Condensed Representations for Inductive Logic Programming

Luc De Raedt
Institut fï Informatik
Albert-Ludwigs-University Freiburg
Georges Koehler Allee 79
D-79110 Freiburg, Germany
deraedt@informatik.uni-freiburg.de

Jan Ramon
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
Jan.Ramon@cs.kuleuven.ac.be

Abstract

When mining frequent Datalog queries, many queries will cover the same examples; i.e., they will be equivalent and hence, redundant. The equivalences can be due to the data set or to the regularities specified in the background theory. To avoid the generation of redundant clauses, we introduce various types of condensed representations. More specifically, we introduce δ-free and closed clauses, that are defined w.r.t. the data set, and semantically free and closed clauses, that take into account a logical background theory. A novel algorithm that employs these representations is also presented and experimentally evaluated on a number of benchmark problems in inductive logic programming.

Keywords: inductive logic programming, relational learning, relational data mining, condensed representations, frequent query mining, knowledge representation.

Introduction

One of the central tasks in data mining is that of finding all patterns that are frequent in a given database. The inductive logic programming instantiation of this task considers patterns that are logical queries or clauses and data in the form of a Datalog (or Prolog) knowledge base (Dehaspe & De Raedt, 1997; Dehaspe & Toivonen, 1999). This problem is known under the name of frequent Datalog query mining and has received quite some attention in the literature, cf. (Nijssen & Kok, 2001, 2003; Malerba & Lisi, 2001; Weber, 1997). Despite its popularity, there are several problems that arise when applying these techniques. First, it is computationally expensive to compute all frequent queries. This is due to the use of a very expressive formalism in which testing the generality of clauses (w.r.t. θ-subsumption) as well as the computation of the frequency of a clause are computationally hard. Second, a vast number of queries is being generated, tested and discovered. In contrast to simple pattern languages, such as item-sets, the clausal pattern space is infinitely large. Third, even though background knowledge is used when evaluating the frequency of clauses, no background knowledge is employed when generating the candidate clauses. So far, the standard way of guiding inductive logic programming systems computing frequent Datalog queries is through the specification of syntactic restrictions that the clauses should satisfy (the so-called language bias). Syntactic restrictions (and language bias) are not really declarative, they are hard to to specify and hence, they hinder the application of inductive logic programming. Furthermore, when analyzing the discovered clauses, it turns out that many of the patterns discovered will be equivalent, and hence, redundant. These redundancies do not only lead to unnecessary inefficiencies but also impose an unnecessary burden on the user. The redundancies can be due to the data set or to the background knowledge. As an illustration from the domain of beer consumption, consider the hypotheses *hoegaarden* $\wedge$ *duvel* and *duvel*. For a particular data set, these two hypotheses may cover exactly the same examples because in the available data set everybody that drinks *duvel* also drinks *hoegaarden*. On the other hand, it might also be that our background theory specifies that *hoegaarden* and *duvel* are both *beer*. In the light of this background theory, the patterns *hoegaarden* $\wedge$ *beer* and *hoegaarden* would be *semantically* equivalent, regardless of the data set.

In this paper, we address these problems by introducing various types of condensed representations for frequent Datalog query mining. Condensed representations are well studied for simple pattern domains such as item sets (Zaki, 2000; Boulicaut, Bykowski, & Rigotti, 2003; Pasquier *et al.*, 1999) and have recently also been applied in the context of graph mining (Yan & Han, 2002). The key idea underlying condensed representations (such as closed and free sets) is that one aims at finding a representative for each equivalence class of frequent patterns (the so-called closed set) instead of finding all frequent patterns. In this context, we first introduce the novel concept of *semantic* freeness and closedness, where the term *semantic* refers to the use of a declarative background knowledge about the domain of interest that is specified by the user. Under this notion, two clauses c_1 and c_2 are semantically equivalent if and only if $KB \models c_1 \leftrightarrow c_2$, e.g. *hoegaarden* $\wedge$ *beer* and *hoegaarden* are semantically equivalent in the light of *beer* $\leftarrow$ *hoegaarden*. Employing such a background knowledge and semantics is an elegant alternative to specifying complex and often ad hoc language bias constraints (such as those for dealing with symmetric or transitive predicates). In addition, it provides the user with a powerful and declarative tool for guiding the mining process. Secondly, we upgrade the

already existing notions of freeness and closedness (Zaki, 2000) for use in an inductive logic programming setting, and show how these traditional notions are related to the new notions of semantic condensed representations. If the two only predicates in the database are *hoegaarden* and *duvel*, and the examples covered by *hoegaarden* $\wedge$ *duvel* and *duvel* are the same, then *duvel* is free (i.e., a shortest clause in the equivalence class) and *hoegaarden* $\wedge$ *duvel* is closed (i.e., the longest clause in the equivalence class). As a third contribution, a novel frequent Datalog clause engine, called *c-armr*, that works with condensed representations, is presented and experimentally validated.

Problem Setting

Even though we assume some familiarity with Datalog and Prolog, the reader can find a brief overview of the most important concepts from logic programming in the Appendix.

The task of frequent query mining was first formulated in (Dehaspe & De Raedt, 1997; Dehaspe & Toivonen, 1999):

- **Given**
  - a language of clauses $\mathcal{L}$
  - a database $\mathcal{D}$
  - a predicate $key/1$ belonging to $\mathcal{D}$
  - a frequency threshold t
- **Find** all clauses $c \in \mathcal{L}$ such that $freq(c, \mathcal{D}) \geq t$.

The database $\mathcal{D}$ is a Prolog knowledge base; it contains the data to be mined. In this paper, we will assume that all queries posed to the knowledge base terminate and also that all clauses are range-restricted (i.e. all variables in the conclusion part of the clause also occur in the condition part). The first requirement can be guaranteed by employing Datalog (i.e., functor free Prolog). The database $\mathcal{D}$ is assumed to contain a special predicate $key/1$ which determines the entities of interest and what is being counted. The language of clauses $\mathcal{L}$ defines the set of well-formed clauses. All clauses in $\mathcal{L}$ are assumed to be of the form $p(K) \leftarrow key(K), q_1, ..., q_n$ where the q_i are different literals. Within inductive logic programming, $\mathcal{L}$ typically imposes syntactic restrictions on the clauses to be used as patterns, most notably, type and mode restrictions. The types indicate which arguments of the predicates belong to the same types. The modes provide information about the input/output behavior of the predicates. E.g., the predicate *member*(X, Y), which succeeds when X is an element of the list Y, runs safely when the second argument is instantiated upon call time. Therefore the typical mode restrictions states that the second argument is an *input* ('+') argument. The first can be an input or an output ('-') argument.

The *frequency* of a clause c with head $key(K)$ in database $\mathcal{D}$ is defined as

$$freq(c, \mathcal{D}) = \mid \{\theta \mid \mathcal{D} \cup c \models p(K)\theta\} \mid \quad (1)$$

So, the frequency of a clause is the number of instances of $p(K)$ that are logically entailed by the database and the clause. The frequency can be computed by asserting the clause c in the database already containing $\mathcal{D}$ and then running the query $?-p(K)$. The frequency then corresponds to the number of different answer substitutions for this query.

Example 1 *Consider the database consisting of the following facts:*

```
drinks(jan,duvel).          key(jan).
drinks(hendrik,cognac).     key(hendrik).
drinks(luc,hoegaarden).     key(luc).

beer(duvel).                brandy(cognac).
beer(hoegaarden).
```

Consider also the following clauses:

```
alcohol(X) ← beer(X).
alcohol(X) ← brandy(X).
false ← brandy(X), beer(X).
```

The frequency of the clause $p(X) \leftarrow key(X), drinks(X, B), beer(B)$ *is 2 as it has two answer substitutions:* $X = jan$ *and* $X = luc$.

Observe that the constraint $freq(c, \mathcal{D}) \geq t$ is anti-monotonic. A constraint c on patterns is *anti-monotonic* if and only if $c(p)$ and $q \preceq p$ implies $c(q)$, where, $q \preceq p$ denotes that q subsumes (i.e. generalizes) p.

An important problem with the traditional setting for frequent pattern mining in inductive logic programming is that – due to the expressiveness of clausal logic – the number of patterns that is considered and generated is extremely large (without syntactic restrictions, it is even infinitely large). This does not only cause problems when interpreting the results but also leads to computational problems. The key contribution of this paper is that we address this problem by introducing condensed representations for inductive logic programming.

Knowledge for Condensed Representations

It is often argued that one of the advantages of inductive logic programming is the ease with which one can employ background knowledge in the mining process. Background knowledge typically takes the form of dividing the database $\mathcal{D} = KB \cup D$ into two components: the background knowledge KB and the data D, where KB constitutes the *intensional* and D the *extensional* part of the database. The idea is then that both intensional and extensional predicates are used in the hypotheses and that their nature is transparent to the user. With only a few exceptions, most inductive logic programming systems do not employ the background theory during hypothesis generation[1] but only while computing the coverage or frequency of candidate clauses. Indeed, the typical inductive logic programming system structures the search space using θ-subsumption (Plotkin, 1970) or OI-subsumption (Malerba & Lisi, 2001), and starts searching at the empty clause and repeatedly applies a refinement operator. Because typical refinement operators do not employ background knowledge, they generate many clauses that are

[1] Perhaps, the only exception is Progol (Muggleton, 1995), which employs the background knowledge to generate a most specific clause in $\mathcal{L}$ that covers a specified example, and older heuristic systems that employ the background knowledge during refinement using a resolution based operator, cf. (Bergadano, Giordana, & Saitta, 1990).

semantically equivalent. As the inductive logic programming system is unaware of this, unnecessary work is being performed and the search space is blown up resulting in severe inefficiencies.

Example 2 *Reconsider Example 1. A typical refinement operator will generate* p(K) ← key(K), beer(K), alcohol(K); p(K) ← key(K), beer(K)*; and* p(K) ← key(K), alcohol(K). *However, given the clause stating that* beer *is* alcohol*, the first two clauses are equivalent.*

To alleviate this problem, we introduce, as the first contribution of this paper, the notions of semantically closed and semantically free clauses. These novel concepts assume that a background theory KB is given in the form of a set of Horn-clauses. At this point, we wish to stress that the background theory used should not contain information about specific examples and also that the theory used here might well be different than the one implicitly used in $\mathcal{D}$. So, KB here denotes a set of properties about the domain of interest. Continuing our example, KB consists of the two proper clauses defining *alcohol*.

Definition 1 *A clause* $h \leftarrow k, q_1, ..., q_n$ *is* semantically free, *or* s-free, *w.r.t. the background knowledge* KB*, if and only if there is no clause* $p_0 \leftarrow k, p_1, ..., p_m$ *(*$m < n$*) where each* p_i *corresponds to a single* q_j*, for which* $KB \models p_0 \leftarrow p_1, ..., p_m$[2].

Definition 2 *A clause* $h \leftarrow k, q_1, ..., q_n$ *is* semantically closed, *or* s-closed, *w.r.t. the background knowledge* KB *if and only if* $\{k\theta, q_1\theta, q_2\theta, ..., q_n\theta\}$ *is the least Herbrand model of* $KB \cup \{k\theta, q_1\theta, ..., q_n\theta\}$ *where* θ *is a skolem substitution for* $h \leftarrow k, q_1, ..., q_n$.[3]

Definition 3 *A clause* $h \leftarrow k, q_1, ..., q_n$ *is* consistent *if and only if* $KB \cup \{k\theta, q_1\theta, ..., q_n\theta\} \not\models \square$ *where* θ *is a skolem substitution.*

Intuitively, a clause is s-free if it is not possible to delete literals without affecting the semantics; it is s-closed if it is not possible to add literals without affecting the semantics. In Example 2, *p(K) ← key(K), beer(K), alcohol(K)* is s-closed, the other clauses are s-free, all clauses are consistent and the clause *p(K) ← key(K), beer(K), brandy(K).* is inconsistent because of the clause *false ← beer(X), brandy(X)*. It acts as a constraint on the set of legal clauses.

The reader familiar with the theory of inductive logic programming might observe that s-closed clauses are closely related to Progol's bottom clauses (Muggleton, 1995) as well as to Buntine's notion of generalized subsumption (Buntine, 1988). Indeed, the bottom clause of an s-closed clause w.r.t. the background theory is the s-closed clause itself, and the two clauses *p(K) ← key(K), beer(K), alcohol(K)* and *p(K) ← key(K), beer(K)* in Example 2, are equivalent under generalized subsumption. Observe also that each s-free clause has a unique s-closed clause, the *s-closure*, that is equivalent. Furthermore, several s-free clauses may have the same s-closure.

From the above considerations, it follows that it would be beneficial if the search could be restricted to generate only s-closed clauses. Then one would generate only one clause for each equivalence class of clauses and the s-closures would act as the canonical forms. This would closely correspond to having a refinement operator working under Buntine's generalized subsumption framework. Unfortunately, it is not easy to enforce the s-closed constraint as it is not anti-monotonic. Indeed, in our running example, *p(K) ← key(K), beer(K), alcohol(K)* is s-closed, but its generalization *p(K) ← key(K), beer(K)* is not. Fortunately, it turns out that s-freeness is anti-monotonic. Therefore it can easily be integrated in traditional frequent query mining algorithms. This integration will be discussed below.

Once the s-free clauses have been found, their s-closures can be computed and filtered to eliminate doubles. The s-closure of a clause $h \leftarrow k, q_1, ..., q_m$ w.r.t. the background theory can be computed as indicated in Algorithm 1.

Algorithm 1 Computing the s-closure of $h \leftarrow k, q_1, ..., q_m$ w.r.t. KB.

Compute a skolemization substitution θ for
    $h \leftarrow k, q_1, ..., q_m$
Compute the least Herbrand model $\{r_1, ..., r_n\}$
    of $KB \cup \{k\theta, q_1\theta, ..., q_m\theta\}$
Deskolemize $h\theta \leftarrow k\theta, r_1, ..., r_n$ and return the result

Example 3 *To compute the s-closure of* p(K) ← key(K), brandy(K)*, we first skolemize the clause using substitution* $\theta = \{\text{K} \leftarrow sk\}$ *where* sk *is the skolem constant. We then compute the least Herbrand model of the theory containing the facts* key(sk), brandy(sk) *and the clauses defining* alcohol*, yielding* key(sk), brandy(sk), alcohol(sk). *Deskolemizing then yields* p(K) ← key(K), brandy(K), alcohol(K).

Observe that the constraint of s-closedness provides the user with a powerful means to influence the results of the mining process. Indeed, adding or removing clauses from the background theory will strongly influence the number as well as the nature of the discovered patterns. Basically, adding a clause of the form $h \leftarrow p, q$ has the effect of ignoring h in clauses where p and q are already present. Therefore, the user may also desire to declare clauses in the background theory that do not possess a 100 per cent confidence. This in turn will increase the number of clauses that are semantically equivalent, and hence reduce the number of s-closed clauses. So, less clauses will be generated.

When working with a refinement operator that simply adds literals to hypotheses, one can also easily enforce the consistency constraint. Indeed, under these conditions, whenever a clause $p \leftarrow k, q_1, ..., q_m$ is consistent, all clauses of the form $p \leftarrow k, p_1, ..., p_n$ with $\{p_1, ..., p_n\} \subseteq \{q_1, ..., q_m\}$ will also be consistent. Therefore, the consistency constraint is anti-monotonic and can be incorporated in the same way as the s-freeness constraint.

[2] A more general but less operational definition requires that there is no clause $p_0 \leftarrow k, p_1, ..., p_m$ such that $p_0, ..., p_m$ subsumes $q_1, ..., q_n$.

[3] Here it is assumed that the least Herbrand model is finite, which can be enforced when using Datalog and range-restriction.

Discovering Associations

Specifying all clauses that hold in the domain may be cumbersome and the question arises as to whether the data mining system may not be able to discover the clausal regularities that hold among the data. For item sets, sequential patterns and even graphs (Yan & Han, 2002; Boulicaut, Bykowski, & Rigotti, 2003; Zaki, 2000), techniques have been developed that discover high confidence association rules during the mining process and once discovered, employ them to prune the search. To this aim, we upgrade the notions of δ-free and closed item sets (Boulicaut, Bykowski, & Rigotti, 2003) to clausal logic:

Definition 4 *A clause $h \leftarrow k, q_1, ..., q_n$ is* δ-free *(with δ being a small positive integer), if and only if there exists no clause c of the form $h \leftarrow k, p_1, ..., p_m, not\ p_0$, where each p_i corresponds to a single q_j, for which $freq(c, \mathcal{D}) \leq \delta$.*

To understand this concept, first consider the clauses $h \leftarrow k, p_1, ..., p_m, not\ p_0$ and the case that $\delta = 0$. If such a clause has frequency 0, this implies that the association rule $p_0 \leftarrow k, p_1, ..., p_m$ has a confidence of 100 per cent. The negative literal $not\ p_0$ is used because we need to know that the rule holds *for all* substitutions. Now, the definition of 0-freeness is analogous to that of s-freeness except that these association rules are not specified in the background theory but instead are regularities that hold in the data. Indeed,

Theorem 1 *For $\delta = 0$, if KB would contain all 100 per cent confidence association rules then a clause is δ-free if and only if it is s-free w.r.t. KB.*

Now consider the case that $\delta \neq 0$. Then rather than requiring association rules to be perfect, a (small) number of exceptions are allowed in each association rule.

Example 4 *The clause* p(K) ← key(K), drinks(K,B), beer(B) *is s-free and therefore 0-free. It is however not 1-free because* p(K) ← key(K), drinks(K,B), not beer(B) *has a frequency of 1. Similarly,* p(K) ← key(K), drinks(K,B), brandy(B) *is 0-free and 1-free but not 2-free.*

Observe that as δ increases, the number of δ-free clauses will decrease.

For $\delta = 0$, we can define a corresponding notion of closedness.

Definition 5 *A clause $h \leftarrow k, q_1, ..., q_n$ is* closed *if and only if there exists no clause c of the form $h \leftarrow k, p_1, ..., p_m, not\ p$,where each p_i (but not p) corresponds to a single q_j, for which $freq(c, \mathcal{D}) = 0$.*

Theorem 2 *If KB contains all 100 per cent confidence association rules then a clause is closed if and only if it is s-closed w.r.t. KB.*

We have not defined a corresponding notion of δ-closedness because traditional deduction rules do not hold any more. Indeed, from the fact that the association rules $p \leftarrow q$ and $q \leftarrow r$ have at most δ exceptions, one may not conclude that $p \leftarrow r$ has only δ exceptions.

Again, it is easy to see that δ-freeness is an anti-monotonic property, which will be useful when developing algorithms.

One of the interesting properties of δ-free clauses is that they can be used to closely approximate the frequencies of any frequent clause, cf. (Boulicaut, Bykowski, & Rigotti, 2003). The following theorem follows from a corresponding result for item sets due to (Boulicaut, Bykowski, & Rigotti, 2003).

Theorem 3 *Let $\mathcal{D}$ be a database. Let S be a set of δ-free clauses (w.r.t. $\mathcal{D}$). Let $c_1 : p(K) \leftarrow x_1 \ldots x_m$ and $c_2 : p(K) \leftarrow x_1 \ldots x_n$ be clauses with $n > m$ such that $\forall i \in \{m \ldots n-1\} : \exists (h \leftarrow b) \in S, \exists \theta : h\theta = x_{i+1} \wedge b\theta \subset \{x_1, \ldots, x_i\}$. Then, $freq(c_1, \mathcal{D}) \geq freq(c_2, \mathcal{D}) \geq freq(c_1, \mathcal{D}) - \delta.n$.*

An Algorithm for Mining Frequent Clauses

Algorithms for finding frequent Datalog queries are similar in spirit to those traditionally employed in frequent item set mining. A high-level algorithm for mining frequent clauses along these lines is shown in Algorithm 2. It searches the subsumption lattice breadth-first: it repeatedly generates candidate clauses C_i that are potentially frequent and tests for their frequency. To generate candidates, a refinement operator ρ is applied. To employ a minimum frequency threshold in Algorithm 2 one must set $con = (freq(f, \mathcal{D}) \geq t)$. Using the abstract constraint con, it is also possible to employ other types of anti-monotonic constraints in Algorithm 2. Despite the similarity with traditional frequent pattern mining algorithms, there are also some important differences.

First, traditional frequent pattern mining approaches assume that the language $\mathcal{L}$ is anti-monotonic. For clausal logic, a language is anti-monotonic when for all clauses $h \leftarrow k, p_1, ..., p_m \in \mathcal{L}$, all generalizations of the form $h \leftarrow k, p_1, ..., p_{i-1}, p_{i+1}, ..., p_m \in \mathcal{L}$. Even though this assumption holds for expressive pattern languages such as those involving trees or graphs, cf. (Inokuchi, Washio, & Motoda, 2003), it is typically invalid in the case of inductive logic programming because of the mode and type restrictions. Indeed, consider for instance the clause *p(K) ← key(K), benzene(K,S), member(A,S), atom(K,A,c)*. Even though this clause will typically satisfy the syntactic constraints, its generalization *p(K) ← key(K), member(A,S)* will typically not be mode-conform. Because the language $\mathcal{L}$ employed in inductive logic programming is not anti-monotonic, one need not only keep track of the frequent clauses, but also of the (maximally general) infrequent ones. Furthermore, when a new candidate is generated, it is tested whether the candidate is not subsumed by an already known infrequent one. Our *c-armr* implementation uses an indexing scheme to avoid subsumption tests wherever possible. This scheme relies on the observation that a clause c_1 can only θ-subsumes clause c_2 when all constant, predicate, and function symbols occurring in c_1 also occur in c_2.

Second, in order to search efficiently for solutions, it is important that each relevant pattern is generated at most once. Early implementations (Dehaspe & Toivonen, 1999) of frequent pattern mining systems in inductive logic programming were inefficient because they generated several syntactic variants of the same clause (clauses that are equiv-

alent under θ-subsumption) and had to filter these away using computationally expensive subsumption tests. One approach that avoids this problems defines a canonical form for clauses and employs a so-called optimal refinement operator that only generates clauses in canonical form, cf. (Nijssen & Kok, 2001, 2003). More formally, the canonical form we employ is defined as follows:

Definition 6 *A clause $h \leftarrow k, p_1, ..., p_m$ with variables $V_1, ..., V_n$ (ordered according to their first occurrence from left to right) is in canonical form if and only if $(h \leftarrow k, p_1, ..., p_m)\theta$ where $\theta = \{V_1 \leftarrow 1, ..., V_n \leftarrow n\}$ is the smallest clause according to the standard lexicographic order on clauses that can be obtained when changing the order of the literals p_i in the clause.*

If one furthermore requires that all variables in a pattern are instantiated to a different term (as in OI-identity (Malerba & Lisi, 2001)) when determining whether a clause covers an example it is possible to define an optimal refinement operator. A refinement operator ρ is optimal for a language $\mathcal{L}$ if and only if for all clauses $c \in \mathcal{L}$ there is exactly *one* sequence of clauses $c_0, ..., c_n$ such that $c_0 = \top$ (the most general element in the search space) and $c_n = c$ for which $c_i \in \rho(c_{i+1})$. So, when employing optimal refinement operators, there is exactly one path from $\top$ to each clause in the search space (De Raedt & Dehaspe, 1997). This approach is related to those employed when searching for frequent subgraphs, cf. (Yan & Han, 2002; Inokuchi, Washio, & Motoda, 2003), and is largely adapted in our implementation. More specifically, *c-armr*'s refinement operator works as follows[4]:

- a total order on the predicates is imposed
- only clauses $h \leftarrow b_1, ..., b_n$ in canonical form are refined
- all refinements are obtained by adding a literal b to the end of the clause
- the predicate in b must be larger than or equal to the predicate in b_n
- if the predicates in b and b_n are different, the refinement will be in canonical form
- if the predicate p in b and b_n is identical, it is tested whether re-ordering the literals containing p results in a smaller clause w.r.t. the lexicographic order.

Third, computing the frequency of a clause is computationally expensive as one evaluates a query against the whole database. Several optimizations have been proposed in this context, cf. (Blockeel *et al.*, 2002). In our implementation, we employ the smartcall introduced in (Santos Costa *et al.*, 2002) in combination with a heuristic for speeding up the computation of a coverage test (i.e. a test whether a clause covers a specific example). The heuristic orders the literals in a clause according to the number of answers it has. A more detailed description of the effect of such an optimisation is given in Struyf & Blockeel (2003). We also store the identifiers of the covered example with each frequent clause, a kind of vertical representation, which allows us to reduce the number of coverage tests needed as well as further optimizations.

One further feature of our implementation is worth mentioning. It was a design goal to produce a *light* Prolog implementation that would be small but still reasonably efficient. In this regard, because many Prolog systems lack intelligent garbage collection and have indexing problems for huge amounts of data, it turned out crucial that the main memory required by Prolog is kept as small as possible. This was realized by writing the sets F_i to files at level i, and then reading the frequent clauses c again at level $i+1$ in order to compute the refinements $\rho(c)$ potentially belonging to C_{i+1}. Similarly, the I_i are written to a file and indexed after each level. The code will be released in the public domain.

Algorithm 2 Computing all clauses that satisfy an anti-monotonic constraint con.

$C_0 := \{h(K) \leftarrow key(K)\}$
$i := 0; F_0 := \emptyset; I_0 := \emptyset$
while $C_i \neq \emptyset$ **do**
  $F_i := \{h \in C_i \mid con(h) = true\}$
  $I_i := C_i - F_i$
  $C_{i+1} := \{h \mid h \in \rho(h'), h' \in C_i\}$
  $i := i + 1$
  $C_i := \{h \mid h \in C_i$ and $\neg \exists s \in \bigcup_j I_j : s \preceq h\}$
end while

Adaptations for Mining Free Clauses

Let us now discuss how to adapt the previously introduced algorithm for mining free sets.

First, concerning the s-free clauses, we have made the following enhancements:

- use the constraint $(freq(c, \mathcal{D}) \geq t) \wedge$ *s-free*(c, KB) instead of only the minimum frequency threshold; this constraint is also anti-monotonic, hence the algorithm can directly be applied;
- those candidates that do not satisfy the s-freeness constraint are simply added to the appropriate I_i
- finally, before testing whether a candidate clause c is subsumed by an already known infrequent clause, replace c by its s-closure under KB; this will allow further pruning to take place.

Observe that the consistency constraint can be enforced in the same way.

Second, for what concerns the δ-free clauses, we employ the first two enhancements. Observe that it is possible as well as desirable to employ the constraint $(freq(c, \mathcal{D}) \geq t) \wedge \delta - free(c, \mathcal{D}) \wedge s - free(c, KB)$. Then one does not only use the already available knowledge in the background but also tries to discover new knowledge. One interesting alternative for adding the non-δ-free candidates to the I_i is to simply add them to the background theory. Doing so results in propagating the effects of the discovered association rules by combining their conclusions with those already in the background theory. When δ=0, this will always yield correct results. However, when $\delta \neq 0$, this might lead – in

[4]Some further complications arise when mode-declarations are employed.

```
molecule(225).                                  bond(225,f1_1,f1_2,7).
logmutag(225,0.64).                             bond(225,f1_2,f1_3,7).
lumo(225,-1.785).                               bond(225,f1_3,f1_4,7).
logp(225,1.01).                                 bond(225,f1_4,f1_5,7).
nitro(225,[f1_4,f1_8,f1_10,f1_9]).              bond(225,f1_5,f1_1,7).
atom(225,f1_1,c,21,0.187).                      bond(225,f1_8,f1_9,2).
atom(225,f1_2,c,21,-0.143).                     bond(225,f1_8,f1_10,2).
atom(225,f1_3,c,21,-0.143).                     bond(225,f1_1,f1_11,1).
atom(225,f1_4,c,21,-0.013).                     bond(225,f1_11,f1_12,2).
atom(225,f1_5,o,52,-0.043).                     bond(225,f1_11,f1_13,1).
...
ring_size_5(225,[f1_5,f1_1,f1_2,f1_3,f1_4]).
hetero_aromatic_5_ring(225,[f1_5,f1_1,f1_2,f1_3,f1_4]).
...
```

Figure 1: An example from the mutagenesis benchmark

some cases – to some unsound conclusions (because deduction using δ-free is not sound as discussed above).

Experiments

In this section we present an empirical evaluation of our claims. In particular, we will compare the time needed to find all frequent patterns and the number of clauses produced using the different settings discussed in this paper.

A first set of experiments will use the mutagenesis dataset, a popular benchmark in inductive logic programming (Srinivasan *et al.*, 1996). This database contains the descriptions of 230 molecules, including their structure (atoms and bonds), information on functional groups and some global chemical properties. Figure 1 contains part of the relational description of one molecule.

In a first experiment we investigate the effect of introducing a simple background theory and requiring the clauses to be s-free. Table 1 summarizes the results of this experiment. Two settings are considered. In the first setting, the standard frequent pattern discovery algorithm is used without a background theory. In the second one, the s-freeness constraint is enforced. Timings are given in seconds. The total running time up to each level as well as the total time used for testing which examples are covered by the clauses are displayed. As expected, enforcing s-freeness eliminates a substantial number of redundant clauses and also significantly reduces the computation time.

In a second experiment, we investigate δ-freeness. Table 2 specifies the number of discovered clauses and the cputime, when patterns are required to be δ-free with $\delta = 0$. Here, also the total time for discovering all δ-free rules up to each level are displayed. We again consider the cases with and without a background theory. As the background theory only contains rules that are 100 per cent correct, these rules are found as 0-free rules in the setting without a background theory and hence the number of patterns is equal in both settings. Still, the setting without background theory needs more time to find these rules.

The generated association rules include both simple ones such as

atom(Mol,Atm,_, 92,_) ← *atom(Mol,Atm,f,_,_)*

(all fluor atoms are of type 92) and more complex ones such as

sbond(Mol,Atm3,Atm2,1)←
atom(Mol,Atm3,c,_,_), atom(Mol,Atm2,o,49,_),
atom(Mol,Atm1,o,_,_), sbond(Mol,Atm3,Atm1,1)

The latter clause is not valid in general even though it holds for this specific database.

The number of clauses discovered in this setting is smaller than that in the previous case, where only a simple background theory was provided. On the other hand, discovering the δ-free rules requires a lot of time. Fortunately, association rules need to be discovered only once. The cost for coverage testing is significantly smaller as compared to the setting without δ-freeness. As this cost is proportional to the number of examples, one can expect more significant gains when larger databases are employed.

In a third experiment we consider the effect of setting $\delta > 0$. This causes a further reduction of the number of patterns found and the computation time needed. E.g.for $\delta = 40$, 7898 clauses are found in 6911 seconds. Table 3 compares the number of clauses generated by the standard algorithm to the number of δ-free clauses and the number of closed clauses. One can see that, while more costly to compute, the number of closed clauses is much smaller than the number of free clauses. Note that when one would close all δ-free clauses with $\delta > 0$, one would obtain a still smaller set. E.g. for $\delta = 20$, the number of closed clauses is less than 50% of the number for $\delta = 0$.

Tables 4, 5 and 6 report on similar experiments but now for the carcinogenesis dataset (Srinivasan, King, & Bristol, 1999). This dataset also contains descriptions of molecules, but it contains more variation and more complex molecules.

In Table 4, one can see that here too, using s-free clauses reduces the number of frequent patterns and the running time. In Table 5, we then consider the influence of δ-freeness. Again, we see that the number of patterns found is further reduced. For this dataset, the background theory also contains some properties that cannot be discovered as δ-free rules. Therefore, using s-freeness combined with δ-freeness

level	No theory			S-free clauses		
	#	run	cover	#	run	cover
0	1	0.3	0.0	1	0.3	0.0
1	6	0.4	0.0	6	0.4	0.0
2	45	1.9	0.9	45	0.8	0.0
3	225	9.6	3.8	206	5.0	2.2
4	1147	59.7	21.8	921	31.8	13.4
5	5458	404.1	157.3	3674	218.1	89.5
6	24611	2820.2	1298.9	14144	1429.1	627.8
7	83322	20880.0	11372.3	50366	9401.0	4582.2

Table 1: Number of patterns generated, total runtime and time needed for the coverage test on the Mutagenesis benchmark using the standard algorithm and the algorithms finding s-free clauses.

level	No theory				S-free clauses		
	#	run	cover	delta	run	cover	delta
0	1	0.4	0.0	0.0	0.3	0.0	0.0
1	6	0.4	0.0	0.0	0.4	0.0	0.0
2	43	2.7	0.6	1.0	0.9	0.2	0.2
3	193	12.2	2.7	3.5	7.7	2.1	2.7
4	811	73.6	13.5	21.4	51.3	12.1	19.7
5	3057	515.3	86.3	191.5	383.6	80.1	177.3
6	10976	3228.2	592.9	1324.3	2498.6	546.7	1220.3
7	36610	26062.7	4191.9	14840.6	21563.1	3853.0	13577.5

Table 2: Number of patterns generated, total runtime, time needed for the coverage test and time needed for the discovery of the δ-free rules on the Mutagenesis benchmark using the standard algorithm and the algorithms finding s-free patterns.

further reduces the number of frequent patterns as compared to using only δ-freeness. Table 6 then indicates the effect of closing the patterns. For this dataset, the number of patterns is reduced by more than 50% after 5 levels of mining while still yielding a very close approximation ($\delta = 1$). Of course, here too, increasing δ will further reduce the number of discovered patterns.

Conclusions and Related Work

We have introduced various types of condensed representations for use in inductive logic programming and we have demonstrated that this reduces the number of patterns searched for, the number of solutions as well as the time needed for the discovery task. Although condensed representations have been used in the context of simpler pattern languages (such as item sets, sequences and graphs), it is the first time that they have been employed within an inductive logic programming setting. A related and independently developed approach (Stumme, 2004), that was still under review, was brought to our attention after our paper had been accepted. It is related in that it investigates the theoretical properties of free and closed Datalog queries using formal concept analysis. However, it neither deals with semantic closures nor reports on an implementation or experiments. On the other hand, it contains exciting ideas about the use of formal concept analysis for visualization of the mining results.

The notions δ-free and closed clauses are a direct upgrade of the corresponding notions for item sets. However, the semantic notions are novel and could easily be applied to the simpler pattern domains as well. The semantic notions are somewhat related to the goal of deriving a non-redundant theory in the clausal discovery engines by (Helft, 1989; De Raedt & Dehaspe, 1997). In these engines, one was computing a set H of 100 per cent confidence association rules in the form of clauses such that no clause in H was logically redundant (i.e. entailed by the other clauses). Employing our semantic notions has a similar effect as working with Buntine's (1988) generalized subsumption notion.

At this point, we wish to stress that working with semantically free and/or closed clauses is not only useful when mining for frequent patterns, but can also be beneficial when mining for other types of patterns, such as for classification rules. This could – in an inductive logic programming setting – easily be implemented by employing a semantic refinement operator.

Appendix: Logic Programming Concepts

A first order *alphabet* is a set of predicate symbols, constant symbols and functor symbols. A *definite clause* is a formula of the form $A \leftarrow B_1, ..., B_n$ where A and B_i are logical atoms. An atom $p(t_1, ..., t_n)$ is a predicate symbol p/n followed by a bracketed n-tuple of terms t_i. A term t is a variable V or a function symbol $f(t_1, ..., t_k)$ 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 if B_1 and ... and B_n. All variables in clauses are universally quantified, although this is not explicitly written. We call A the *head* of the clause and $B_1, ..., B_n$ the

	# queries		# free sets		# closed sets		
	level	cumul	level	cumul	level	cumul	time
0	1	1	1	1	1	1	0.01
1	6	7	6	7	6	7	0.03
2	45	52	43	50	43	50	0.30
3	206	258	193	243	165	215	2.04
4	921	1179	811	1054	613	828	15.09
5	3674	4853	3057	4111	1961	2789	134.88
6	14144	18997	10976	15087	5895	8684	1631.02
7	50366	69363	36610	51697	16730	25414	27547.37

Table 3: Number of patterns compared to the number of δ-free patterns and the number of closed patterns, and the time needed to compute the closed clauses from the free clauses

level	No theory			S-free clauses		
	#	run	cover	#	run	cover
0	1	0.9	0.0	1	1.0	0.0
1	8	1.1	0.5	8	1.1	0.0
2	58	4.4	2.6	58	1.7	0.4
3	326	65.5	57.5	302	32.8	29.0
4	1758	446.3	392.2	1448	219.3	193.3
5	8819	4480.1	4093.4	6187	2067.1	1872.6

Table 4: Number of patterns generated, total runtime and time needed for the coverage test on the carcinogenesis benchmark using the standard algorithm and the algorithms finding s-free clauses.

body of the clause. A *fact* is a definite clause with an empty body, $(m = 1,\ n = 0)$. Throughout the paper, we assume that all clauses are *range restricted*, which means that all variables occurring in the head of a clause also occur in its body. A *substitution* $\theta = \{V_1 \leftarrow t_1, ..., V_k \leftarrow t_k\}$ is an assignment of terms to variables. Applying a substitution θ to a clause, atom or term e yields the expression $e\theta$ where all occurrences of variables V_i have been replaced by the corresponding terms. The *least Herbrand model* of a set of definite clauses is the smallest set of ground facts (over the alphabet) that is logically entailed by the definite clauses.

Acknowledgements

The first author was partly supported by the EU FET project cInQ (Consortium on Inductive Querying). The authors would like to thank Jean-Francois Boulicaut and Jan Struyf for interesting discussions about this work, and Andreas Karwath for his comments on an early version of this paper.

References

Bergadano, F.; Giordana, A.; and Saitta, L. 1990. Biasing induction by using a domain theory: An experimental evaluation. In *European Conference on Artificial Intelligence*, 84–89.

Blockeel, H.; Dehaspe, L.; Demoen, B.; Janssens, G.; Ramon, J.; and Vandecasteele, H. 2002. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research* 16:135–166.

Boulicaut, J.-F.; Bykowski, A.; and Rigotti, C. 2003. Free-sets: a condensed representation of boolean data for the approximation of frequency queries. *Data Mining and Knowledge Discovery journal* 7(1):5–22.

Buntine, w. 1988. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence* 36:375–399.

De Raedt, L., and Dehaspe, L. 1997. Clausal discovery. *Machine Learning* 26(2-3):99–146.

Dehaspe, L., and De Raedt, L. 1997. Mining association rules in multiple relations. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, 125–132. Springer-Verlag.

Dehaspe, L., and Toivonen, H. 1999. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery* 3(1):7–36.

Helft, N. 1989. Induction as nonmonotonic inference. In *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, 149–156. Morgan Kaufmann.

Inokuchi, A.; Washio, T.; and Motoda, H. 2003. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning* 50(3):321–354.

Malerba, D., and Lisi, F. A. 2001. Discovering associations between sapatial objects: An ILP application. In *Proceedings of the 11th International Conference on Inductive Logic Programming*, volume 2157 of *Lecture Notes in Artificial Intelligence*, 156–163. Springer-Verlag.

Muggleton, S. 1995. Inverse entailment and progol. *New Generation Computing* 13(3–4):245–286.

level	No theory				S-free clauses			
	#	run	cover	delta	#	run	cover	delta
0	1	0.9	0.0	0.0	1	1.0	0.0	0.0
1	8	1.1	0.0	0.0	8	1.1	0.0	0.0
2	56	7,7	2.8	3.0	56	2.2	0.5	0.4
3	310	80.5	57.5	15.0	286	43.9	29.0	10.7
4	1623	627.6	390.8	241.9	1346	350.6	194.3	127.3
5	7903	11420.0	4070.0	6969.0	5642	5725.1	1891.8	3576.8

Table 5: Number of patterns generated, total runtime, time needed for the coverage test and time needed for the discovery of the δ-free rules on the carcinogenesis benchmark using the standard algorithm and the algorithms finding s-free patterns.

	# queries		# free sets		# closed sets		
	level	cumul	level	cumul	level	cumul	time
0	1	1	1	1	1	1	0.01
1	8	9	8	9	8	9	0.02
2	56	65	56	65	56	65	0.14
3	302	367	286	351	262	327	0.64
4	1448	1815	1346	1697	1112	1439	3.80
5	6167	7982	5642	7339	4038	5477	35.65

Table 6: For the carcinogenesis dataset, the number of patterns compared to the number of δ-free patterns and the number of closed patterns, and the time needed to compute the closed clauses from the free clauses

Nijssen, S., and Kok, J. N. 2001. Faster association rules for multiple relations. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001*, 891–896. Seattle, Washington, USA: Morgan Kaufmann.

Nijssen, S., and Kok, J. N. 2003. Efficient frequent query discovery in farmer. In *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 2838 of *Lecture Notes in Computer Science*. Springer Verlag.

Pasquier, N.; Bastide, Y.; Taouil, R.; and Lakhal, L. 1999. Efficient mining of association rules using closed itemset lattices. *Journal of Information Systems* 24:25–46.

Plotkin, G. 1970. A note on inductive generalization. *Machine Intelligence* 5:153–163.

Santos Costa, V.; Srinivasan, A.; Camacho, R.; Blockeel, H.; Demoen, B.; Janssens, G.; Struyf, J.; Vandecasteele, H.; and Van Laer, W. 2002. Query transformations for improving the efficiency of ilp systems. *Journal of Machine Learning Research* 4:465 – 491.

Srinivasan, A.; Muggleton, S.; Sternberg, M. J. E.; and King, R. D. 1996. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence* 85(1–2):277–299.

Srinivasan, A.; King, R.; and Bristol, D. 1999. An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, 291–302.

Struyf, J., and Blockeel, H. 2003. Query optimization in inductive logic programming by reordering literals. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, volume 2835 of *Lecture Notes in Computer Science*, 329–346.

Stumme, G. 2004. Iceberg query lattices for datalog. Technical Report.

Weber, I. 1997. Discovery of first-order regularities in a relational database using offline candidate determination. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, 288–295. Springer-Verlag.

Yan, X., and Han, J. 2002. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*. Japan: IEEE Computer Society.

Zaki, M. J. 2000. Generating non-redundant association rules. In *Proceedings of ACM SIGKDD Conference on Knowledge Discovery in Databases*, 34–43.