A Theory of Inductive Query Answering

Luc DE RAEDT[†] Manfred JAEGER[†,‡,⋆] Sau Dan LEE[†] Heikki MANNILA[‡]

[†]Inst. für Informatik
University of Freiburg
Georges Koehler Allee 79
D-79110 Freiburg, Germany
deraedt@informatik.uni-freiburg.de
danlee@informatik.uni-freiburg.de

[‡] Helsinki Institute of
Information Technology
PO Box 26
FIN-00014 Univ. of Helsinki
Finland
mannila@cs.helsinki.fi

[⋆] MPI Informatik
Stuhlsatzenhausweg 85
D-66123 Saarbrücken
Germany
jaeger@mpi-sb.mpg.de

Abstract

We introduce the boolean inductive query evaluation problem, which is concerned with answering inductive queries that are arbitrary boolean expressions over monotonic and anti-monotonic predicates. Secondly, we develop a decomposition theory for inductive query evaluation in which a boolean query Q is reformulated into k sub-queries $Q_i = Q_A \wedge Q_M$ that are the conjunction of a monotonic and an anti-monotonic predicate. The solution to each subquery can be represented using a version space. We investigate how the number of version spaces k needed to answer the query can be minimized. Thirdly, for the pattern domain of strings, we show how the version spaces can be represented using a novel data structure, called the version space tree, and can be computed using a variant of the famous Apriori algorithm. Finally, we present some experiments that validate the approach.

1. Introduction

Many data mining problems address the problem of finding a set of patterns that satisfy a constraint. Formally, this can be described as the task of finding the set of patterns $Th(Q, \mathcal{D}, \mathcal{L}) = \{\varphi \in \mathcal{L} \mid Q(\varphi, \mathcal{D})$, i.e. those patterns φ satisfying query Q on database $\mathcal{D}\}$. Here $\mathcal{L}$ is the language in which the patterns or rules are described and Q is a predicate or constraint that determines whether a pattern φ is a solution to the data mining task or not [15]. This framework allows us to view the predicate or the constraint Q as an *inductive query* to an *inductive database system*. It is then the task of the inductive database management system to efficiently generate the answers to the query. This view of data mining as a declarative querying process is also appealing as the basis for a theory of data mining. Such a theory would be analogous to traditional database theory in the sense that one could study properties of different pattern languages $\mathcal{L}$, different types of queries (and query languages), as well as different types of databases. Such a theory could also serve as a sound basis for developing algorithms that solve inductive queries.

It is precisely such a theory that we introduce in this paper. More specifically, we study inductive queries that are boolean expressions over monotonic and anti-monotonic predicates. An example query could ask for molecular fragments that have frequency at least 30 per cent in the active molecules or frequency at most 5 per cent in the inactive ones [14]. To the best of our knowledge this type of boolean inductive query is the most general type of inductive query that has been considered so far in the data mining literature. Indeed, most contemporary approaches to constraint based data mining use either single constraints (such as minimum frequency), e.g. [2], a conjunction of monotonic constraints, e.g. [17, 10], or a conjunction of monotonic and anti-monotonic constraints, e.g. [4, 14]. However, [6] has studied a specific type of boolean constraints in the context of association rules and item sets. It should also be noted that even these simpler types of queries have proven to be useful across several applications, which in turn explains the popularity of constraint based mining in the literature.

Our theory of boolean inductive queries is first of all concerned with characterizing the solution space $Th(Q, \mathcal{D}, \mathcal{L})$ using notions of convex sets (or version spaces [13, 12, 16]) and border representations [15]. This type of representations have a long history in the fields of machine learning [13, 12, 16] and data mining [15, 3]. These data mining and machine learning viewpoints on border sets have recently been unified by [4, 14], who introduced the level-wise version space algorithm that computes the S and G set w.r.t. a conjunction of monotonic and anti-monotonic constraints.

In the present paper, we build on these results to develop a decomposition approach to solving arbitrary boolean queries over monotonic and anti-monotonic predicates. More specifically, we investigate how to decompose arbitrary queries Q into a set of sub-queries Q_k such that $Th(Q, \mathcal{D}, \mathcal{L}) = \bigcup_i Th(Q_i, \mathcal{D}, \mathcal{L})$, k is minimal, and each $Th(Q_i, \mathcal{D}, \mathcal{L})$ can be represented using a single version space. This results in an operational and effective decomposition procedure for solving queries. Indeed, the overall query Q is first reformulated into the sub-queries Q_i, which can then be solved by existing algorithms such as the level-wise version space algorithm of [4].

Our theory is then instantiated to answer boolean queries about string patterns. String patterns are widely applicable in the many string databases that exist today, e.g. in DNA or in proteins. Furthermore, the present work is to a large extent motivated by the earlier MolFea system [14, 4], in which conjunctive queries (over anti-monotonic and monotonic constraints) for molecular features were solved using a version space approach. MolFea features are essentially strings that represent sequences of atoms and bonds. For string patterns, we introduce a novel data structure, i.e. version space trees, for compactly representing version spaces of strings. Version space trees combine ideas of version spaces with those of suffix trees. They have various desirable properties. Most notably, they can be computed using a variant of traditional level wise algorithms for tries, recognizing whether a string belongs to the version space is linear in the size of the string, and the size of the version space tree is at most quadratic in the size of the elements in the S set of the version space.

This paper is organized as follows. In Section 2, we define the inductive query evaluation problem and illustrate it on the pattern domains of strings and item-sets; in Section 3, we introduce a decomposition approach to reformulate the original query in simpler sub-queries; in Section 4, we introduce version space trees that compactly represent the solutions to a sub-query in the pattern domain of strings; in Section 5, we report on some experiments in this domain, and, finally, in Section 6, we conclude.

2 Boolean Inductive Queries

A pattern language $\mathcal{L}$ is a formal language for specifying patterns. Each pattern $\phi \in \mathcal{L}$ matches (or covers) a set of examples ϕ_e, which is a subset of the universe $\mathcal{U}$ of possible examples. In general, pattern languages will not allow to represent all subsets $\mathcal{P}(\mathcal{U})$ of the universe[1].

Example 2.1 *Let $\mathcal{I} = \{i_1, \ldots, i_n\}$ be a finite set of possible items, and $\mathcal{U}_\mathcal{I} = 2^\mathcal{I}$ be the universe of item sets over $\mathcal{I}$. The traditional pattern language for this domain is $\mathcal{L}_\mathcal{I} = \mathcal{U}_\mathcal{I}$. A pattern $\phi \in \mathcal{L}_\mathcal{I}$ covers the set $\phi_e := \{\psi \subseteq \mathcal{I} \mid \phi \subseteq \psi\}$. An alternative, less expressive, pattern language is the language $\mathcal{L}_{\mathcal{I},k} \subseteq \mathcal{L}_\mathcal{I}$ of item sets of size at most k.*

[1]The terminology used here is similar to that in concept-learning, where $\mathcal{U}$ would be the space of examples, $\mathcal{P}(\mathcal{U})$ the set of possible concepts, and $\mathcal{L}$ the set of concept-descriptions.

Example 2.2 *Let Σ be a finite alphabet and $\mathcal{U}_\Sigma = \Sigma^*$ the universe of all strings over Σ. We will denote the empty string with ϵ. The traditional pattern language in this domain is $\mathcal{L}_\Sigma = \mathcal{U}_\Sigma$. A pattern $\phi \in \mathcal{L}_\Sigma$ covers the set $\phi_e = \{\psi \in \Sigma^* \mid \phi \sqsubseteq \psi\}$, where $\phi \sqsubseteq \psi$ denotes that ϕ is a substring of ψ. An alternative, more expressive, language is the language of all regular expressions over Σ.*

One pattern ϕ is *more general* than a pattern ψ, written $\phi \succeq \psi$, if and only if $\phi_e \supseteq \psi_e$.

A pattern *predicate* defines a primitive property of a pattern, usually relative to some data set D (a set of examples), and sometimes other parameters. For any given pattern, it evaluates to either *true* or *false*.

We now introduce a number of pattern predicates that will be used for illustrative purposes throughout this paper. Most of these predicates are inspired by MolFea [14]. Our first pattern predicates are very general in that they can be used for arbitrary pattern languages:

- min_freq(p,n,D) evaluates to true iff p is a pattern that occurs in database D with frequency at least $n \in \mathbb{N}$. The frequency $f(\phi, D)$ of a pattern ϕ in a database D is the (absolute) number of data items in D covered by ϕ. Analogously, the predicate max_freq(p, n, D) is defined.

- ismoregeneral(p,ψ) is a predicate that evaluates to true iff pattern p is more general than pattern ψ. Dual to the ismoregeneral predicate one defines the ismorespecific predicate.

The following predicate is an example predicate tailored towards the specific domain of string-patterns over $\mathcal{L}_\Sigma$.

- length_atmost(p,n) evaluates to true for $p \in \mathcal{L}_\Sigma$ iff p has length at most n. Analogously the length_atleast(p,n) predicate is defined.

In all the preceding examples the pattern predicates have the form pred(p,*params*) or pred(p,D,*params*), where *params* is a tuple of parameter values, D is a data set and p is a pattern variable.

We also speak a bit loosely of pred alone as a pattern predicate, and mean by that the collection of all pattern predicates obtained for different parameter values *params*.

We say that m is a *monotonic* predicate, if for all possible parameter values *params* and all data sets D:

$$\forall \phi, \psi \in \mathcal{L} \text{ such that } \phi \succeq \psi : \quad \mathsf{m}(\psi, D, \mathit{params}) \rightarrow \mathsf{m}(\phi, D, \mathit{params})$$

The class of *anti-monotonic* predicates is defined dually. Thus, min_freq, ismoregeneral, and length_atmost are monotonic, their duals are anti-monotonic.

A pattern predicate pred(p,D,*params*) that can be applied to the patterns from a language $\mathcal{L}$ defines relative to D the *solution set* $Th(\mathsf{pred}(p, D, \mathit{params}), \mathcal{L}) = \{\phi \in \mathcal{L} \mid \mathsf{pred}(\phi, D, \mathit{params}) = \mathit{true}\}$. Furthermore, for monotonic predicates m these sets will be monotone, i.e. for all $\phi \succeq \psi \in \mathcal{L} : \psi \in Th(\mathsf{m}, \mathcal{L}) \rightarrow \phi \in Th(\mathsf{m}, \mathcal{L})$.

Example 2.3 *Consider the string data set* $D = \{abc, abd, cd, d, cd\}$. *Here we have pattern frequencies* $f(abc, D) = 1$, $f(cd, D) = 2$, $f(c, D) = 3$, $f(abcd, D) = 0$. *And trivially,* $f(\epsilon, D) = |D| = 5$. *Thus, the following predicates evaluate to true: min_freq(c; 2; D), min_freq(cd; 2; D), max_freq(abc; 2; D), max_freq(cd; 2; D).*

The pattern predicate $m := \mathit{min\_freq}(p, 2, D)$ *defines* $Th(m, \mathcal{L}_\Sigma) = \{\epsilon, a, b, c, d, ab, cd\}$, *and the pattern predicate* $a := \mathit{max\_freq}(p, 2, D)$ *defines the infinite set* $Th(a, \mathcal{L}_\Sigma) = \mathcal{L}_\Sigma \setminus \{\epsilon, c, d\}$.

The definition of $Th(\mathsf{pred}(\mathrm{p}, D, \mathit{params}), \mathcal{L})$ is extended in the natural way to a definition of the solution set $Th(Q, \mathcal{L})$ for boolean combinations Q of pattern predicates over a unique pattern variable: $Th(\neg Q, \mathcal{L}) := \mathcal{L} \setminus Th(Q, \mathcal{L})$, $Th(Q_1 \vee Q_2, \mathcal{L}) := Th(Q_1, \mathcal{L}) \cup Th(Q_2, \mathcal{L})$. The predicates that appear in Q may reference one or more data sets $D_1, \ldots, D_n$. To emphasize the different data sets that the solution set of a query depends on, we also write $Th(Q, D_1, \ldots, D_n, \mathcal{L})$ or $Th(Q, \mathcal{D}, \mathcal{L})$ for $Th(Q, \mathcal{L})$.

We are interested in computing solution sets $Th(Q, \mathcal{D}, \mathcal{L})$ for boolean queries Q that are constructed from monotonic and anti-monotonic pattern predicates. As anti-monotonic predicates are negations of monotonic predicates, we can, in fact, restrict our attention to monotonic predicates. We can thus formally define the *boolean inductive query evaluation problem* addressed in this paper.

Given

- a language $\mathcal{L}$ of patterns,
- a set of monotonic predicates $\mathcal{M} = \{\mathsf{m}_1(p, D_1, \mathit{params}_1), ..., \mathsf{m}_n(p, D_n, \mathit{params}_n)\}$,
- a query Q that is a boolean expression over the predicates in $\mathcal{M}$ (and over a single pattern variable),

Find the set of patterns $Th(Q, D_1, \ldots, D_n, \mathcal{L})$, i.e. the solution set of the query Q in the language $\mathcal{L}$ with respect to the data sets $D_1, \ldots, D_n$.

3 A decomposition approach

The query evaluation problem for a query Q will be solved by decomposing Q into k sub-queries Q_i such that Q is equivalent to $Q_1 \vee ... \vee Q_k$, and then computing $Th(Q, \mathcal{D}, \mathcal{L})$ as $\cup_i Th(Q_i, \mathcal{D}, \mathcal{L})$. Furthermore, each of the sub-queries Q_i will be such that $Th(Q_i, \mathcal{D}, \mathcal{L})$ is a version space (also called a convex space), and therefore can be efficiently computed for a wide class of pattern languages $\mathcal{L}$, and queries Q_i.

Definition 3.1 *Let* $\mathcal{L}$ *be a pattern language, and* $I \subseteq \mathcal{L}$. I *has* dimension *1, if* $\forall \phi, \phi', \psi \in \mathcal{L} : \phi \preceq \psi \preceq \phi'$ *and* $\phi, \phi' \in I \implies \psi \in I$. I *has dimension* k *if it is the union of* k *subsets of dimension 1, but not the union of* $k-1$ *subsets of dimension 1.*

A query Q *has dimension* k *(with respect to the pattern language* $\mathcal{L}$*) if* k *is the maximal dimension of any solution set* $Th(Q, \mathcal{D}, \mathcal{L})$ *of* Q *(where the maximum is taken w.r.t. all possible data sets* $\mathcal{D}$ *and w.r.t. the fixed language* $\mathcal{L}$*).*

If Q has dimension 1 w.r.t. $\mathcal{L}$, then $Th(Q, \mathcal{D}, \mathcal{L})$ is a version space [16] or a convex space [13]. Version spaces are particularly useful when they can be represented by boundary sets, i.e. by the sets $G(Q, \mathcal{D}, \mathcal{L})$ of their maximally general elements, and $S(Q, \mathcal{D}, \mathcal{L})$ of their minimally general elements. For the theoretical framework of the present section we need not assume boundary representability for convex sets. However, concrete instantiations of the general method we here develop, like the one described in sections 4 and 5, usually will assume pattern languages in which convexity implies boundary representability.

Example 3.2 *Reconsider the string domain. Let*

$$\begin{aligned} Q_1 &= \mathit{ismoregeneral}(p, abcde) \wedge \mathit{length\_atleast}(p, 3) \\ Q_2 &= \mathit{ismorespecific}(p, ab) \wedge \mathit{ismorespecific}(p, uw) \\ &\quad \wedge (\mathit{length\_atleast}(p, 6) \vee \mathit{min\_freq}(p, 3, D)) \end{aligned}$$

The query Q_1 *does not reference any dataset, and* $Th(Q_1, \mathcal{L}_\Sigma) = \{abcde, abcd, bcde, abc, bcd, cde\}$. *This set of solutions is completely characterized by* $S(Q_1, \mathcal{L}_\Sigma) = \{abcde\}$ *and* $G(Q_1, \mathcal{L}_\Sigma) = \{abc, bcd, cde\}$. $Th(Q_2, D, \mathcal{L}_\Sigma)$ *cannot in general be represented using a single version space. However, as our general method will show, the dimension of* $Th(Q_2, D, \mathcal{L}_\Sigma)$ *is at most two, so that it can be represented as the union of two version spaces.*

With the following definition and lemma we provide an alternative characterization of dimension k sets.

Lemma 3.3 Let $I \subseteq \mathcal{L}$. Call a chain $\phi_1 \preceq \phi_2, \preceq \ldots \preceq \phi_{2k-1} \subseteq \mathcal{L}$ an *alternating chain (of length k) for I* if $\phi_i \in I$ for all odd i, and $\phi_i \notin I$ for all even i. Then the dimension of I is equal to the maximal k for which there exists in $\mathcal{L}$ an alternating chain of length k for I.

Example 3.4 *Consider the following queries:*
$Q_3 = \mathit{ismoregeneral}(p, abc) \wedge \mathit{ismorespecific}(p, a)$,
$Q_4 = \mathit{ismoregeneral}(p, c)$, *and* $Q_5 = Q_3 \vee Q_4$.
Then c, bc, abc *is an alternating chain of length 2 for* $Th(Q_5, \mathcal{L}_\Sigma)$.

Given Q and $\mathcal{L}$ we are now interested in computing the dimension k of Q, and transforming Q into a disjunction $\vee_{h=1}^{k} Q_i$, such that each $Th(Q_i, \mathcal{D}, \mathcal{L})$ is a version space. The approach we take is to first evaluate Q in a reduced pattern language $\mathcal{L}^{adm}_{\mathcal{M}(Q)}$, so that the desired partition $\vee Q_i$ can be derived from the structure of $Th(Q, \mathcal{L}^{adm}_{\mathcal{M}(Q)})$. The solution set $Th(Q, \mathcal{L}^{adm}_{\mathcal{M}(Q)})$ does not depend on the datasets $\mathcal{D}$ that Q references, and the complexity of its computation only depends on the size of Q, but not on the size of any datasets.

Definition 3.5 *For a query* Q, *let* $\mathcal{M}(Q) = \{m_1, \ldots, m_n\}$ *be the set of monotonic predicates contained in* Q *(where predicates that only differ with respect to parameter values also are considered distinct). Define* $\mathcal{L}_{\mathcal{M}(Q)} := 2^{\mathcal{M}(Q)}$. *A subset* $\phi \subseteq \mathcal{M}(Q)$ *is called* admissible *if there exists data sets* $\mathcal{D}$ *such that* $\mathrm{Th}(\wedge_{m_i \in \phi} m_i \wedge_{m_j \notin \phi} \neg m_j, \mathcal{D}, \mathcal{L})$ *is not empty. Let* $\mathcal{L}^{\mathrm{adm}}_{\mathcal{M}(Q)} = \{\phi \in \mathcal{L}_{\mathcal{M}(Q)} \mid \phi \text{ admissible}\}$.

For the predicates m_i *we define* $\mathrm{Th}(m_i, \mathcal{L}_{\mathcal{M}(Q)})$, *respectively* $\mathrm{Th}(m_i, \mathcal{L}^{\mathrm{adm}}_{\mathcal{M}(Q)})$, *as the set of (admissible)* ϕ *that contain* m_i. *By the general definition this also determines* $\mathrm{Th}(Q, \mathcal{L}_{\mathcal{M}(Q)})$ *and* $\mathrm{Th}(Q, \mathcal{L}^{\mathrm{adm}}_{\mathcal{M}(Q)})$.

Example 3.6 *Using only monotonic predicates, the query* Q_2 *from example 3.2 can be rewritten as* $\neg m_1 \wedge \neg m_2 \wedge (\neg m_3 \vee m_4)$, *with*

$$\begin{aligned} m_1 &= \mathit{not\text{-}ismorespecific}(p, ab) \\ m_2 &= \mathit{not\text{-}ismorespecific}(p, uw) \\ m_3 &= \mathit{not\text{-}length\_atleast}(p, 6) \\ m_4 &= \mathit{min\_freq}(p, 3, D) \end{aligned}$$

(where e.g. not-ismorespecific is the (monotonic) complement of the anti-monotonic predicate ismorespecific; note that this is distinct from ismoregeneral).

Here every $\phi \subseteq \{m_1, \ldots, m_4\}$ *is admissible (a witness for the admissibility of* $\{m_3, m_4\}$, *for instance, is a dataset* D *in which the string* abuw *appears at least three times, i.e.* abuw $\in \mathrm{Th}(\neg m_1 \wedge \neg m_2 \wedge m_3 \wedge m_4, D, \mathcal{L}_\Sigma)$*).*

Figure 1 (a) shows $\mathcal{L}_{\mathcal{M}(Q_2)} = \mathcal{L}^{\mathrm{adm}}_{\mathcal{M}(Q_2)}$, *where e.g. pattern* $\{m_3, m_4\}$ *is just represented by its "index string" 34.*

Now consider a variant Q_6 *of* Q_2 *obtained by replacing* m_3 *with* $m_3' := \mathit{not\text{-}length\_atleast}(p, 4)$. *Here not every* $\phi \subseteq \{m_1, m_2, m_3', m_4\}$ *is admissible: as* $\mathit{ismorespecific}(p, ab) \wedge \mathit{ismorespecific}(p, uw)$ *implies* $\mathit{lengthatleast}(p, 4)$, *we have that neither* $\{m_3'\}$ *nor* $\{m_3', m_4\}$ *are admissible. These are the only two inadmissible subsets of* $\mathcal{M}(Q)$, *so that* $\mathcal{L}^{\mathrm{adm}}_{\mathcal{M}(Q_6)}$ *here is as in figure 1 (b).*

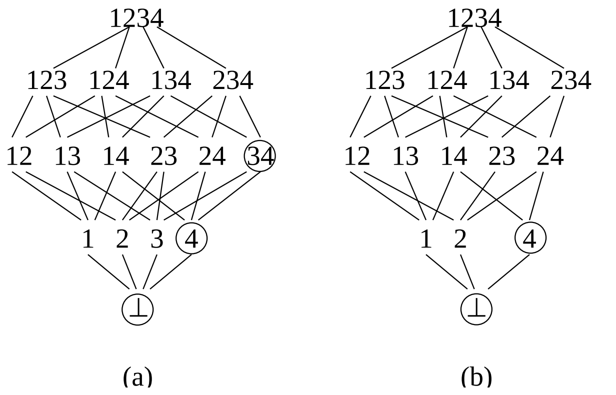

Figure 1. Pattern languages $\mathcal{L}^{adm}_{\mathcal{M}(Q)}$

Assuming that we can decide admissibility of subsets of $\mathcal{M}(Q)$ (for the types of pattern languages and predicates we have considered so far admissibility can always be decided), we can construct $\mathcal{L}^{adm}_{\mathcal{M}(Q)}$ and compute $Th(Q, \mathcal{L}^{adm}_{\mathcal{M}(Q)})$. These solution sets are indicated for the queries Q_2 and Q_6 in their respective languages $\mathcal{L}^{adm}_{\mathcal{M}(Q)}$ by circles in figure 1. One sees that $Th(Q_2, \mathcal{L}^{adm}_{\mathcal{M}(Q_2)})$ has dimension 2, and $Th(Q_6, \mathcal{L}_{\mathcal{M}(Q_6)})$ has dimension 1. This gives an upper bound for the dimensions of the solutions to the query:

Theorem 3.7 *The dimension of* $Th(Q, \mathcal{L}^{\mathrm{adm}}_{\mathcal{M}(Q)})$ *is an upper bound for the dimension of* $Th(Q, \mathcal{D}, \mathcal{L})$ *for all datasets* $\mathcal{D}$.

The dimension of $Th(Q, \mathcal{L}_{\mathcal{M}(Q)})$ is greater or equal the dimension of $Th(Q, \mathcal{L}^{adm}_{\mathcal{M}(Q)})$, and therefore also can serve as an upper bound for the dimension of $Th(Q, \mathcal{D}, \mathcal{L})$. In general, this will be a coarser bound: for Q_6, for instance, we obtain for $Th(Q, \mathcal{L}_{\mathcal{M}(Q_6)})$ the same structure as shown for Q_2 in figure 1 (a), and therefore only the bound 2.

When $Th(Q, \mathcal{L}^{adm}_{\mathcal{M}(Q)})$ is of dimension k, we can define each of its convex components I_h as a solution to a query Q_h in the predicates m_i: if $\phi_1, \ldots, \phi_l$ are the maximal and $\psi_1, \ldots, \psi_m$ the minimal elements of I_h, then $I_h = Th(Q_h, \mathcal{L}^{adm}_{\mathcal{M}(Q)})$ for

$$Q_h := (\vee_{i=1}^{l} \wedge_{m_j \notin \phi_i} \neg m_j) \wedge (\vee_{i=1}^{m} \wedge_{m_j \in \psi_i} m_j) \qquad (1)$$

Theorem 3.8 $Th(Q_h, \mathcal{D}, \mathcal{L})$ *is convex for all datasets* $\mathcal{D}$, *and* $Th(Q, \mathcal{D}, \mathcal{L}) = Th(\vee_{h=1}^{k} Q_h, \mathcal{D}, \mathcal{L}) = \cup_{h=1}^{k} Th(Q_h, \mathcal{D}, \mathcal{L})$.

Example 3.9 *Continuing from example 3.6, we can partition* $\mathrm{Th}(Q_2, \mathcal{L}^{\mathrm{adm}}_{\mathcal{M}(Q_2)})$ *into two convex components* $I_1 = \{\{m_4\}, \{m_3, m_4\}\}$ *and* $I_2 = \{\perp\}$. *We thus obtain the de-*

composition of the query Q_2 into the two subqueries

$$\begin{aligned} Q_{2_1} = \;& \textit{ismorespecific}(p, ab) \wedge \textit{ismorespecific}(p, uw) \\ & \wedge \textit{min\_freq}(p, 3, D) \\ Q_{2_2} = \;& \textit{ismorespecific}(p, ab) \wedge \textit{ismorespecific}(p, uw) \\ & \wedge \textit{length\_atleast}(p, 6) \wedge \neg \textit{min\_freq}(p, 3, D) \end{aligned}$$

For Q_6 we have that $\mathrm{Th}(Q_6, \mathcal{L}^{\mathrm{adm}}_{\mathcal{M}(Q_6)})$ consists of one version space $\{\bot, \{m_4\}\}$, so that Q_6 is equivalent to the query

$$\begin{aligned} Q_{6_1} = \;& \textit{ismorespecific}(p, ab) \wedge \textit{ismorespecific}(p, uw) \\ & \wedge \textit{length\_atleast}(p, 4) \end{aligned}$$

The sub-queries (1) to which the original query Q is reduced not only are known to have convex solution sets $Th(Q_h, \mathcal{D}, \mathcal{L})$, they also are of a special syntactic form $Q_h = Q_{h,M} \wedge Q_{h,A}$, where $Q_{h,M}$ defines a monotone set $Th(Q_{h,M}, \mathcal{D}, \mathcal{L})$, and $Q_{h,A}$ defines an anti-monotone set $Th(Q_{h,A}, \mathcal{D}, \mathcal{L})$. This factorization of Q_h facilitates the computation of the border sets $G(Q_h, \mathcal{D}, \mathcal{L})$ and $S(Q_h, \mathcal{D}, \mathcal{L})$, for which the level wise version space algorithm [4, 14] can be used. In the following section we will present an algorithm that for queries in the string domain uses the syntactic form of the Q_h for efficiently computing and representing the solution sets $Th(Q_h, \mathcal{D}, \mathcal{L})$ with *version space trees*.

4. Version space trees

In this section, we introduce a novel data structure, called the version space tree, that can be used to elegantly represent and index a version space of strings, e.g. the $Th(Q_h, \mathcal{D}, \mathcal{L}_\Sigma)$ introduced in the previous section. Furthermore, we present effective algorithms that compute version space trees containing all strings that satisfy the conjunction of a monotonic and an anti-monotonic predicate (as in the queries Q_h).

4.1. The data structure

A *trie* is a tree where each edge is labelled with a symbol from the alphabet Σ. Moreover, the labels on every edge emerging from a node must be unique. Each node n in a trie thus uniquely represents the string $s(n)$ containing the characters on the path from the root r to the node n. The root node itself represents the empty string ϵ.

A *suffix trie* is a trie with the following properties:

- For each node n and for each suffix t of $s(n)$, there is also a node n' in the trie representing t, i.e. $t = s(n')$.
- Each node n has a *suffix link* *suffix*$(n) = n'$, where $s(n')$ represents the suffix obtained from $s(n)$ by dropping the first symbol. The root node represents ϵ, which has no suffixes. We define *suffix*$(root) = \bot$, where $\bot$ is a unique entity.

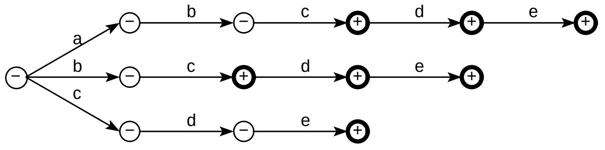

Figure 2. An example version space tree

Suffix tries have been well studied in the literature [18, 19]. However, we make some important deviations from the main stream approach:

- instead of building a suffix trie on all the suffixes of a *single* string, we are indexing all the suffixes of a *set of strings*; thus multiple strings are stored in the trie.
- we use *fully labelled* tries, in which each node is labelled with either "$\oplus$" or "$\ominus$"; the $\oplus$ label to indicate nodes that are interesting to us (say: belong to the version space) and $\ominus$ for those that are not.
- because we need to store labels and counts for all substrings represented in the trie, we do not coalesce chains of nodes with only one out-going edge into a single edge label.

Note that a fully labelled trie may contain nodes for which its label as well as those of its descendants are all $\ominus$. Thus the node as well as the subtrie below it are uninteresting. Therefore, in practice we will often use a *pruned labelled trie*. This is a fully labelled trie with the additional property that all leaf nodes have the sign $\oplus$. Both trees have the same semantics and each fully labelled tree has a unique equivalent pruned tree. Furthermore,as most of our results are valid for both types of trees, we will often employ the term "labelled trie".

Now *a version space tree* V is a labelled trie that represents a version space of strings over $\mathcal{L}_\Sigma$. More formally, let V be a set of strings of dimension 1. Then the corresponding (pruned) version space trie T is such that $V = \{v \mid n \text{ is a node in } T \text{ with label } \oplus \text{ and } s(n) = v\}$. Figure 2 illustrates the (pruned) version space tree representing $Th(Q_7, \mathcal{D}, \mathcal{L}_\Sigma)$, where $Q_7 = \mathsf{is\_more\_general}(t, abcde) \wedge (\mathsf{is\_more\_specific}(t, bc) \vee \mathsf{is\_more\_specific}(t, cde))$.

A version space tree VST representing version space V has the following properties:

1. All leaf nodes are labelled $\oplus$.
2. Along every path from root to a leaf there is at most one sign change (from $\ominus$ to $\oplus$); cf. Lemma 3.3.
3. If $S = min\ V$ then VST will 1) have a leaf corresponding to each $s \in S$ and 2) have a node corresponding to each suffix s' of each $s \in S$ for which $s' \in V$.

4. Therefore, the number of nodes in the version space tree VST is at most $\Sigma_{i \in S} |s_i|^2$, where $|s|$ denotes the length of the string. However, the size of a VST is usually much smaller.

5. Testing whether a string s belongs to the version space represented by a version space VST is linear in $|s|$, as the VST can be interpreted as a deterministic automaton on input s.

6. Property 3 can be used as the basis for an algorithm for constructing a version space tree based on S and G.

7. For a given version space tree, one can easily and efficiently construct the S and G-sets. Indeed, the S-set will contain all leafs l of the version space tree to whom no suffix pointer points; and the G-set will contain all nodes g with label $\oplus$ whose parent node has label $\ominus$ and for which the node *suffix*(g) either does not exist or also has the label $\ominus$.

As one can see, there is a close correspondence between version spaces of strings and version space trees. We will now show that there is also a close correspondence between version space trees and algorithms such as Apriori [2].

4.2. The algorithms

In this section, we sketch the VST algorithm to build a version space tree that satisfies the conjunction $Q_A \wedge Q_M$ of an anti-monotonic predicate Q_A and a monotonic one Q_M. This form of query corresponds to the one of the queries Q_h that would be generated by our decomposition (over anti-monotonic and monotonic constraints) approach. Algorithm VST is a level-wise algorithm based on the well-known Apriori [2] algorithm. The algorithm assumes 1) that the version space tree to be computed is finite and 2) that the alphabet Σ is given. It consists of two phases:

DESCEND: top-down growing of the version space tree using the monotonic predicate Q_M.

ASCEND: bottom-up marking of the version space tree using the anti-monotonic predicate Q_A.

Both phases are designed to minimize the number of database scans[2]. As such, they both exhibit the cyclic pattern: candidate generation, candidate testing (database scan) and pruning. The cycle terminates when no more new candidates patterns are generated.

Since only the monotonic pattern predicate is handled in the descend phase, we can reuse the idea of Apriori. The algorithm searches the strings satisfying Q_M in a top-down, breadth-first manner. At each depth level k, the descend algorithm first expands the $\oplus$ nodes found in the previous iteration (L_{k-1}). The nodes resulting from the expansion constitute the set C_k. These candidate nodes are then tested against the predicate Q_M. The testing involves one database scan for the whole iteration. The candidate patterns in C_k that satisfy the constraints are put into L_k. Those that do not are pruned away from the tree. This process is repeated in a level wise fashion until C_k becomes empty. All generated nodes are labelled with $\oplus$ and the necessary suffix links are set up during this phase.

Note that the sets C_k and L_k are the same as the candidate sets and "large" sets in the Apriori algorithm. Moreover, the generation of C_k from L_{k-1} also mimics the Apriori-join operation in the Apriori algorithm.[3] The descend algorithm makes use of the suffix like and parent-child relationship of a suffix trie to perform the join efficiently. More specifically, the candidate child nodes of a node n in L_{k-1} (as well as the edges) correspond to the children of the node *suffix*(n). So, the major difference between DESCEND and Apriori is that the former also organizes the discovered strings into a suffix trie, facilitating the join operation and the second phase of the VST algorithm.

The second phase is implemented with algorithm ASCEND. This phase handles the anti-monotonic constraint Q_A. Here we assume that we have the set F_0 of leaf nodes in the tree T generated during the descend phase. While DESCEND works top-down, ASCEND starts from the leaves and works upwards. It first checks the leaf nodes against the predicate Q_A. The labels of all the nodes n that do not satisfy Q_A, are changed into $\ominus$. In addition, all their ancestors are also labelled as $\ominus$. This is sound due to the anti-monotonicity. So, we can propagate these $\ominus$ marks upwards until we have marked the root with $\ominus$. Actually, we can stop as soon as we reach an ancestor already marked with $\ominus$, as another such leaf node n' may share some ancestors with n. So, all the ancestors from that point upwards have already been marked with $\ominus$. Secondly, for those nodes p in F_0 that satisfy Q_A, the label remains unchanged (i.e. $\oplus$). Furthermore, we will enter their parent into the set F_1 (and remove possible duplicates). F_1 contains the nodes to be considered at the next iteration. This process is then repeated until F_k becomes empty.

So, after these two phases, namely DESCEND and then ASCEND, both the monotonic and the anti-monotonic constraints are handled. With a simple tree traversal, we can prune away those subtrees that contain only $\ominus$ labels. The

[2]As in Apriori, one only needs to scan the data sets at most once for each level of the tree.

[3]There are some differences here since we are dealing with strings instead of sets. E.g., while Apriori-join generates item set {a, b, c} from {a, b} and {a, c}, the descend algorithm generates `abc` from `ab` and `bc`, because these are the *only* immediately shorter *substrings* of `abc`. At the same time, it is not hard to imagine a variant of the version space tree algorithm for use with item sets. Indeed, the kind of trie searched is quite similar to some of the data structures used by e.g. [3, 11].

result is a tree that is a pruned suffix trie representing the version space of strings that satisfy the query $Q_A \wedge Q_M$.

Theorem 4.1 *The* VST *algorithm performs at most* $2m$ *database scans, where* m *is length of the longest strings satisfying the monotonic query* Q_M.

5. Experiments

We have implemented the VST algorithm and performed experiments on datasets of command histories collected from 168 Unix users over a period of time [7]. The users are divided into four groups: computer scientists, experienced programmers, novice programmers and non-programmers. The corresponding data sets are denoted "sci", "exp", "nov" and "non", respectively. When each user accesses the Unix system, he first logs in, then types in a sequence of commands, and finally logs out. Each command is recorded as a symbol in the database. The sequence of commands from log in to log out constitutes a login session, and is mapped to a string in our experiment. Each user contributes to many login sessions in the database. Table 1 gives some statistics on the data.

In the first set of experiments we determined solutions of queries $\mathsf{min\_freq}(p, n, D)$ for the four different datasets and for thresholds n that were selected so as to produce solution sets of around 300 frequent string patterns. Table 1 summarizes the datasets, the queries, and their solutions. Timings (wall-clock time on a Pentium III 600 Mhz) are reported as well.

Whereas the first set of experiments only used the min_freq predicate, the second set of experiments involves the computation of two version space trees T_1 and T_2 corresponding to the queries Q_8 and Q_9:

$$Q_8 : \mathsf{min\_freq}(p, non, 24) \wedge \mathsf{max\_freq}(p, sci, 60)$$

$$Q_9 : \mathsf{min\_freq}(p, nov, 80) \wedge \mathsf{max\_freq}(p, exp, 36)$$

Q_8 and Q_9 are conjunctions of an anti-monotonic predicate and a monotonic one, thus their solution space is a version space. Furthermore, they are the sub-queries that are generated for the query $Q_{10} = Q_8 \vee Q_9$ using the decomposition approach outlined in Section 3.

The results of the second experiment are shown in Table 2. Each row shows the time the algorithm spent on building that tree. The columns of the table show the number of nodes and total length of strings represented by those nodes. Each of the five sub-column in each case shows the number for a subset of the nodes in the final trie. The column "all" shows the figure for all trie nodes. The columns "$\oplus$" and "$\ominus$" show the figure aggregated over nodes with the respective labels only. The columns "S" and "G" show the figures for the maximally specific strings and the minimally specific strings, respectively. For what concerns the query Q_{10}, there are in total 401 strings in its answer set, and together they have length 1953.

Our experimental results confirm our claim that the sets S and G constitute a compact representation of the set of all patterns satisfying the given constraints Q_M and Q_A. From Table 2, it can be seen that the total length of strings for S and G together is always smaller than that for all interesting patterns (i.e. $\oplus$). In the case of T_2, the space saving is significant. Moreover, algorithm VST is also very efficient in terms of time and space. This shows that using suffix tries in the mining of string patterns is a promising approach.

The longest pattern found (represented by the deepest node in either T_1 or T_2 having a $\oplus$ label) was "`pix umacs pix umacs pix umacs pix umacs pix umacs pix umacs pix umacs pix umacs pix umacs pix`", which has a length of 19.

6. Conclusions

We have described an approach to the general pattern discovery problem in data mining. The method is based on the decomposition of the answer set to a collection of components defined by monotonic and anti-monotonic predicates. Each of the components is a convex set or version space, the borders of which can be computed using the level wise version space algorithm or - for the pattern domain of strings - using the VSTalgorithm, which employs a novel data structure called the version space tree. Experiments have been presented that validate the approach.

The results we have presented in this paper are by no means complete, a lot of open problems and questions remain. First, it seems possible to adapt the version space trees and algorithm for use in other domains (such as itemsets). However, at present it is unclear how to do this for some more expressive domains such as Datalog queries or even the string domain where one is using a coverage relation based on subsequence matching rather than substring matching. Secondly, for the string domain, it is possible to further optimize these algorithms for specific predicates (e.g. involving frequency counting on a database of strings). Thirdly, we are at present also studying set operations on version space trees. Such operations would allow us to perform some of the logical operations directly on solution spaces. Fourthly, the framework seems also useful in the context of optimizing a sequence of inductive queries. Here, it would be interesting to see how the results to previous (sub) queries could be reused for more efficiently answering the next question.

Although there are many remaining questions, the authors hope that the introduced framework provides a sound theoretical framework for studying these open questions as well as for developing practical inductive database systems based on the idea of inductive querying.

Table 1. Summary statistics of the data

Data set (D)	number of users	number of sequences	minimum frequency (n)	frequent strings found	execution time (seconds)	memory used (bytes)
nov	55	5164	24	294	3.24	56994
exp	36	3859	80	292	2.88	88706
non	25	1906	80	293	0.72	59754
sci	52	7751	48	295	4.89	94290

Table 2. Results on finding the union of two version spaces

Suffix	Time	number of nodes					total length of strings				
Trie	(sec)	$\oplus$	$\ominus$	all	S	G	$\oplus$	$\ominus$	all	S	G
T_1	2.55	166	40	206	104	68	472	75	547	305	147
T_2	5.51	237	18	255	85	15	1489	23	1512	416	24

Acknowledgements

This work was partly supported by the European IST FET project cInQ.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. VLDB*, 1994.

[2] R. Agrawal, T. Imielinski, A. Swami. Mining association rules between sets of items in large databases. In *Proc. SIGMOD*, pp. 207-216, 1993.

[3] R. Bayardo. Efficiently mining long patterns from databases. In *Proc. SIGMOD*, 1998.

[4] L. De Raedt, S. Kramer. The level wise version space algorithm and its application to molecular fragment finding. In *Proc. IJCAI*, 2001.

[5] L. De Raedt. Query evaluation and optimisation in inductive databases using version spaces. In *Proc. EDBT Workshop on DTDM*, 2002.

[6] B. Goethals, J. Van den Bussche. On supporting interactive association rule mining. In *Proc. DAWAK*, LNCS Vol. 1874, Springer Verlag, 2000.

[7] S. Greenberg. Using unix: Collected traces of 168 users. Research Report 88/333/45, Department of Computer Science, University of Calgary, Canada, 1988.

[8] D. Gunopulos, H. Mannila, S. Saluja. Discovering All Most Specific Sentences by Randomized Algorithms. In *Proc. ICDT*, LNCS Vol. 1186, Springer Verlag, 1997.

[9] J. Han, Y. Fu, K. Koperski, W. Wang, and O. Zaiane. DMQL: A Data Mining Query Language for Relational Databases.In *Proc. SIGMOD'96 Workshop on Research Issues on Data Mining and Knowledge Discovery*, Montreal, Canada, June 1996.

[10] J. Han, L. V. S. Lakshmanan, and R. T. Ng. Constraint-Based, Multidimensional Data Mining, *Computer*, Vol. 32(8), pp. 46-50, 1999.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD*, 2000.

[12] H. Hirsh. Generalizing Version Spaces. *Machine Learning*, Vol. 17(1): 5-46 (1994).

[13] H. Hirsh. Theoretical underpinnings of versionspaes. In *Proc. IJCAI*, 1991.

[14] S. Kramer, L. De Raedt, C. Helma. Molecular Feature Mining in HIV Data. In *Proc. SIGKDD*, 2001.

[15] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery, *Data Mining and Knowledge Discovery*, Vol. 1, 1997.

[16] T. Mitchell. Generalization as Search, *Artificial Intelligence*, Vol. 18 (2), pp. 203-226, 1980.

[17] R. T. Ng, L. V.S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. SIGMOD*, 1998.

[18] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[19] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.