \dots, H_m \leftarrow B_m\}$ be a \mathcal{DLAB}^\ominus grammar.

$$dlab_size(DGRAM) = \sum_{i=1}^m (ds(H_i) * ds(B_i))$$

$$ds(A) = 1, \text{ where } A \neq Min \cdot Max : L$$

$$ds(Min \cdot Max : [L_1, \dots, L_n]) = \sum_{k=Min}^{Max} e_k(ds(L_1), \dots, ds(L_n))$$

$$e_0(L) = 1$$

$$e_n(s_1, \dots, s_n) = \prod_{i=1}^n s_i$$

$$e_k(s_1, s_2, \dots, s_n) = e_k(s_2, \dots, s_n) + s_1 * e_{k-1}(s_2, \dots, s_n), \text{ with } k < n$$

Proof The first rule states that the size of the language defined by a \mathcal{DLAB}^\ominus grammar equals the sum of the sizes of the languages defined by its individual \mathcal{DLAB}^\ominus templates. The latter size can be found by multiplying the number of headlists and the number of bodylists covered by the head and body \mathcal{DLAB}^\ominus atoms.

A \mathcal{DLAB}^\ominus atom which is not of the form $Min \cdot Max : L$ has a coverage of exactly one, as is expressed in the second rule.

Some more intricate combinatorics underlies the third rule. Basically, we select k objects from $\{L_1, \dots, L_n\}$, for each k in range $Min \dots Max$, hence the summation $\sum_{k=Min}^{Max}$. Inside this summation we would have the standard formula $n!/k! * (n - k)!$ if our case had been an instance of the prototypical problem of finding all combinations, without replacement, of k marbles out of an urn with n marbles. This formula does not apply due to the fact that we rather have n urns ($\{L_1, \dots, L_n\}$) with one or more marbles ($ds(L_i) \geq 1$), and only combinations that use at most one marble from each urn should be counted. Therefore we need $e_k(s_1, \dots, s_n)$, where e_k is the elementary symmetric function [MacDonald, 1979] of degree k and the s_i are the numbers of marbles in each urn. The first base case of this recursive function accounts for the fact that there is only one way to select 0 objects. In the second base case, where $k = n$, one has to take an object from each urn. As for each urn there are s_i choices, the number of combinations equals the product of all s_i . The final recursive case applies if $k < n$. It is an addition of two terms, one for each possible operation on urn 1 (represented by s_1). Either we skip this urn, and then we still have to select k elements from urns 2 to n . The number of such combinations is given by $e_k(s_2, \dots, s_n)$. Or else we do take a marble from the first urn. We then have to multiply s_1 , the choices for the first urn, with $e_{k-1}(s_2, \dots, s_n)$, the number of $k - 1$ order combinations of elements from urns 2 to n . \square

A few illustrations of the definitions above should give a first idea of the expressive power of the relatively simple \mathcal{DLAB}^\ominus formalism. Given a \mathcal{DLAB}^\ominus atom $Min \cdot Max : L$, we can distinguish the following cases of special interest:

- **all subsets:** $Min = 0, Max = length(L)$

$$DGRAM_1 = \{0 \cdot 1 : [gorilla(X)] \leftarrow 0 \cdot 2 : [female(X), male(X)]\}$$

$$dlab_generate(DGRAM_1) = \left\{ \begin{array}{l} [] \leftarrow [] \\ [] \leftarrow [male(X)] \\ [] \leftarrow [female(X)] \\ [] \leftarrow [female(X), male(X)] \\ [gorilla(X)] \leftarrow [] \\ [gorilla(X)] \leftarrow [male(X)] \\ [gorilla(X)] \leftarrow [female(X)] \\ [gorilla(X)] \leftarrow [female(X), male(X)] \end{array} \right.$$

$$dlab_size(DGRAM_1) = 8$$

- **all non-empty subsets:** $Min = 1, Max = length(L)$

$$DGRAM_2 = \{0 \cdot 1 : [gorilla(X)] \leftarrow 1 \cdot 2 : [female(X), male(X)]\}$$

$$dlab_generate(DGRAM_2) = \left\{ \begin{array}{l} [] \leftarrow [male(X)] \\ [] \leftarrow [female(X)] \\ [] \leftarrow [female(X), male(X)] \\ [gorilla(X)] \leftarrow [male(X)] \\ [gorilla(X)] \leftarrow [female(X)] \\ [gorilla(X)] \leftarrow [female(X), male(X)] \end{array} \right.$$

$$dlab_size(DGRAM_2) = 6$$

- **exclusive or:** $Min = Max = 1$

$$DGRAM_3 = \{0 \cdot 1 : [gorilla(X)] \leftarrow 1 \cdot 1 : [female(X), male(X)]\}$$

$$dlab_generate(DGRAM_3) = \left\{ \begin{array}{l} [] \leftarrow [male(X)] \\ [] \leftarrow [female(X)] \\ [gorilla(X)] \leftarrow [male(X)] \\ [gorilla(X)] \leftarrow [female(X)] \end{array} \right.$$

$$dlab_size(DGRAM_3) = 4$$

- **combined occurrence:** $Min = Max = length(L)$

$$DGRAM_4 = \{gorilla(X) \leftarrow \begin{array}{l} 0 \cdot 2 : [2 \cdot 2 : [female(X), is_daughter(X)], \\ 2 \cdot 2 : [male(X), is_son(X)] \\] \end{array} \}$$

$$dlab_generate(DGRAM_4) = \begin{cases} [gorilla(X)] \leftarrow [] \\ [gorilla(X)] \leftarrow [male(X), is_son(X)] \\ [gorilla(X)] \leftarrow [female(X), is_daughter(X)] \\ [gorilla(X)] \leftarrow [female(X), is_daughter(X), \\ \quad male(X), is_son(X)] \end{cases}$$

$$dlab_size(DGRAM_4) = 4$$

The fact that nesting of \mathcal{DLAB}^\ominus atoms to arbitrary depth is allowed can for instance be exploited to encode taxonomies, such that each atomic formula necessarily co-occurs with all its ancestors and never combines with other nodes. Thus for instance, $DGRAM_5$ encodes the taxonomy for suits of playing cards:

$$DGRAM_5 = \{ok(C) \leftarrow \begin{array}{l} 2 \cdot 2 : [suit(C), \\ \quad 0 \cdot 1 : [2 \cdot 2 : [red(C), \\ \quad \quad 0 \cdot 1 : [hearts(C), diamonds(C)] \\ \quad \quad]], \\ \quad 2 \cdot 2 : [black(C), \\ \quad \quad 0 \cdot 1 : [clubs(C), spades(C)] \\ \quad \quad] \\ \quad] \end{array} \}$$

$$dlab_generate(DGRAM_5) = \begin{cases} [ok(C)] \leftarrow [suit(C)] \\ [ok(C)] \leftarrow [suit(C), red(C)] \\ [ok(C)] \leftarrow [suit(C), red(C), hearts(C)] \\ [ok(C)] \leftarrow [suit(C), red(C), diamonds(C)] \\ [ok(C)] \leftarrow [suit(C), black(C)] \\ [ok(C)] \leftarrow [suit(C), black(C), clubs(C)] \\ [ok(C)] \leftarrow [suit(C), black(C), spades(C)] \end{cases}$$

$$dlab_size(DGRAM_5) = 7$$

4.4.3 A \mathcal{DLAB}^\ominus refinement operator

This subsection shows how a refinement operator for a \mathcal{DLAB} language can be obtained from the \mathcal{DLAB} grammar. Furthermore, it touches upon some of the key implementation aspects of the $\mathcal{CLAUDIEN}$ engine. This rather technical subsection can be safely skipped by the casual reader not interested in this topic.

A refinement operator ρ (cf. Definition 10) for \mathcal{DLAB}^\ominus is based on the observation that clauses c in $dlab_generate(DGRAM)$ are defined by a sequence of subset selections from \mathcal{DLAB}^\ominus atoms occurring in $DGRAM$. If we enlarge one of these subsets then the clause $c' \supseteq c$ defined by the new sequence is a specialization of c under θ -subsumption. If we somehow enlarge one subset in a minimal way, then c' will be a refinement, i.e. a maximally

general specialization of c^{10} . To implement this idea we adapt the definite clause grammar $dlab1$ in Definition 17 in three steps.

First, in order to formalize the above notion of a sequence of subset selections, we add to $dlab1$ an extra argument we will refer to as the \mathcal{DLAB}^\ominus path. The \mathcal{DLAB}^\ominus path is meant to keep track of applications of Rules (3) and (4) in $dlab1$. The application of these rules determines whether the first \mathcal{DLAB}^\ominus atom in list L of $Min \cdot \cdot Max : L$ is either skipped (Rule (3)) or included in the subset (Rule (4)).

Definition 19 (\mathcal{DLAB}^\ominus path) Let $DATOM$ be a \mathcal{DLAB}^\ominus atom, and C a list of literals generated by $dlab1(DATOM)$. $DPATH$ is a \mathcal{DLAB}^\ominus path of C with regard to $DATOM$ if and only if

- $DATOM \neq Min \cdot \cdot Max : L$ and $DPATH = DATOM$ or
- $DATOM = Min \cdot \cdot Max : [L_1, \dots, L_n]$ and $DPATH = [P_1, \dots, P_n]$, with, for each $P_i \in DPATH$,
 - $P_i = *$ and L_i is excluded during generation of C (application of Rule (3)/(7)), or
 - P_i is the \mathcal{DLAB}^\ominus path of C with regard to \mathcal{DLAB}^\ominus atom L_i and L_i is included during generation of C (application of Rule (4)/(8))

For instance,

$DATOM = 0 \cdot \cdot 2 : [gorilla(X), 1 \cdot \cdot 1 : [female(X), male(X)]]$	
$C = dlab1(DATOM)$	\mathcal{DLAB}^\ominus path of C with regard to $DATOM$
$[]$	$[*, *]$
$[male(X)]$	$[*, [*, male(X)]]$
$[female(X)]$	$[*, [female(X), *]]$
$[gorilla(X)]$	$[gorilla(X), *]$
$[gorilla(X), male(X)]$	$[gorilla(X), [*, male(X)]]$
$[gorilla(X), female(X)]$	$[gorilla(X), [female(X), *]]$

The following is an adaptation of $dlab1$, with the \mathcal{DLAB}^\ominus path in the second argument position.

$$dlab2(A, A) \longrightarrow [A], \{A \neq Min \cdot \cdot Max : L\}. \quad (5)$$

$$dlab2(Min \cdot \cdot Max : [], []) \longrightarrow \{Min \leq 0\}, []. \quad (6)$$

$$dlab2(Min \cdot \cdot Max : [_|L], [*|Y]) \longrightarrow dlab2(Min \cdot \cdot Max : L, Y). \quad (7)$$

$$dlab2(Min \cdot \cdot Max : [A|L], [X|Y]) \longrightarrow \{Max > 0\}, dlab2(A, X), \\ dlab2((Min - 1) \cdot \cdot (Max - 1) : L, Y). \quad (8)$$

In a second step, we can use the \mathcal{DLAB}^\ominus path DP of a list of literals C to generate supersets of C . Every $*$ in DP marks an occasion for extending C . In terms of Definition 19: we have to locate a $P_i = *$ in DP indicating the corresponding \mathcal{DLAB}^\ominus atom L_i is excluded

¹⁰Depending on the \mathcal{DLAB}^\ominus grammar, this refinement (under θ -subsumption) can be proper or not.

during generation of C , and then include L_i during generation of supersets C' of C . Definite clause grammar $dlabs$ does that, and moreover returns the \mathcal{DLAB}^\ominus path DP' of C' in the third argument position.

$$dlabs(_ \cdot \cdot Max : [], [], []) \longrightarrow []. \quad (9)$$

$$dlabs(_ \cdot \cdot Max : [A|L], [*|Y], [X|Z]) \longrightarrow \{Max > 0\}, dlab2(A, X), \\ dlabs(_ \cdot \cdot (Max - 1) : L, Y, Z). \quad (10)$$

$$dlabs(_ \cdot \cdot Max : [_|L], [*|Y], [*|Z]) \longrightarrow dlabs(_ \cdot \cdot Max : L, Y, Z). \quad (11)$$

$$dlabs(_ \cdot \cdot Max : [A|L], [P|Y], [Q|Z]) \longrightarrow \{P \neq *, Max > 0\}, dlabs(A, P, Q), \\ dlabs(_ \cdot \cdot (Max - 1) : L, Y, Z). \quad (12)$$

$$dlabs(_ \cdot \cdot Max : [A|L], [X|Y], [X|Z]) \longrightarrow \{X \neq *, Max > 0\}, dlab2(A, X), \\ dlabs(_ \cdot \cdot (Max - 1) : L, Y, Z). \quad (13)$$

Notice how in Rule (10) of $dlabs$ the previously excluded A (cf. the $*$ in Arg2) is now included with the call of $dlab2(A, X)$. For instance,

$DATOM = 0 \cdot \cdot 3 : [gorilla(X), female(X), male(X)]$ $C = [female(X)]$ $DP = [*, female(X), *]$	
$C' = dlabs(DATOM, DP, DP')$	DP'
$[gorilla(X), female(X), male(X)]$ $[gorilla(X), female(X)]$ $[female(X), male(X)]$ $[female(X)]$	$[gorilla(X), female(X), male(X)]$ $[gorilla(X), female(X), *]$ $[*, female(X), male(X)]$ $[*, female(X), *]$

The rules in $dlabs$ can be used to find specializations c' of c . As we want our refinement operator to generate only maximally general specializations of c , a final adaptation of $dlabs$ is required such that it will generate only smallest supersets of C . Roughly stated, exactly one $*$ in the \mathcal{DLAB}^\ominus path DP of a list of literals C should be expanded, and then only in a minimal way. The first requirement, again in terms of Definition 19, says that we should locate exactly one $P_i = *$ in DP , and then include L_i during generation of supersets of C . The second requirement says that the inclusion of L_i should be minimal in the sense that the corresponding \mathcal{DLAB}^\ominus path P'_i should contain the maximally allowed number of $*$'s. For this we need a modified version of $dlab2$, that, given a \mathcal{DLAB}^\ominus atom $Min \cdot \cdot Max : L$, will only generate subsets of length Min . The first requirement is realized in $dlabr$ by eliminating some recursive calls, the second by initialization the newly included \mathcal{DLAB}^\ominus atom A with $dlabi$ instead of $dlab2$.

$$dlabr(Min \cdot \cdot Max : [A|L], [*|Y], [X|Y]) \longrightarrow \{not(dlab_optimal, member(E, Y), E \neq *)\}, \\ \{Max > 0\}, dlabi(A, X), \\ dlab2((Min - 1) \cdot \cdot (Max - 1) : L, Y). \quad (14)$$

$$dlabr(Min \cdot \cdot Max : [_|L], [*|Y], [*|Z]) \longrightarrow dlabr(Min \cdot \cdot Max : L, Y, Z). \quad (15)$$

$$dlabr(Min \cdot \cdot Max : [A|L], [X|Z], [Y|Z]) \longrightarrow \{X \neq *, Max > 0\}, dlabr(A, X, Y), \\ dlab2((Min - 1) \cdot \cdot (Max - 1) : L, Z). \quad (16)$$

$$dlabr(Min \cdot \cdot Max : [A|L], [X|Y], [X|Z]) \longrightarrow \{X \neq *, Max > 0\}, dlab2(A, X), \\ dlabr((Min - 1) \cdot \cdot (Max - 1) : L, Y, Z). \quad (17)$$

$$dlabi(A, A) \longrightarrow [A], \{not(A = Min \cdot \cdot Max : L)\}. \quad (18)$$

$$dlabi(0 \cdot \cdot _ : [], []) \longrightarrow []. \quad (19)$$

$$dlabi(Min \cdot \cdot _ : [A|L], [X|Y]) \longrightarrow dlabi(A, X), \quad (20)$$

$$dlabi((Min - 1) \cdot \cdot _ : L, Y).$$

$$dlabi(Min \cdot \cdot _ : [_|L], [*|Y]) \longrightarrow dlabi(Min \cdot \cdot _ : L, Y). \quad (21)$$

Notice that Rule 14 of *dlabr* contains an extra initial condition:

$$not(dlab_optimal, member(E, Y), E \neq *)$$

A call to *dlab_optimal* should succeed, if we want the refinement operator to be optimal (cf. Definition 11), and fail otherwise.

The extra condition ensures that when working in optimal mode, the refinement operator will never expand *'s to the left of already expanded *'s. For instance,

$DATOM = 0 \cdot \cdot 3 : [gorilla(X), female(X), male(X)]$ $C = [female(X)]$ $DP = [*, female(X), *]$		
<i>dlab_optimal</i>	$C' = dlabr(DATOM, DP, DP')$	DP'
false	$[gorilla(X), female(X)]$ $[female(X), male(X)]$	$[gorilla(X), female(X), *]$ $[*, female(X), male(X)]$
true	$[female(X), male(X)]$	$[*, female(X), male(X)]$

To further enforce optimality we have to make sure refinement of the head of a clause blocks all future refinements of the body, or vice-versa¹¹.

We can now formulate the definition of a \mathcal{DLAB}^\ominus refinement operator based on the twelve definite clause grammar rules of *dlabr*, *dlabi*, and *dlab2*.

Definition 20 (*dlab_refine*(*DINFO*, *c*)) *Given*

- \mathcal{DLAB}^\ominus template $HA \leftarrow BA$,
- clause $c = H \leftarrow B$, with $c \in dlab_generate(\{HA \leftarrow BA\})$
- HP a \mathcal{DLAB}^\ominus path of H with regard to HA ,
- BP a \mathcal{DLAB}^\ominus path of B with regard to BA ,
- $DINFO = (HA, HP, BA, BP)$,

If *dlab_optimal* = false

$$dlab_refine(DINFO, c) = dlab_refh(DINFO, c) \cup dlab_refb(DINFO, c)$$

If *dlab_optimal* = true

$$dlab_refine(DINFO, c) = dlab_refh((HA, HP, [], []), c) \cup dlab_refb(DINFO, c)$$

¹¹In fact, both measures merely prevent the same couple of \mathcal{DLAB}^\ominus paths (one for the head, one for the body) from being generated more than once. In case the list of body- or headliterals of a single clause corresponds to $n > 1$ \mathcal{DLAB}^\ominus paths, e.g. $[male(X)]$ given \mathcal{DLAB}^\ominus atom $1 \cdot \cdot 1 : [male(X), male(X), male(X)]$ ($n = 3$), \mathcal{DLAB}^\ominus is likely to generate this clause n times. Part of the responsibility for optimality is thus left to the \mathcal{DLAB}^\ominus user.

$$dlab_refh((HA, HP, BA, BP), H \leftarrow B) = \\ \{((HA, HP', BA, BP), H' \leftarrow B) \mid H' = dlabr(HA, HP, HP')\}$$

$$dlab_refb((HA, HP, BA, BP), H \leftarrow B) = \\ \{((HA, HP, BA, BP'), H \leftarrow B') \mid B' = dlabr(BA, BP, BP')\}$$

An initialisation function that returns the most general clauses in \mathcal{L} completes the \mathcal{DLAB}^\ominus refinement operator:

Definition 21 (dlab_initialize(DGRAM)) *Let DGRAM be a \mathcal{DLAB}^\ominus grammar, then the following function returns the top nodes in the refinement lattice:*

$$dlab_initialize(DGRAM) = \{dlab_refh(dlab_refb(DINFO, \square)) \mid \\ (HA \leftarrow BA) \in DGRAM, \\ DINFO = (0 \cdot 1 : [HA], [*], 0 \cdot 1 : [BA], [*])\}$$

We are now ready to instantiate the refinement operator in the ClausalDiscovery algorithm (see Figure 1) to \mathcal{DLAB}^\ominus , with $dlab_optimal = true$. The basic idea is to store elements of type $(DINFO, c)$ in queue Q . As in practise queue Q often grows to a size above 10^5 , the explicit storage of nodes $(DINFO, c)$ might quickly exhaust memory resources. The \mathcal{DLAB}^\ominus formalism however allows for a straightforward optimization, where only the \mathcal{DLAB}^\ominus paths are stored in Q together with the a pointer to the \mathcal{DLAB}^\ominus template. Corresponding clauses can then be recovered using $dlab2$ ¹². We then use $dlab_initialize(DGRAM)$ to initialize Q to the most general element(s) in \mathcal{L} , and $dlab_refine(DINFO, c)$ to calculate refinements of the elements we retrieve from Q .

4.4.4 Extended \mathcal{DLAB}^\ominus : \mathcal{DLAB}

Although \mathcal{DLAB}^\ominus grammars as they are introduced in the previous section are purely declarative, their readability rapidly deteriorates as their complexity rises. In an extended version \mathcal{DLAB} mainly two features have been added to alleviate this problem: second order predicate variables, and subsets on the term level. We will now define \mathcal{DLAB} and conversion functions that map grammars from the \mathcal{DLAB} to the \mathcal{DLAB}^\ominus format, such that the refinement operator defined above will also work with the enriched formalism.

Definition 22 (\mathcal{DLAB} grammar) *DGRAM is a \mathcal{DLAB} grammar if and only if $DGRAM = (DTEMP S, DVARS)$, where $DTEMP S$ is a set of \mathcal{DLAB} templates, and $DVARS$ is a set of \mathcal{DLAB} variables.*

Definition 23 (\mathcal{DLAB} variable) *DVAR is a \mathcal{DLAB} variable if and only if $DVAR = dlab_variable(P_0, Min \cdot Max, [P_1, \dots, P_n])$, where Min and Max are integers with $0 \leq Min \leq Max \leq n$, and for all $P_i \in \{P_0, \dots, P_n\}$, P_i is a predicate symbol or a function symbol.*

¹²In a more sophisticated version of \mathcal{DLAB}^\ominus the \mathcal{DLAB}^\ominus paths are flat lists of symbols 0, 1, *, such that groups of 4 elements in the path can be further compressed to one 81-ary digit.

```

function ConvertToDLAB⊖1
  inputs : (DTEMPS, DVARS): DLAB grammar
  outputs : equivalent DLAB grammar (DTEMPS' ,  $\emptyset$ )

DTEMPS' := DTEMPS
CONTINUE := true
while CONTINUE do
  find  $P(t_1, \dots, t_n)$  in DTEMPS' , with  $0 \leq n$ ,
    for which  $dlab\_variable(P, Min \cdot \cdot Max, [P_1, \dots, P_m]) \in DVARS
  if found
  then replace  $P(t_1, \dots, t_n)$  in DTEMPS' with
     $Min \cdot \cdot Max : [P_1(t_1, \dots, t_n), \dots, P_m(t_1, \dots, t_n)]$ 
  else CONTINUE := false
endwhile
endfunction$ 
```

Figure 4: Removal of DLAB variables from DLAB grammars

Definition 24 (DLAB template) *DTEMP* is a DLAB template if and only if $DTEMP = HA \leftarrow BA$, where *HA* and *BA* are DLAB atoms.

Definition 25 (DLAB atom) *DATOM* is a DLAB atom if and only if

- $DATOM = P(t_1, \dots, t_n)$, where *P* is a predicate symbol t_1, \dots, t_n ($0 \leq n$) is a sequence of *n* DLAB terms, or
- $DATOM = Min \cdot \cdot Max : L$, where *Min* and *Max* are integers with $0 \leq Min \leq Max \leq length(L)$, and *L* is a list of DLAB atoms.

Definition 26 (DLAB term) *DTERM* is a DLAB term if and only if

- *DTERM* is a variable symbol, or
- $DTERM = F(t_1, \dots, t_n)$, where *F* is a function symbol and t_1, \dots, t_n ($0 \leq n$) is a sequence of *n* DLAB terms, or
- $DTERM = Min \cdot \cdot Max : L$, where *Min* and *Max* are integers with $0 \leq Min \leq Max \leq length(L)$, and *L* is a list of DLAB terms.

The function *convert_to_dlab_minus*(*DGRAM*) required for converting grammars from DLAB to the DLAB[⊖] format is defined with two helpfunctions: one to remove (and expand) the DLAB variables (see Figure 4), and one to move the subsets on the termlevel outside to the atomlevel (see Figure 5).

Definition 27 (convert_to_dlab_minus(DGRAM)) Let *DGRAM* be a DLAB grammar and $ConvertToDLAB^{\ominus 2}(ConvertToDLAB^{\ominus 1}(DGRAM)) = (DGRAM^{\ominus}, \emptyset)$, then *DGRAM*[⊖] is the DLAB[⊖] grammar equivalent to *DGRAM*.

```

function ConvertToDLAB⊖2
  inputs : (DTEMPS, DVARS): DLAB grammar
  outputs : equivalent DLAB grammar without subsets on termlevel

DTEMPS' := DTEMPS
CONTINUE := true
while CONTINUE do
  find  $P(t_1, \dots, t_i, \text{Min} \cdot \text{Max} : [L_1, \dots, L_n], t_{i+2}, \dots, t_m)$  in DTEMPS',
    with for all  $t_j \in \{t_1, \dots, t_i\}, t_j \neq A \cdot B : C$ 
  if found
  then replace  $P(t_1, \dots, t_i, \text{Min} \cdot \text{Max} : [L_1, \dots, L_n], t_{i+2}, \dots, t_m)$  in DTEMPS' with
     $\text{Min} \cdot \text{Max} : [P(t_1, \dots, t_i, L_1, t_{i+2}, \dots, t_m), \dots, P(t_1, \dots, t_i, L_n, t_{i+2}, \dots, t_m)]$ 
  else CONTINUE := false
endwhile
endfunction

```

Figure 5: Removal of subsets on termlevel from *DLAB* grammars

For a demonstration of the power of *DLAB*[⊖] and *DLAB*¹³ we refer to the experiments in Section 5.

4.5 Quantifying Validity

There are at least three reasons why the *logical* validity requirement should be quantified and sometimes relaxed. First, when coping with real data, it is an illusion to find rules that are valid on all of the observations. The same situation arises in explanatory induction when trying to discriminate two classes of observations. As very often complete and consistent hypotheses do not exist, explanatory induction allows to relax the completeness and consistency requirements. It is therefore also of practical interest to see how the validity requirement of characteristic induction from closed observations can be relaxed. This corresponds to relaxing the q in Mannila's definition. Secondly, a quantified notion of validity will also be useful to label the induced clauses, and to rank them according to validity. Such a ranking is essential for expert evaluation and post-processing of discovered rules. Thirdly, quantified notions of validity may turn out useful for heuristically searching the space, cf. Section 4.6.

There are two natural ways to quantify validity. For the first one we introduce the concept of non-trivial observations. The set $O' \subset O$ of non-trivial observations contains all observations for which clause c is non-trivial (cf. Definition 5). We can then relax the condition that clauses in hypotheses are valid on *all* observations, and rather require validity on a certain percentage of all non-trivial observations. This can be realized by setting $GA(c)$

¹³As a minor additional extension we will also allow *DLAB* atoms of the type $\text{Min} \cdot \text{len} : L$ or $\text{len} \cdot \text{len} : L$, where len is a constant symbol that abbreviates $\text{length}(L)$.

larger than a fixed percentage.

Definition 28 (Global Accuracy) *Let c be a clause, let O' be the non-trivial observations for c , let $pg(c)$ be the number of observations in O' which are a model for c , let $ng(c)$ be the number of observations in O' which are not a model for c . Then $GA(c)$, the global accuracy of the clause c , is $pg(c)/(pg(c) + ng(c))$.*

Global accuracy still requires that the clause is completely true on a number of observations. When the observations are not completely closed but only partly, even global accuracy will be hard to obtain. Furthermore, there is the special case of the framework, where only a single observation is taken into account. This special case is important in a data mining context, as one often deals with a single interpretation (in which various observations are mixed). Local accuracy, which measures the degree to which a clause is true in an interpretation may offer a solution in this case. Local accuracy employs the notions of positive and negative substitutions.

We first introduce the notions of positive and negative substitutions of a clause.

Definition 29 (Positive and Negative Substitutions) *θ is a positive substitution for a clause $p_1, \dots, p_m \leftarrow q_1, \dots, q_n$ with $m > 0$, background theory B and observations O , if and only if there exists a substitution σ 1) $(p_1, \dots, p_m \leftarrow q_1, \dots, q_n)\theta\sigma$ is ground, 2) there exists an observation $o_i \in O$ such that (a) $(q_1 \wedge \dots \wedge q_n)\theta$ is true and ground in $M(B \cup o_i)$, and (b) $(p_1 \vee \dots \vee p_m)\theta\sigma$ is true in $M(B \cup o_i)$. θ is a negative substitution if and only if it satisfies (1) and (2a) and does not satisfy (2b).*

The σ in the definition is needed only when the clause is not range-restricted. From a practical point of view, there are often problems when merely counting substitutions because there is no direct correspondence guaranteed between what is being counted (substitutions) and the entities the clause deals with (e.g. birds, or meshes, or molecules, ...). Secondly, the above definition will result in problems when applying it to denials (i.e. clauses of the form $\leftarrow q_1, \dots, q_n$). Therefore it is often convenient to transform a clause

$$p_1, \dots, p_m \leftarrow q_1, \dots, q_n$$

into the following logically equivalent form

$$p_1, \dots, p_m, \neg q_{i+1}, \dots, \neg q_n \leftarrow q_1, \dots, q_i$$

before constructing positive and negative substitutions. The positive and negative substitutions of the two clauses will not necessarily be the same. However, by appropriately choosing the literals q_1, \dots, q_i it is possible that meaningful entities are counted. In the *CLAUDIEN* implementation, the user is offered the possibility of specifying which literals to consider in the body of the clause and which ones in the head, when considering positive and negative substitutions.

By now we can define local accuracy.

Definition 30 (Local Accuracy) *Let c be a clause, let O be the observations considered, let $pl(c)$ be the number of positive substitutions for c , let $nl(c)$ be the number of negative substitutions for c . Then $LA(c)$, the local accuracy of the clause c , is $pl(c)/(nl(c) + pl(c))$.*

Again, validity can be relaxed by setting $LA(c)$ larger than a fixed percentage.

In data mining, one often labels the induced rules with information indicating accuracy of the rule and in how many cases it applies, i.e. the coverage. The above notions of accuracy are useful as an accuracy label of clauses. The following notions of global and local coverage will be used as coverage labels of clauses.

Definition 31 (*Global Coverage*) *Let O' be the non-trivial observations for c , let $pg(c)$ and $ng(c)$ be computed w.r.t. the observations O' . Then the global coverage of a clause $GC(c) = pg(c) + ng(c)$.*

The reason for restricting the attention to those observations for which the clause is non-trivial is that otherwise all clauses will have a global coverage equal to the number of observations. When applying global coverage to valid denials, the coverage will be 0. Therefore, in that case one should first apply the clause transformation introduced above in the discussion following Definition 29.

Definition 32 (*Local Coverage*) *The local coverage of a clause $LC(c) = pl(c) + nl(c)$.*

4.6 Heuristics

Explanatory inductive approaches employ various types of heuristics to guide the search towards those clauses that best discriminate the positive from the negative examples, to characterize the positive data, or to prune clauses from the search space. Various heuristics have been proposed, e.g. information content [Quinlan, 1990], minimal length description [Srinivasan *et al.*, 1992], accuracy estimates [Lavrač and Džeroski, 1994], etc.

Our induction framework can easily adapt these heuristics using the measures of validity defined in the previous subsection. More specifically, whereas explanatory induction heuristics are based on the proportions of positive and negative examples, clausal discovery can use the notions of positive and negative substitutions pl and nl , or alternatively, the number of positive and negative observations pg and ng . Given a clause c , a set of observations O , and a background theory, one can now basically employ all favourite heuristics. One basically has to substitute our numbers in the well-known formulae. This procedure works for evaluating clauses as such, i.e. without taking into account the way the clause was obtained, as well as for evaluating refinements of clauses, i.e. taking into account the ancestors in a refinement graph. An example of the first type of heuristic is accuracy, and of the second type of heuristic, entropy as applied in FOIL [Quinlan, 1990]. Many other heuristics are known in the literature, for an overview see [Lavrač and Džeroski, 1994].

As clausal discovery aims at a maximally general hypothesis, and the number of clauses in such a maximally general hypothesis may be very large, characterizing induction procedures should try to discover as many interesting clauses as possible using a limited amount of resources. Indeed, as resources are always limited (one cannot search forever), clausal discovery heuristics should employ heuristics of the first type, focusing on the most interesting clauses first. Using heuristics and limited resources (whether time or space), certain unpromising parts of the search space may not be considered. This leads to the view that characterizing induction procedures should be *any time* algorithms, i.e. algorithms that are

able to find approximate solutions in any time, and improve upon those (by discovering more clauses) when more resources are available.

In the experiments with the *CLAUDIEN* system we will mainly employ the following heuristic (based on the minimal description length principle): $p/(l+n)$ where p accounts for the positive substitutions or interpretations, n for the negative ones, and l is the clause length, computed as the number of literals in the clause tested. The heuristic is then combined with the local or global measures provided earlier. It is merely used to order the clauses on the queue, implementing an any time algorithm. Though the heuristic works fine in practice, it is unclear whether it is the most adequate one.

5 Applications of Clausal Discovery

The distinction between characteristic and discriminating induction discussed in Section 3 cascades to the level of the presentation of experimental results. For discriminating learners there is a standard two-phased assessment method in which classification rules learnt in a training stage are tested on (unseen) data. The quality of the system is typically associated with the percentage of successful class predictions. The domain of clausal discovery lacks such a clear cut evaluation criterion. The main goal is to discover *interesting* properties, but *interestingness* is in general hard to quantify, subjective and dated. Even worse, contrary to classification accuracy, which is based on elementary statistics, it can only be judged upon by an expert in the application domain.

An alternative evaluation criterion for discovery systems is then based on the iterative nature of the knowledge discovery process. Feedback from the domain expert will often trigger new, slightly altered experiments. Discovery systems that are highly tunable and versatile are better prepared to take this kind of feedback into account, and thus are *more likely* to produce interesting output in the end. Our aim in this section is then to give a flavour of the tunability and versatility of *CLAUDIEN*. We will demonstrate how *CLAUDIEN* can solve different discovery tasks, and how the system can be tuned to discover different types of rules in the same dataset. All tests were done on a SPARCserver1000.

5.1 Clausal discovery for data mining

One of the popular subjects in the field of knowledge discovery in databases is to induce large sets of rules of a particular type or syntax, cf. Mannila's definition of data mining in Section 3.2.3. The types of rules considered include: functional and multivalued dependencies (see e.g. [Flach, 1993; Savnik and Flach, 1993; Kantola *et al.*, 1992]), determinations (see e.g. [Schlimmer, 1991; Shen, 1992]), association rules (cf. [Agrawal *et al.*, 1993]), and strong rules (cf. [Piatetsky-Shapiro, 1991]). Various special purpose algorithms have been developed to handle the different types of rules. However, it turns out that because of the expressiveness of first order logic and the *DLAB* formalism of *CLAUDIEN*, that many of the tasks performed by these special purpose algorithms can be reformulated in terms of the *CLAUDIEN* framework. As a consequence, the task performed by these algorithms is a special case of that performed by *CLAUDIEN*.

Let us first provide evidence for this claim, and then discuss its implications and restrictions.

We start by showing how *CLAUDIEN* can induce functional and multi-valued dependencies on an example that is due to Flach [Flach, 1993]. We ran *CLAUDIEN* on the following data from Flach (the term *train(From,Hour,Min,To)* denotes that there is a train from *From* to *To* at time *Hour,Min*):

```

train(utrecht,8,8,den-bosch)      train(tilburg,8,10,tilburg)
train(maastricht,8,10,weert)      train(utrecht,8,25,den-bosch)
train(utrecht,9,8,den-bosch)      train(tilburg,9,10,tilburg)
train(maastricht,9,10,weert)      train(utrecht,9,25,den-bosch)
train(utrecht,8,13,eindhoven-bkln) train(tilburg,8,17,eindhoven-bkln)
train(utrecht,8,43,eindhoven-bkln) train(tilburg,8,47,eindhoven-bkln)
train(utrecht,9,13,eindhoven-bkln) train(tilburg,9,17,eindhoven-bkln)
train(utrecht,9,43,eindhoven-bkln) train(tilburg,9,47,eindhoven-bkln)
train(utrecht,8,31,utrecht)

```

using *DLAB* grammar (*train_temps*, \emptyset):

```

train_temps = {1-1 : [From1 = From2, Hour1 = Hour2, Min1 = Min2, To1 = To2]
  <--
    len-len : [train(From1,Hour1,Min1,To1,Plat1),
               train(From2,Hour2,Min2,To2,Plat2),
               0-len:[From1 = From2, Hour1 = Hour2,
                     Min1 = Min2, To1 = To2]
    ]
}

```

CLAUDIEN found (as Flach's *INDEX*) the following two dependencies:

```

From1 = From2 <-- train(From1,Hour1,Min1,To1),train(From2,Hour2,Min2,To2),
                  To1=To2,Min1=Min2
From1 = From2 <-- train(From1,Hour1,Min1,To1),train(From2,Hour2,Min2,To2),
                  From1=From2,Min1=Min2

```

It is straightforward to write *DLAB* statements that would find only determinations of the form $P(X, Y) \leftarrow Q(X, Z), R(Z, Y)$ (as [Shen, 1992]), determinations as [Schlimmer, 1991] and multivalued dependencies as in [Flach, 1993].

Very popular in the data mining literature are association rules. Association rules are defined over a single relation composed of a set of attributes R over the binary domain $\{0, 1\}$. An association rule is then of the form $X \Rightarrow Y$ where $X \subset R$ and $Y \subset (R - X)$. Typically, one is interested in all association rules c for which $LA(c) > \sigma$ and $LC(c) > \gamma$, for a certain threshold. Using local validity and the following type of *DLAB* declaration, *CLAUDIEN* would also solve the problem of finding association rules. The *DLAB* declaration (*assoc_temps*, *assoc_vars*) assumes that the relation under consideration is r with arity n , '=' denotes unification, and further that each attribute can have only two values: 0 and 1. The statement can be trivially generalized when an attribute can have more or other values.

```

assoc_temps = {
  {(X1, ..., Xn) = (Y1, ..., Yn)
  <--
  len-len:
    [r(X1, ..., Xn),
     1-1:[len-len:[Y1 = bit,
                  0-len:[1-1:[X2,Y2],1-1:[X3,Y3],...,1-1:[Xn,Yn]] = bit
                    ]
     len-len:[Y2 = bit,
              0-len:[1-1:[X1,Y1],1-1:[X3,Y3],...,1-1:[Xn,Yn]] = bit
              ]
     ...
     len-len:[Yn = bit,
              0-len:[1-1:[X1,Y1],1-1:[X2,Y3],...,1-1:[Xn-1,Yn-1]] = bit
              ]
    ]
  }

```

```

assoc_vars = {dlab_variable(bit, 1-1, [0,1])}

```

The *DLAB* statement will allow at most one literal per attribute in the body of the clause. If the literal is of the form $X=value$, then it occurs in the X part of the association rule $X \Rightarrow Y$, otherwise in the Y part. A clause generated by this *DLAB* grammar could be e.g.

$$(X1, X2, X3, X4) = (Y1, Y2, Y3, Y4) \leftarrow r(X1, X2, X3, X4), X1 = 0, Y2 = 1, Y4 = 0$$

denoting the association rule $X1 = 0 \Rightarrow Y2 = 1 \wedge Y4 = 0$.

Strong rules can be defined in a similar way. Facilities offered by *CLAUDIEN* to prune potentially large sets of association rules include:

- increase the $LA(c)$ threshold
- increase the $LC(c)$ threshold
- make the *DLAB* template more specific

These examples clearly illustrate that *CLAUDIEN* can perform many of the tasks addressed in the data mining literature. We therefore believe that *CLAUDIEN* should be considered as a general purpose data mining environment and framework, which can be used for reasoning about and experimenting with various data mining problems. Of course, data mining research has always aimed at coping with large data sets in an efficient way, leading to very fast algorithms. As there is a general trade-off between generality of systems and their efficiency, *CLAUDIEN* cannot be expected to solve the above data mining problems as efficient as the best data mining algorithms. Nevertheless, we believe (and the other experiments in this section confirm our belief) that *CLAUDIEN* is reasonably efficient and can cope with reasonably large data sets. Furthermore, though data mining has focused on handling large data sets, inductive logic programming has focused on searching large hypotheses spaces.

5.2 Finite element mesh-design

One standard benchmark for inductive logic programming systems operating under the discriminatory setting, is that of learning finite element mesh-design (see e.g. [Dolšak and Muggleton, 1992; Lavrač and Džeroski, 1994]). Here we will address the same learning task. However, whereas the other approaches require positive as well as negative examples, *CLAUDIEN* needs only the positive. Secondly, the other approaches employ Michalski's covering algorithm, where the aim is to find hypotheses that cover each positive example once. *CLAUDIEN* follows an alternative approach, as it merely looks for valid rules. There is therefore no guarantee that hypotheses found by *CLAUDIEN* will cover all positives and also a hypothesis may cover a positive example several times. We believe – and our experiments in mesh-design show – that when the data are sparse, the *CLAUDIEN* approach is to be preferred.

The original mesh-application contains data about 5 different structures (a-e), with the number of edges per structure varying between 28 and 96. There are 278 positive examples (and 2840 negative ones) and the original background theory contains 1872 facts. The original background theory was made determinate (because the *GOLEM* system of [Muggleton and Feng, 1990] cannot work with indeterminate clauses). As *CLAUDIEN* does not suffer from this restriction, we could compact the database to 639 (equivalent facts). An example of a positive example is *mesh(b11,6)* meaning that edge 11 of structure *b* should be divided in 6 subedges. Background knowledge contains information about edge types, boundary conditions, loading, and the geometry of the structure. Some of the facts are shown below:

Edge types: long(b19), short(b10), notimportant(b2), shortforhole(b28), halfcircuit(b3), halfcircuithole(b1)

Boundary conditions: fixed(b1), twosidefixed(b6)

Loading: notloaded(b1), contloaded(b22)

Geometry: neighbour(b1, b2), opposite(b1, b3), same(b1, b3)

We ran *CLAUDIEN* on this data-set using a slightly different but equivalent representation for examples, using the leave-one-out strategy. All data were put into one observation. Counts of local accuracy $LA(c)$ and local coverage $LC(c)$ were done w.r.t. to the literal $mesh(E, R)$. Further settings include:

search strategy: best first

heuristic: $p/(l+n)$

$LA(c)$ threshold: 0.9

$LC(c)$ threshold: 2

DLAB grammar: see Figure 6

The *DLAB* grammar in Figure 6 defines a language of about $4.9 * 10^7$ rules. The antecedents of these rules specify at least the type, boundary conditions, loading or resolution of the edges that occur in the rule. Moreover, if two edges occur, the antecedent specifies their topology. The power of the *DLAB* formalism is thus used to prevent the generation of a large class of uninteresting rules.

On average *CLAUDIEN* halted after 7972 cpu seconds, visited 48534 nodes, which corresponds to about 0.01%, of the total search space, and discovered 495 valid rules. The high

```

mesh_temps =
  {R = resolution
  <--
  len-len: [ mesh(E,R),
             1-len: [type(E),boundary(E),loading(E)],
             0-len: [len-len: [geometry(E,E2),
                               1-len: [mesh(E2,resolution),
                                       type(E2),boundary(E2),loading(E2)]
                               ]
             ]
  ]
}
mesh_vars =
{dlab_variable(resolution,1-1,[1,2,3,4,5,6,7,8,9,10,11,12,17]),
 dlab_variable(type,1-1,[long,usual,short,circuit,half_circuit,
                        quarter_circuit,short_for_hole,long_for_hole,
                        circuit_hole,half_circuit_hole,notimportant]),
 dlab_variable(boundary,1-1,[free,one_side_fixed,two_side_fixed,fixed]),
 dlab_variable(loading,1-1,[noload,one_side_loaded,two_side_loaded,cont_loaded]),
 dlab_variable(geometry,1-1,[neighbour,opp,eq])}

```

Figure 6: A *DLAB* grammar for the mesh application

number of solutions can be explained by the low $LC(c)$ threshold.

In accordance to the any time character of *CLAUDIEN*, the discovered rules were tested against the structure left out at regular cpu time intervals. In cases where more than one rule applied, the earliest found rule with the highest heuristic value was preferred. In Figure 7 the percentage of correct predictions is plotted against cpu time elapsed. Notice the quality of the theory improves more or less logarithmically. Figure 7 also shows results for *GOLEM* and *FOIL* as they are reported in [Lavrač and Džeroski, 1994].

We believe the results of these tests are very encouraging because the rules learned by *CLAUDIEN* have by far the best classification accuracy and also because the cpu-requirements of *CLAUDIEN* are of the same order as those by the other systems. The high classification accuracy can be explained by the sparseness of the data and the non-covering approach. *FOIL* and *GOLEM* are implemented in C, and *CLAUDIEN* in *PROLOG*. The experiment clearly shows that an any time algorithm (implemented in *PROLOG*) is not necessarily slower than a covering approach. (Part of) a possible explanation for this may be that *CLAUDIEN* is the only system that does not need to employ the (large number) of negative examples.

5.3 Mutagenesis

To illustrate the scientific discovery potential of *CLAUDIEN* we selected a problem from the field of organic chemistry which was recently brought to the attention of the inductive logic programming community by the Oxford University Computing Laboratory, in collaboration with the London Biomolecular Modelling Laboratory [Srinivasan *et al.*, 1995b]. An observation here corresponds to a nitroaromatic compound with an associated mutagenicity value.

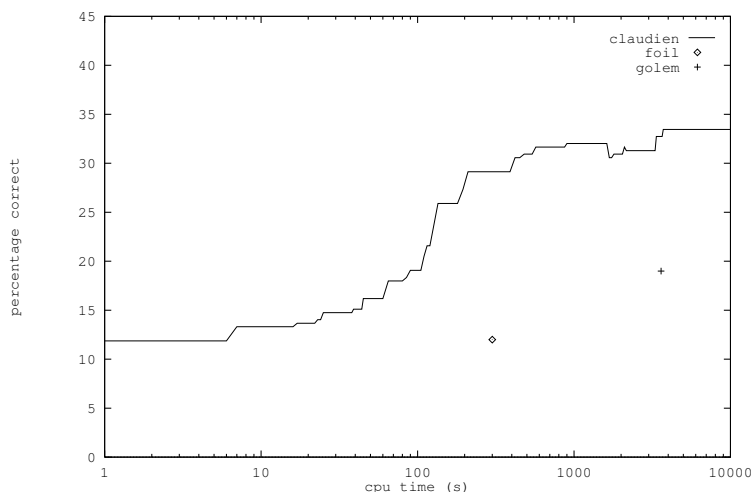


Figure 7: Comparing *CLAUDIEN* to *FOIL* and *GOLEM*.

There are 188 observations, 125 of which are labelled “active”, meaning they have high mutagenicity. The observations further list information on atom and bond structures, a measure of hydrophobicity (*logp*), the energy of the compound’s lowest unoccupied molecular orbital (*lumo*), and generic structural characteristics. For more details we refer to [Srinivasan *et al.*, 1995b].

So far experiments have focused on finding theories that discriminate between active and inactive compounds. For instance, with *PROGOL* [Muggleton, 1995] a predictive accuracy of 0.88 was obtained from a 10-fold cross-validation [Srinivasan *et al.*, 1995a]. Despite the classification oriented approach of *PROGOL*, the most interesting outcome of the experiments of the Oxford - London team is *not* a classification criterion, but rather a new structural alert for mutagenic compounds. The new structural alert basically encodes one of the rules found by *PROGOL*. However, as *PROGOL* aims at classification, it is interested in as short a hypothesis as possible, implying that it aims at a minimal number of rules. Indeed, according to Michalski’s covering approach, if a positive example is covered once by a rule in the hypothesis, it is no longer considered. Because of this, greedy classification algorithms may miss alternative explanations of the same data. *CLAUDIEN* performing essentially an informed exhaustive search, will not miss such alternative explanations.

To test this hypothesis, we ran *CLAUDIEN* on the mutagenesis problem with the aim of finding as much regularities of high accuracy and coverage as possible. The full *DLAB* grammar for this task can be found in Appendix B. We here mention only a special feature *#* borrowed from *PROGOL* to generate thresholds for the values *logp*, *lumo*, and atomic charge. Clauses output by *DLAB* contain bodyliterals such as *geteq(logp, LP, #(T))*, where, before validity of the clause is calculated, *#(T)* is replaced by a constant such that the clause is non-trivially valid in at least one observation.

A sample of the results is shown below and was obtained in several runs of *CLAUDIEN*, with a best-first search, with heuristic $p/(l+n)$, sometimes with slight variants of the *DLAB* grammar, sometimes with alternative thresholds for *GA(c)* and *GC(c)*. We ran first ran *CLAUDIEN* with settings $GA(c) > 0.9$ and $GC(c) > 80$. In 90 cpu seconds, 35 rules were

discovered, all variants of the following two:

```
active <-- lumo(Lumo) , lteq(lumo,Lumo,-1.62)
(accuracy: 0.9, coverage: 90)
```

```
active <-- not methyl(SP) , logp(LP) , gteq(logp,LP,3)
(accuracy: 0.9, coverage: 103)
```

We then lowered the $GC(c)$ threshold to 70. In two short subsequent runs, first with tests on thresholds for *logp*, *lumo*, and atomic charge disallowed, then with the structural characteristic *methyl* removed from the language, two alternative explanations were discovered:

```
active <-- not methyl(SP) , atom(A1,Elem1,Type1,Charge1) , Type1 = 27,
          atom(A2,Elem2,Type2,Charge2) , bond(A1,A2,7)
(accuracy: 0.91, coverage: 76)
```

```
active <-- benzene(SP),atom(A1,Elem1,Type1,Charge1),Type1 = 27,
          lteq(charge,Charge1,0.006)
(accuracy: 0.93, coverage: 70)
```

The underlying idea here is that the insights of one run, can be used in the next run. E.g. if the *not methyl* condition was allowed, nearly all rules discovered contained that condition. By excluding this condition, alternative explanations were found. Thus, the expert can and should guide the discovery process.

5.4 River water quality

The next application is taken from the domain of environmental monitoring [Džeroski *et al.*, 1994] (see also [Džeroski, 1995]). The goal here is to capture the expertise of an expert river ecologist who classified 292 field samples of benthic communities from British Midland Rivers. Each sample is described by means of the abundances (recorded on a scale of 0 to 6) of eighty different microinvertebrate families. The expert classified the samples into five classes.

In a first experiment we limited ourselves to discovering characteristics of poorest quality water. A simplified version of the *DLAB* grammar used is shown in Figure 8.

The size of the actual language used was of order 10^{96} . The accuracy threshold for $GA(c)$ was set to 1, but we used an extra feature of *CLAUDIEN* to list (but not prune) all rules with accuracy above a lower accuracy level set to 0.3. With 20% of the samples belonging to water quality class 0, the idea here was to delineate subgroups of water samples with a percentage of class 0 above average. Other relevant settings were:

```
search strategy: best first
heuristic:  $p/(l+n)$ 
 $GC(c)$  threshold: 10
```

```

eco_temps = {class(0)
  <--
    0-len:[len-len:[ancyliidae(A1),0-1:[compare(abundance,A1)]],
          len-len:[aselliidae(A2),0-1:[compare(abundance,A2)]],
          ...
          len-len:[veliidae(A80),0-1:[compare(abundance,A80)]]
    ]
}

eco_vars = {dlab_variable(compare, 1-1, [=,<,>],
  dlab_variable(abundance, 1-1, [0,1,2,3,4,5,6])}

```

Figure 8: A *DLAB* grammar for the river water quality application

We ran *CLAUDIEN* for about 1500 cpu seconds. In this period 2752 rules were discovered. After post-processing, we derived chains of the following type, where the addition of extra conditions leads to an increase of $GA(c)$ and a decrease of $GC(c)$.

	$GA(c)$	$GC(c)$
class(0) if true,	0.20	292
heptageniidae(D32),	0.69	75
hydropsychidae(D37),	0.73	49
oligochaeta(D54),	0.74	46
perlodidae(D57),	0.89	35
rhyacophilidae(D69),	0.93	29
tipulidae(D76),	0.96	26
D76 = 2	1	17

This setting where low accuracy rules are shown but not pruned, seems particularly interesting in cases where no rules with both high accuracy and high coverage are to be expected, for instance when sufficient conditions have to be discovered for the occurrence of certain “faults” in processes, machines, or human beings.

For a second experiment with the river quality data, we turned the lower accuracy facility off, set $GA(c)$ to 0.95, and modified the language such that rules could cover more than one class:

```

eco_temps = {class(1-2:[0,1,2,3,4])
  <--
  ....}

```

In a search space, now of order 10^{97} , *CLAUDIEN* discovered 49 rules in 24 hours of cpu time. For instance,

```

class(2) <-- aselliidae(A2), chironomidae(A11), gammaridae(A26),
  A26 = 2, lymnaeidae(A46)
(accuracy: 0.96, coverage: 28)

```

```
class(2), class(3) <-- asellidae(A2), glossiphoniidae(A28), physidae(A59)
(accuracy: 0.95, coverage: 22)
```

Ten of these rules have the disjunction $class(2), class(3)$ in the head, the others only $class(2)$. After eliminating the abundance level tests, and lowering the $GA(c)$ threshold to 0.9, *CLAUDIEN* discovered the following two rules with class disjunction within 20 cpu seconds:

```
class(2), class(3) <-- physidae(A59), tubificidae(A77)
(accuracy: 0.9, coverage: 40)
```

```
class(2), class(3) <-- asellidae(D2), physidae(D59)
(accuracy: 0.92, coverage: 39)
```

Finally, we removed $class(2)$ from the language, and raised the $GC(c)$ threshold to 30. In this modified setting, *CLAUDIEN* discovered 65 rules within 14 hours of cpu time, three of which are shown below:

```
class(0), class(1) <-- perlodidae(D57)
(accuracy: 1, coverage: 57)
```

```
class(0), class(1) <-- elminthidae(D21) , tubificidae(D77)
(accuracy(0.9), coverage: 80)
```

```
class(0), class(1) <-- heptageniidae(D32)
(accuracy: 1, coverage: 75)
```

In a similar experiment reported in [Džeroski *et al.*, 1994] class disjunction turned out to be the main reason why domain experts judged *CLAUDIEN* rules to be the most intuitive and promising, as compared to rules discovered by an extended version of the propositional learner CN2 [Clark and Niblett, 1989; Džeroski *et al.*, 1993] and *GOLEM*. This experiment illustrates *CLAUDIEN* can also be applied when class boundaries are vague or based on a discretization of a continuous space. If permitted by the *DLAB* bias, *CLAUDIEN* will attempt to disjunctively combine classes to construct valid rules. An analysis of the discovered hypothesis might then inspire the expert to introduce new (super)classes for frequent class combinations.

5.5 Parallel *CLAUDIEN*

In the final experiment, our aim was to measure and compare the speed at which sequential and parallel *CLAUDIEN* traverse the same hypothesis space. We tuned the mesh and ecology experiments such that in an exhaustive run *CLAUDIEN* visited about 120000 nodes. We then ran *CLAUDIEN* using a depth-first search strategy with 1, 2, 4, 8, and 16 processes. With

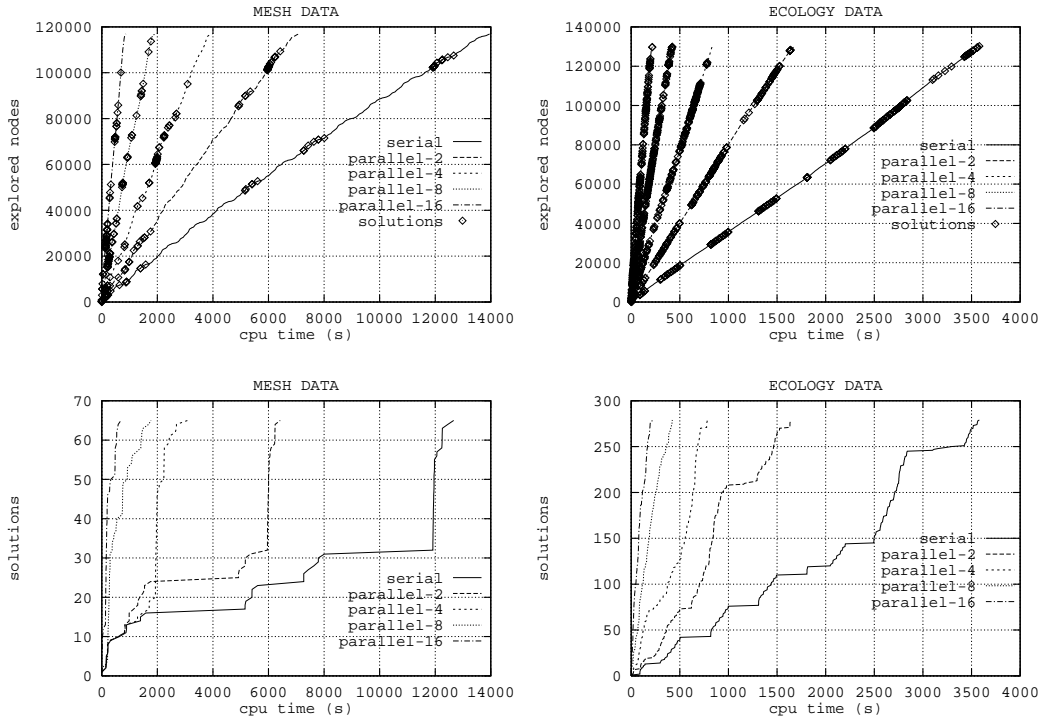


Figure 9: Results of the experiment

each tested clause, and again with each solution found, we recorded the consumed cpu time in seconds¹⁴.

The results of running *CLAUDIEN* with 1, 2, 4, 8, and 16 processes are reported in Figure 9. In the charts on top, the values on the y-axis are the number of explored nodes. If n is the degree of concurrency, and $explored(p, t)$ the number of nodes explored by process p after p has consumed t cpu seconds, then $y = f(t) = \sum_{p=1}^n explored(p, t)$. The clauses that were found to be valid are marked with a diamond. A separate chart with the number of solutions is presented in the lower half of Figure 9.

The results shown in Figure 9 indicate that for up to 16 processes, the speedup is approximately proportional to the number of processes executing the task: the consumed cpu time is roughly halved each time the number of processes is doubled.

An important question related to the results of our experiments with parallel *CLAUDIEN* is how long we can go on adding new processes to reduce the consumed cpu time. Apart from obvious hardware restrictions¹⁵, there are mainly two software related limitations we should take into account when trying to solve this question.

The first, application-dependent, upper boundary on the degree of concurrency stems from the fact that a (near) linear speedup can only be obtained if all processes are more or

¹⁴As cpu time was measured, we could test parallel *CLAUDIEN* with degrees above 4 on a machine with only 4 processors. It should be kept in mind however that the speedups here reported will only correspond to real time speedups if a separate processor is dedicated to all concurrent processes.

¹⁵Remember that we assume every process can execute on a separate processor. If not enough processors are available, they have to be switched between processes. By ever increasing the number of processes scheduled for a single processor we will finally overload the operating system.

less constantly working on a subtask, i.e. if most of the time there are enough sublanguages L_i available. The maximal number of candidate sublanguages available at a given time equals the total size of all local queues QC (see Figure 3) and is related to the application-specific average branching factor. It is for instance easy to see that in the extreme case where the branching factor equals 1, concurrency will produce no speedup at all.

Secondly, interprocess communication requires a certain amount of computational overhead. If this overhead increases with the degree of concurrency, as it does with our *naive* implementation of parallel *CLAUDIEN*, there will be a point where adding more processes is useless, or even counter-productive in terms of consumed cpu time.

6 Related Work

The clausal discovery engine presented here is related to data mining research, semantics for induction and inductive logic programming.

First, the techniques presented fit in an attempt to upgrade the data mining paradigm to considering multiple relations (cf. [Džeroski, 1995]). Evidence for this claim was provided by showing how the semantics for characterizing induction from closed observations fits in Manilla's general framework for data mining as well as by showing that *CLAUDIEN* can emulate many of the existing data mining systems. The emulations also demonstrate the generality of a first order clausal discovery engine as compared to propositional ones. As we discussed, the price to pay for generality and for expressive power, is a potential loss in efficiency on specific tasks. However, *CLAUDIEN* was shown not only to be able to search complex and vast hypotheses spaces, but also to handle reasonably large data sets. Furthermore, the task addressed by *CLAUDIEN* is PAC-learnable (cf. [De Raedt and Džeroski, 1994]), and the implemented engine is much more efficient than the naive algorithm used to prove the PAC-learning results. Thus *CLAUDIEN* should not be considered inefficient.

Secondly, the presented work also contributes to the semantics for induction. More specifically, it adopts the frameworks by [De Raedt and Džeroski, 1994] and [Helft, 1989]. It generalizes the work of Helft by the use of multiple observations (and models) as well as the use of Herbrand interpretations. Furthermore, it discusses many variants, options and extensions of the pure logical view of Helft and De Raedt and Džeroski.

Thirdly, clausal discovery is also a contribution to the field of inductive logic programming, in that it shows how a slightly different formalization of induction within logic programming results in new possibilities and challenges for inductive logic programming. One important contribution in this respect is the extension from definite clause logic to full clausal logic made possible by the novel semantics.

7 Conclusions

We have presented a clausal discovery engine based on a novel semantics for induction for use in a data mining setting. Theoretical properties of the engine as well as experiments with the engine were presented. A key ingredient of the engine was a declarative language bias formalism, with a corresponding refinement operator.

The clausal discovery engine and theory can be extended in various directions. First, it would be interesting to see how it can handle open observations (e.g. by using partial models). Secondly, how it can perform discriminating induction. A step in this direction was already taken by [De Raedt and Van Laer, 1995]. Thirdly, it would be interesting to see how the engine can be coupled to a relational database system and evaluate its performance on huge data bases. Finally, we wonder whether the clausal logic representation can be extended towards full first order logic.

We hope that the presented framework will provide a sound basis for combining data mining principles with inductive logic programming.

Acknowledgements

We would like to thank Sašo Džeroski and Maurice Bruynooghe for their involvement in the research that finally lead to this paper. Further discussions with Nada Lavrač, Stephen Muggleton and Peter Flach proved to be very fruitful. Bojan Dolsak, Sašo Džeroski and Ashwin Srinivasan generously provided the mesh, ecology and mutagenesis data used in the experiments. Patrick Weemeeuw provided advice on the parallel implementation of *CLAUDIEN*. We also thank Wim Van Laer for his significant contribution to the implementation of *CLAUDIEN*, and for his comments on this paper. Finally, Hendrik Blockeel as well as a (large) number of master's students experimented with earlier versions of the *CLAUDIEN* implementation.

Luc De Raedt is supported by the Belgian National Fund for Scientific Research and by the ESPRIT projects no. 6020 and 20237 on Inductive Logic Programming and Inductive Logic Programming II.

References

- [Adé *et al.*, 1995] H. Adé, L. De Raedt, and M. Bruynooghe. Declarative Bias for Specific-to-General ILP Systems. *Machine Learning*, 20(1/2):119 – 154, 1995.
- [Agrawal *et al.*, 1993] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 International Conference on Management of Data (SIGMOD 93)*, pages 207–216, May 1993.
- [Bergadano and Gunetti, 1993] F. Bergadano and D. Gunetti. An interactive system to learn functional logic programs. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1044–1049. Morgan Kaufmann, 1993.
- [Bergadano, 1993] F. Bergadano. Towards an inductive logic programming language. Technical Report ESPRIT project no. 6020 ILP Deliverable TO1, Computer Science Department, University of Torino, 1993.
- [Cameron-Jones and Quinlan, 1993] R.M. Cameron-Jones and J.R. Quinlan. Avoiding pitfalls when learning recursive theories. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1050–1055. Morgan Kaufmann, 1993.

- [Clark and Niblett, 1989] P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–284, 1989.
- [Clocksin and Mellish, 1981] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, Berlin, 1981.
- [Cohen, 1994] W.W. Cohen. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.
- [De Raedt and Bruynooghe, 1993] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann, 1993.
- [De Raedt and Džeroski, 1994] L. De Raedt and S. Džeroski. First order jk -clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.
- [De Raedt and Van Laer, 1995] L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the 5th Workshop on Algorithmic Learning Theory*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1995.
- [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*, pages 1037–1042. Morgan Kaufmann, 1993.
- [Dolšak and Muggleton, 1992] B. Dolšak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S. Muggleton, editor, *Inductive logic programming*, pages 453–472. Academic Press, 1992.
- [Džeroski *et al.*, 1993] S. Džeroski, B. Cestnik, and I. Petrovski. Using the m-estimate in rule induction. *Journal of Computing and Information Technology*, 1(1):37 – 46, 1993.
- [Džeroski *et al.*, 1994] S. Džeroski, L. Dehaspe, B. Ruck, and W. Walley. Classification of river water quality data using machine learning. In *Proceedings of the 5th International Conference on the Development and Application of Computer Techniques to Environmental Studies*, 1994.
- [Džeroski, 1995] S. Džeroski. Inductive logic programming and knowledge discovery in databases. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 118–152. The MIT Press, 1995.
- [Emde *et al.*, 1983] W. Emde, C.U. Habel, and C.R. Rollinger. The discovery of the equator or concept driven learning. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 455–458. Morgan Kaufmann, 1983.
- [Fayyad *et al.*, 1995] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. The MIT Press, 1995.
- [Fensel *et al.*, 1995] D. Fensel, M. Zickwolff, and M. Wiese. Are substitutions the better examples? In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 1995.

- [Flach, 1992] P. Flach. A framework for inductive logic programming. In S. Muggleton, editor, *Inductive logic programming*. Academic Press, 1992.
- [Flach, 1993] P. Flach. Predicate invention in inductive data engineering. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence, pages 83–94. Springer-Verlag, 1993.
- [Flach, 1994] P.A. Flach. Inductive logic programming and philosophy of science. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, Sankt Augustin, Germany, 1994. Gesellschaft für Mathematik und Datenverarbeitung MBH.
- [Flach, 1995] P. Flach. *An inquiry concerning the logic of induction*. PhD thesis, Tilburg University, Institute for Language Technology and Artificial Intelligence, 1995.
- [Genesereth and Nilsson, 1987] M. Genesereth and N. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann, 1987.
- [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):561–607, 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*, pages 335–359. Academic Press, 1992.
- [Lavrač and Džeroski, 1994] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
- [Lloyd, 1987] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 2nd edition, 1987.
- [MacDonald, 1979] I.G. MacDonald. *Symmetric functions and Hall polynomials*. Clarendon Oxford, 1979.
- [Mannila, 1995] H. Mannila. Aspects of data mining. In Y. Kodratoff, G. Nakhaeizadeh, and G. Taylor, editors, *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, pages 1–6, Heraklion, Crete, Greece, 1995.
- [Mitchell, 1982] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.
- [Muggleton and De Raedt, 1994] S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

- [Muggleton and Feng, 1990] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st conference on algorithmic learning theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.
- [Muggleton, 1995] S. Muggleton. Inverse entailment and prolog. *New Generation Computing*, 13, 1995.
- [Piatetsky-Shapiro, 1991] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, pages 229–248. The MIT Press, 1991.
- [Plotkin, 1970] G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
- [Quinlan, 1990] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
- [Rouveirol, 1994] C. Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14:219–232, 1994.
- [Savnik and Flach, 1993] I. Savnik and P.A. Flach. Bottom-up induction of functional dependencies from relations. In *Proceedings of the AAAI'93 Workshop on Knowledge Discovery in Databases*, pages 174–185. AAAI Press, 1993. Washington DC.
- [Schlimmer, 1991] J. Schlimmer. Learning determinations and checking databases. In *Proceedings of the AAAI'91 Workshop on Knowledge Discovery in Databases*, pages 64–76, 1991. Washington DC.
- [Shen, 1992] W.M Shen. Discovering regularities from knowledge bases. *International Journal of Intelligent Systems*, 7(7), 1992.
- [Srinivasan *et al.*, 1992] A. Srinivasan, S. Muggleton, and M. Bain. Distinguishing exceptions from noise in non-monotonic learning. In *Proceedings of the 2nd International Workshop on Inductive Logic Programming*, 1992.
- [Srinivasan *et al.*, 1995a] A. Srinivasan, S.H. Muggleton, and King. Comparing the use of background knowledge by two inductive logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 199–230, 1995.
- [Srinivasan *et al.*, 1995b] A. Srinivasan, S.H. Muggleton, M.J.E. Sternberg, and R.D. King. Theories for mutagenicity: a study in first-order and feature-based induction. *Artificial Intelligence*, 1995. To appear.
- [Sterling and Shapiro, 1986] Leon Sterling and Ehud Shapiro. *The art of Prolog*. The MIT Press, 1986.

function Injectivity

inputs : O : set of observations, B : background theory, c : clause

outputs : true if c is injective

Let c be $p_1, \dots, p_m \leftarrow q_1, \dots, q_n$.

Let X_1, \dots, X_k be $vars(c)$.

if $m \neq 0$

then return $\exists o \in O, \theta: (q_1 \wedge \dots \wedge q_n)\theta$ is true in $M(B \cup o)$ and $\forall i \neq j: X_i\theta \neq X_j\theta$.

else if $\exists o \in O, \theta: (q_1 \wedge \dots \wedge q_n)\theta$ is true in $M(B \cup o)$ and $\forall i \neq j: X_i\theta \neq X_j\theta$.

then return true.

else return $\forall k \exists \theta, o \in O: (q_1 \wedge \dots \wedge q_{k-1} \wedge q_{k+1} \dots \wedge q_n)\theta$ is true in $M(B \cup o)$ and

$\forall i \neq j: X_i\theta \neq X_j\theta$.

endif

endif

Figure 10: Function *Injectivity*

A Options in \mathcal{C} LAUDIEN

A.1 Injectivity

To incorporate injectivity (cf. Definition 3), let $Prune2(c) = Injectivity(B, o, c)$. *Injectivity* is defined in Figure 10.

Property 8 *If it terminates, $ClausalDiscovery(Prune2(c) = Injectivity(B, o, c))$ returns a characterizing solution in which all clauses are injective.*

Proof: As all clauses considered are injective and all injective clauses are considered, the result follows from property 6.

The fact that all considered clauses are injective inductively follows from the fact that \square , the empty clause, is injective, and the fact that all clauses added to Q are injective.

The fact that all injective clauses are considered follows from the observation that all refinements of a non-empty non-injective clause are non-injective. \square

ClausalDiscovery(Injective) may be further optimised by first counting the maximum number of ground terms in a model $M(B \cup o)$ and then pruning away (*prune2*) all clauses containing more variables than that maximum.

A.2 Maximally general clauses

Restricting hypothesis to include only maximally general clauses (cf. Definition 6) can be realized by setting $prune1(c) = notMaxgeneral(c)$. To implement *notMaxgeneral(c)* a locally complete generalization operator σ is needed.

```

function notmaxgeneral(c)
  inputs :  $O$ : set of observations,  $B$ : background theory,  $c$ : clause
  outputs : false if  $c$  is maxgeneral

for all  $c' \in \sigma(c)$  do
  if  $c'$  is valid in all  $M(B \cup o)$ 
  then return true
  endif
endfor
return false
endfunction

```

Figure 11: Function *notmaxgeneral*

Definition 33 σ is a locally complete generalization operator for a language \mathcal{L} if and only if σ is a mapping from \mathcal{L} to $2^{\mathcal{L}}$ such that $\forall c \in \mathcal{L} : \sigma(c) = \{c' \in \mathcal{L} \mid c' \text{ is a proper maximally specific generalisation of } c \text{ under } \theta\text{-subsumption in } \mathcal{L}\}$

Maxgeneral can then be implemented as shown in Figure 11.

Property 9 *ClausalDiscovery*($\text{prune1}(c) = \text{notmaxgeneral}(c)$) returns a hypothesis in which all clauses are maximally general if it terminates.

Proof: trivial □.

A.3 Non-triviality

Non-triviality (cf. Definition 5) can be implemented in a similar way. To incorporate non-triviality, let $\text{Prune2}(c) = \text{Non-trivial}(B, o, c)$. *Non-trivial* is defined in Figure 12.

Property 10 *If it terminates, ClausalDiscovery*($\text{Prune2}(c) = \text{Non-trivial}(B, o, c)$) returns a solution in which all clauses are non-trivial.

Proof: Similar. □.

A.4 Non-redundancy

Non-redundancy (cf. Definition 7) can be incorporated by taking $\text{Prune2}(c) = \text{Redundant}(B, c)$. Due to the undecidability of the redundancy check, one should employ in practice a depth-bound (see Figure 13).

Property 11 *If it terminates, ClausalDiscovery*($\text{Prune2}(c) = \text{Redundant}(B, c)$) returns a solution in which no clause is redundant.

Proof: Similar. □

function Non-trivial

inputs : O : set of observations, B : background theory, c : clause

outputs : true if c is non-trivial

Let c be $p_1, \dots, p_m \leftarrow q_1, \dots, q_n$.

if $m \neq 0$

then return $\exists o \in O, \theta: (q_1 \wedge \dots \wedge q_n)\theta$ is true in $M(B \cup o)$

else if $\exists o \in O, \theta: (q_1 \wedge \dots \wedge q_n)\theta$ is true in $M(B \cup o)$.

then return true.

else return $\forall k \exists \theta, o \in O: (q_1 \wedge \dots \wedge q_{k-1} \wedge q_{k+1} \dots \wedge q_n)\theta$ is true in $M(B \cup o)$.

endif

endif

Figure 12: Function *Non-trivial*

function Redundant

inputs : T : theory, c : clause

outputs : true if c is non-redundant

return $T \models c$

Figure 13: Function *Redundant*

```

function Compact
  inputs :  $H$ : hypothesis
  outputs : a compacted hypothesis

while  $\exists c \in H : H - \{c\} \models c$  do
  delete  $c$  from  $H$ 
endwhile
return  $H$ 

```

Figure 14: Function *Compact*

A.5 Compactness

First notice that one may always use the optimisation obtained by setting $prune1(c) = Redundant(H, c)$. This does not harm the maximal generality of induced hypotheses because if $H \models c$ then $H \cup c$ and H are logically equivalent.

Second, this optimisation is not sufficient to obtain compact hypotheses. The reason is that clauses induced later may render earlier induced clauses in the hypothesis logically implied. Compactness (cf. Definition 8) can be obtained by setting $Reduce(H) = Compact(H)$ (see Figure 14).

Property 12 *If it terminates, $ClausalDiscovery(Reduce(H) = Compact(H))$ returns a compact hypothesis.*

Proof: trivial. □

B A *D*LAB grammar for the mutagenesis application

```
muta_temps =
{active
  <--
  0-len:
  [toggle(structural_property(SP)),
  len-len:
  [atom(A1, Elem1, Type1, Charge1),
  0-len:[toggle(Elem1=element),
  toggle(Type1=atomtype),
  occurs_in(A1, SP)
  ],
  0-len:[len-len:[atom(A2, Elem2, Type2, Charge2),
  0-len:[toggle(Elem2=element),
  toggle(Type2=atomtype),
  occurs_in(A2, SP),
  bond(A1, A2, 1-1:[_,1,2,3,4,5,7]),
  len-len:[atom(A3, Elem3, Type3, Charge3),
  0-len:[toggle(Elem3=element),
  toggle(Type3=atomtype),
  occurs_in(A3, SP),
  bond(A1, A3, 1-1:[_,1,2,3,4,5,7]),
  bond(A2, A3, 1-1:[_,1,2,3,4,5,7])
  ]
  ]
  ]
  ],
  ],
  1-1:[eqtest(charge,1-1:[Charge1, Charge2, Charge3], #(T)),
  len-len:[lumo(Lumo),eqtest(lumo,Lumo, #(T))],
  len-len:[logp(LP),eqtest(logp,LP,#(T))]
  ]
  ]
}

muta_vars =
{dlab_variable(eqtest,1 - 1,[lteq,gteq]),
dlab_variable(element,1 - 1,[h,c,n,o,br,cl,f,i,s]),
dlab_variable(atomtype,1 - 1,[1,3,8,10,14,16,19,21,22,25,26,27,28,29,31,32,34,
35,36,38,40,41,42,45,49,50,51,52,72,92,93,94,
95,194,195,230,232]),
dlab_variable(structural_property,1 - 1,[nitro,carbon_6_ring,benzene,ring_size_6,
ring_size_5,phenanthrene,anthracene,ball3,
hetero_aromatic_5_ring,hetero_aromatic_6_ring,
carbon_5_aromatic_ring,methyl]),
dlab_variable(toggle,1 - 1,[call,not])
}
```