An Algebra for Inductive Query Evaluation

Sau Dan Lee

Luc De Raedt

Institut für Informatik
Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee, Gebäude 079
D-79110 Freiburg im Breisgau
Germany

E-mail: {danlee, deraedt}@informatik.uni-freiburg.de

Abstract

Inductive queries are queries that generate pattern sets. This paper studies properties of boolean inductive queries, i.e. queries that are boolean expressions over monotonic and anti-monotonic constraints. More specifically, we introduce and study algebraic operations on the answer sets of such queries and show how these can be used for constructing and optimizing query plans. Special attention is devoted to the dimension of the queries, i.e. the minimum number of version spaces needed to represent the answer sets. The framework has been implemented for the pattern domain of strings and experimentally validated.

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 [14]. This framework allows us to view the predicate or the constraint Q as an *inductive query* to an *inductive database system* [6]. 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.

Relational algebra has proven to be useful for database theory because of a variety of reasons. First, there is the so-called closure property, which states that the result of a query is a relation. This allows the result of one query to be used in the next query. Secondly, relational algebra allows one to reason about query execution and optimization.

Thirdly, the relational algebra relies on a simple yet powerful set of mathematical primitives. These provide inspiration for developing inductive database theories. Because inductive databases aim at putting data mining on the same methodological grounds as databases, it should be useful to develop an algebra for manipulating pattern sets.

This paper contributes an algebraic framework for manipulating pattern sets and inductive queries. It extends the theoretical framework of De Raedt *et al.* [7], who have contributed a theory centered around the class of boolean inductive queries. Such queries are boolean expressions over monotonic and anti-monotonic predicates. The present paper is an extension as we consider algebraic set operations on pattern sets and study their properties. A central concept in this theory is the notion of version space [15]. A version space is a convex set, which can be represented by its border sets (of maximally general and maximally specific elements). The solution space corresponding to a conjunctive query is a version space. Furthermore, effective algorithms exist for computing these solution sets [8, 12, 7, 3]. In this context, De Raedt *et al.* have studied the dimension of pattern sets (and boolean queries), i.e. the minimum number of version spaces needed to represent the pattern set. This in turn is related to the number of calls to a conjunctive query answering algorithm needed to answer an inductive query. In the present paper, these results are extended in the light of the algebraic operations and we also show how these results can be employed when reasoning about query execution and optimization.

Version spaces have been introduced in the context of concept-learning by Tom Mitchell [15] and have been rather popular ever since. Hirsh [11, 10] has investigated various set operations on version spaces and has shown that version spaces are closed under intersection but not under union. From this perspective, our present results extend those by Hirsh in that we show that generalized version spaces (i.e. finite unions of version spaces) are closed under set opera-

tions and that we also deduce bounds on the dimensions of the resulting sets.

Finally, as an illustration of our general framework, we apply it to the pattern domain of strings. For this domain, we have designed a data structure, called the generalized version space tree, that is useful for computing and memorizing the solution space w.r.t. inductive queries. The pattern domain of strings is of interest because of the rapid generation of biological string databases.

This paper is organized as follows. Section 2 introduces boolean inductive queries, closely following [7]. Section 3 then studies the algebraic operations, which enable us to construct many equivalent query execution plans for a given query (Section 4) and seek the optimal one. We implemented our framework on the domain of string patterns (Section 5), and performed some experiments on two data sets (Section 6). Finally, we conclude in Section 7.

2. Boolean Inductive Queries

Let us first define boolean inductive queries. We closely follow De Raedt et al. [7].

A pattern language $\mathcal{L}$ is a formal language for specifying patterns. Each pattern $\varphi \in \mathcal{L}$ matches (or covers) a set of examples φ_e , which is a subset of the universe $\mathcal{U}$ of possible examples. One pattern φ is *more general* than a pattern ψ , written $\varphi \preceq \psi$, if and only if $\varphi_e \supseteq \psi_e$. For instance, for the domain of strings over an alphabet Σ , one could choose $\mathcal{L} = \mathcal{U} = \Sigma^*$ and have that a pattern or string is covered by all its substrings in $\mathcal{L}$.

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

For illustration purposes, we introduce a number of pattern predicates inspired on MolFea [12].

- **min_freq**($\varphi, \mathcal{D}, n$) evaluates to true iff φ is a pattern that occurs in database $\mathcal{D}$ with frequency at least n , where the frequency of a pattern φ in a database $\mathcal{D}$ is the (absolute) number of data items in $\mathcal{D}$ covered by φ . Analogously, define **max_freq**($\varphi, \mathcal{D}, n$).
- **more_gen**(φ, ψ) is a predicate that evaluates to true iff pattern $\varphi \preceq \psi$. Dually, define **more_spec**.

These predicates have the form **pred**(φ, Π) or **pred**($\varphi, \mathcal{D}, \Pi$), where Π is a tuple of parameter values, $\mathcal{D}$ is a data set and φ is a pattern variable. We say that **m** is a *monotonic* predicate, if for all possible parameter values Π and all data sets $\mathcal{D}$:

$$\forall \varphi, \psi \in \mathcal{L} \text{ such that } \varphi \preceq \psi : \mathbf{m}(\varphi, \mathcal{D}, \Pi) \rightarrow \mathbf{m}(\psi, \mathcal{D}, \Pi)$$

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

A pattern predicate **pred**($\varphi, \mathcal{D}, \Pi$) that can be applied to the patterns from a language $\mathcal{L}$ defines relative to $\mathcal{D}$ the *solution set* $Th(\mathbf{pred}(\varphi, \mathcal{D}, \Pi), \mathcal{L}) = \{\phi \in \mathcal{L} \mid \mathbf{pred}(\phi, \mathcal{D}, \Pi) = \text{true}\}$. The definition of $Th(\mathbf{pred}(\varphi, \mathcal{D}, \Pi), \mathcal{L})$ 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: $Th(\neg Q, \mathcal{L}) = \mathcal{L} \setminus Th(Q, \mathcal{L})$; $Th(Q_1 \vee Q_2, \mathcal{L}) = Th(Q_1, \mathcal{L}) \cup Th(Q_2, \mathcal{L})$; $Th(Q_1 \wedge Q_2, \mathcal{L}) = Th(Q_1, \mathcal{L}) \cap Th(Q_2, \mathcal{L})$. The predicates that appear in Q may reference one or more data sets $\mathcal{D}_1, \dots, \mathcal{D}_n$.

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. This is the *boolean inductive query evaluation problem* :

Given

- a language $\mathcal{L}$ of patterns,
- a set of monotonic predicates $\mathcal{M} = \{\mathbf{m}_1(p, \mathcal{D}_1, \Pi_1), \dots, \mathbf{m}_n(p, \mathcal{D}_j, \Pi_j)\}$ and a set of anti-monotonic predicates $\mathcal{A} = \{\mathbf{a}_1(p, \mathcal{D}_1, \Pi_1), \dots, \mathbf{a}_k(p, \mathcal{D}_k, \Pi_k)\}$
- a query Q that is a boolean expression over the predicates in $\mathcal{M}$ and $\mathcal{A}$.

Find the set of patterns $Th(Q, \mathcal{D}, \mathcal{L})$, i.e. the solution set of the query Q in the language $\mathcal{L}$ with respect to the data subsets $\mathcal{D}_1, \dots, \mathcal{D}_n \subseteq \mathcal{D}$.

De Raedt *et al.* have proposed a strategy for evaluating inductive queries and also a first step in the direction of query optimization. Their strategy consists of decomposing a boolean query Q into k sub-queries Q_i such that Q is equivalent to $Q_1 \vee \dots \vee Q_k$, k is minimal and each of the Q_i is the conjunction of a monotonic and an anti-monotonic subquery $Q_{m,i} \wedge Q_{a,i}$. Notice that $Q_{m,i}$ and $Q_{a,i}$ may be boolean expressions themselves. Indeed, [7] show that $Q_{m,i}$ may be a DNF formula in which all the literals are monotonic, and similarly for $Q_{a,i}$. A DNF formula of the form $M_1 \vee \dots \vee M_n$ where each of the M_i is a conjunction of monotonic predicates is monotonic. The reason is that the conjunction of monotonic predicates is monotonic (hence all of the M_i are), and also the disjunction of monotonic predicates is monotonic as well (and hence $Q_{m,i}$ is). The query evaluation strategy proposed by [7] first decomposes the query and then computes $Th(Q, \mathcal{D}, \mathcal{L})$ as $\cup_i Th(Q_i, \mathcal{D}, \mathcal{L})$. Because 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), it can be efficiently computed for a wide class of pattern languages $\mathcal{L}$, and queries Q_i , cf. [12, 3]. Furthermore, the number of calls to such an algorithm is minimized

because in the decomposition process, one minimizes the number k of subqueries. This is the so-called dimension of a query, which [7] introduced as follows:

Definition 1 Let $\mathcal{L}$ be a pattern language, and $X \subseteq \mathcal{L}$. Then X is a version space of dimension 1, if $\forall \varphi, \varphi', \psi \in \mathcal{L} : \varphi \preceq \psi \preceq \varphi'$ and $\varphi, \varphi' \in X \implies \psi \in X$. The set of all version spaces of dimension 1 for $\mathcal{L}$ is denoted $\mathcal{VS}^1(\mathcal{L})$.

Definition 2 The subset of patterns in $\mathcal{L}$ that satisfy a boolean query Q , composed of monotonic or anti-monotonic predicates, is a generalized version space, denoted by $Th(Q, \mathcal{D}, \mathcal{L})$. The set of all generalized version spaces for $\mathcal{L}$ is denoted by $\mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$. Note that $\mathcal{VS}^1(\mathcal{L}) \subseteq \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$.

Definition 3 The set $X \in \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$ has dimension k if it is the union of k sets in $\mathcal{VS}^1$, but not the union of $k - 1$ sets in $\mathcal{VS}^1$. 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 [15] or a convex space [10]. 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.

With the following lemma, De Raedt *et al.* [7] provided an alternative characterization of dimension k sets.

Lemma 4 Let $X \subseteq \mathcal{L}$. Call a chain $\varphi_1 \preceq \rho_1 \preceq \varphi_2 \preceq \rho_2 \preceq \dots \preceq \varphi_{k-1} \preceq \rho_{k-1} \preceq \varphi_k$ an alternating chain (of length k) for X if $\varphi_i \in X$ ($i = 1, \dots, k$) and $\rho_j \notin X$ ($j = 1, \dots, k - 1$). Then the dimension of X is equal to the maximal k for which there exists in $\mathcal{L}$ an alternating chain of length k (or a k -chain for short) for X .

Example 5 Consider the following queries:

$$\begin{aligned} Q_3 &= \text{more_gen}(\varphi, abc) \wedge \text{more_spec}(\varphi, a) \\ Q_4 &= \text{more_gen}(\varphi, c) \\ Q_5 &= Q_3 \vee Q_4 \end{aligned}$$

Then c, bc, abc is an alternating chain of length 2 for $Th(Q_5, \mathcal{L}_{\Sigma})$.

The dimension of an inductive query is an important concept because it corresponds to the minimum number of calls to an algorithm for computing a convex solution space.

3. Operations on Solution Spaces

Consider two generalized version spaces $V, W \in \mathcal{VS}^{\mathbb{Z}}$. We would like to know how we can combine them using

the usual set operations. This is of interest because the solution set to an inductive query is, by definition, obtained as the result of applying set operations on solution set of simpler queries. At the same time, these operations promise to be useful when constructing query plans and in interactive querying sessions (cf. Section 4 for a detailed discussion).

We will analyze the operations: intersection ($\cap$), union ($\cup$), and complement ($'$) w.r.t. $\mathcal{L}$. Set difference can be treated as $V \setminus W \equiv V \cap W'$ whereas symmetric difference can be interpreted as $(V \setminus W) \cup (W \setminus V)$.

When $\dim(V) = \dim(W)$ are both 1, this reduces to traditional version spaces $\mathcal{VS}^1$. We know from the previous discussion that $V \cap W$ is in $\mathcal{VS}^1$. We have also shown above by a counter-example that the dimension of the union can be 2, although it may also be 1 (e.g. when $V \subseteq W$). So, traditional version spaces, i.e. $\mathcal{VS}^1$, are not closed under union (as shown by [11]).

Nevertheless, our extension $\mathcal{VS}^{\mathbb{Z}}$ is closed under the usual set operations. The following theorems thus generalize Hirsh's results to the case where solution sets are represented by a finite number of version spaces.

Theorem 6 $\mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$ is closed under the following set operations: intersection, union, complement, difference and symmetric difference.

It suffices to analyze the first 3 of them, as the latter can be defined out of the first 3 operations. The proofs are given below together with bounds for the resulting dimensions.

Theorem 7 Let $V, W \in \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$. Then, $U = V \cap W \in \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$ and $0 \leq \dim(U) \leq \dim(V) + \dim(W) - 1$.

Proof: Let $v = \dim(V)$ and $w = \dim(W)$. By Definition 3, we can write $V = \bigcup_{i=1}^v X_i$ and $W = \bigcup_{k=1}^w Y_k$ for some $X_i, Y_k \in \mathcal{VS}^1(\mathcal{L})$. Then, we have $U = (\bigcup_{i=1}^v X_i) \cap (\bigcup_{k=1}^w Y_k) = \bigcup_{k=1, \dots, w} Z_{i,k}$ where $Z_{i,k} = X_i \cap Y_k$. Since every $Z_{i,k} \in \mathcal{VS}^1(\mathcal{L})$, we have $U \in \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$.

For the bounds, in case V and W are disjoint, U is empty and hence, $\dim(U) = 0$, a lower bound. For the upper bound, we provide a proof by contradiction. Assume that $U = V \cap W$ has dimension at least $u = v + w$. Then, by Lemma 4, U has a u -chain, i.e.

$$\begin{aligned} \exists \varphi_i \in U (i = 1, \dots, u) \exists \rho_j \in \mathcal{L} \setminus U (j = 1, \dots, u - 1) \\ \varphi_1 \preceq \rho_1 \preceq \varphi_2 \preceq \rho_2 \preceq \dots \preceq \rho_{u-1} \preceq \varphi_u \end{aligned}$$

Denote the sequence $\{\rho_j\}_{j=1}^{u-1}$ by R . Observe now that at most $v - 1$ members of R do not belong to V (Otherwise, if there would be at least v such members, one can use these to construct a $v + 1$ chain for V , which would imply $\dim(V) \geq v + 1$, which contradicts our assumption.) This means that at least $(u - 1) - (v - 1) = w$ members of R must belong to V . They form a subsequence

$R_V = \{\rho_{p_r}\}_{r=1}^p$ of R , where $\{p_r\}_{r=1}^p$ is an increasing sequence on $\{0, \dots, u-1\}$ and $p \geq w$.

By a similar argument, at least v members of R must belong to W . They form a subsequence $R_W = \{\rho_{q_s}\}_{s=1}^q$ of R . Now, R_V and R_W together have at least $v + w = u$ members, all coming from R . But R has only $u - 1$ members. By the pigeon hole principle, at least one member of R is shared by R_W and R_V . Let us call this element ρ' . According to the definition of R_W and R_V , $\rho' \in V \cap W = U$. But this contradicts with the fact that $\rho_j \in \mathcal{L} \setminus U$ for all j ■

Theorem 8 Let $V, W \in \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$. Then, $U = V \cup W \in \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$ and $0 \leq \dim(U) \leq \dim(V) + \dim(W)$.

Proof: Let $v = \dim(V)$ and $w = \dim(W)$. By Definition 3, we can write $V = \bigcup_{i=1}^v X_i$ and $W = \bigcup_{k=1}^w Y_k$ for some $X_i, Y_k \in \mathcal{VS}^1(\mathcal{L})$. Then, we have $U = (\bigcup_{i=1}^v X_i) \cup (\bigcup_{k=1}^w Y_k) = \bigcup_{l=1}^{v+w} Z_l$ where $Z_l = X_l$ for $l = 1, \dots, v$ and $Z_{l-v} = Y_{l-v}$ for $l = v+1, \dots, v+w$. Since each $Z_k \in \mathcal{VS}^1(\mathcal{L})$, we have $U \in \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$. Moreover, $\dim(U) \leq w + v$ by Definition 3. We cannot give a tighter lower bound other than 0, because of the following special case. Consider the case when both V and W are empty. This gives an empty U , and hence $\dim(U) = 0$. Therefore, $0 \leq \dim(U) \leq \dim(V) + \dim(W)$. ■

Theorem 9 Let $V \in \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$ and $v = \dim(V)$. Then, $U = \mathcal{L} \setminus V \in \mathcal{VS}^{\mathbb{Z}}(\mathcal{L})$ and $u = \dim(U) = v-1, v$ or $v+1$.

Proof: There is a special case: $V = \emptyset$. In this case $\dim(V) = 0$, and $\mathcal{L} \setminus V = \mathcal{L}$, which has dimension 1. So, the theorem holds.

For the general case, observe first that $u \geq v-1$. Indeed, by Lemma 4, V has a v -chain. i.e.

$$\exists \varphi_i \in V (i = 1, \dots, v) \exists \rho_j \in \mathcal{L} \setminus V (j = 1, \dots, v-1) \\ \varphi_1 \preceq \rho_1 \preceq \varphi_2 \preceq \rho_2 \preceq \dots \preceq \varphi_{v-1} \preceq \rho_{v-1} \preceq \varphi_v$$

Then, $\rho_1 \preceq \varphi_2 \preceq \rho_2 \preceq \dots \preceq \varphi_{v-1} \preceq \rho_{v-1}$ constitutes a $(v-1)$ -chain of $U = \mathcal{L} \setminus V$. Thus, with Lemma 4, we may conclude that $\dim(U) = u \geq v-1$. Similarly, one can derive that $\dim(V) = v \geq u-1$, or equivalently $u \leq v+1$. Combining these results gives $v-1 \leq u \leq v+1$. The result then follows from the fact that u is an integer. ■

4. Query Plans and Algebraic Operations

By the definition of boolean inductive queries, the solution set $Th(Q, \mathcal{D}, \mathcal{L})$ of a query Q is obtained by applying algebraic operations on the solution sets w.r.t. the underlying queries. This can be formalized using the notion of a query plan.

Definition 10 A query plan is a boolean formula with some of its subqueries marked using the symbol $\underbrace{}$. Furthermore, all marked subqueries are the conjunction of a monotonic and an anti-monotonic subquery.

Example 11 Consider the query $Q = (a_1 \wedge m_1) \vee (a_1 \wedge m_2) \vee (a_2 \wedge m_1) \vee (a_2 \wedge m_2)$ where the a_i and m_i are anti-monotonic (resp. monotonic) predicates. The solution set of this query can be obtained by first computing the solution sets to the queries $(a_1 \wedge m_1)$, $(a_1 \wedge m_2)$, $(a_2 \wedge m_1)$ and $(a_2 \wedge m_2)$ and then taking their union. This way of computing $Th(Q, \mathcal{D}, \mathcal{L})$ corresponds to the plan

$$\underbrace{(a_1 \wedge m_1)} \vee \underbrace{(a_1 \wedge m_2)} \vee \underbrace{(a_2 \wedge m_1)} \vee \underbrace{(a_2 \wedge m_2)}$$

Alternatively, one could rewrite query Q into the form $\underbrace{(a_1 \vee a_2) \wedge (m_1 \vee m_2)}$ and obtain the result using one call to a conjunctive query solver (i.e. an algorithm or system that computes the set of all solutions to a conjunctive query). Intermediate forms are also possible.

For any inductive query Q , one can now construct a variety of different query plans by annotating queries that are logically equivalent to Q . The question then arises as to which query plan is optimal, in the sense that the resources (i.e. memory and cpu-time) needed for computing its solution set are as small as possible. A general approach to this problem would involve the use of cost estimates that for each call to a conjunctive solver and operation. One example of a cost function for a call to a conjunctive solver could be *Expected Number of Scans of Data* $\times$ *Size of Data Set*. Another one could be the *Expected Number of Covers Tests*. De Raedt *et al.* have studied (and solved) the query optimization problem under the assumption that each call to a conjunctive solver has unit cost and that the only set operation allowed is union. Under this assumption, decomposing a query Q into k subqueries of the form $Q_{a,i} \wedge Q_{m,i}$ with $(Q_{a,i}$ anti-monotonic and $Q_{m,i}$ monotonic) and $\dim(Q) = k$ is an optimal strategy. In this paper, we will leave open the challenging question as to which cost-estimates to use in practice. However, what should be clear is that given such cost-estimates, one could optimize inductive queries by constructing all possible query plans and then selecting the best one. This is effectively an optimization problem, not unlike the query optimization problem in relational databases.

The optimization problem becomes even more interesting in the light of interactive querying sessions [2], which should be quite common when working with inductive databases. In such sessions, one typically submits a rough query to get some insight in the domain, and when the results of this query are available, the user studies the results and refines the query. This often goes through a few iterations until the desired results are obtained.

These interactive sessions are again similar in spirit to those in traditional databases, where the results of intermediate queries are often cached for optimizing later queries. Indeed, consider that for the above example, we would already have the result of the query $(a_1 \vee a_2) \wedge m_1$. In this

case, one could actually employ the following query plan:

$$\underbrace{((a_1 \vee a_2) \wedge m_1)} \vee \underbrace{((a_1 \vee a_2) \wedge m_2)}$$

where $\dots$ denotes a query for which the solution is readily available. This query plan employs a union and could save a lot of time when m_1 is a minimum frequency query in a large database.

Even when the refined query Q_1 is not related to the original one Q_0 via Boolean algebra, there is still a possibility for reusing the cached results $Th(Q_0, \mathcal{D}, \mathcal{L})$ to obtain the new solution $Th(Q_1, \mathcal{D}, \mathcal{L})$ efficiently. One such situation occurs with the minimum frequency constraints. For example, $Q_0 = \text{min_freq}(\varphi, \mathcal{D}, \theta_1)$ and $Q_1 = \text{min_freq}(\varphi, \mathcal{D}, \theta_2)$. When $\theta_2 \geq \theta_1$, then $Th(Q_1, \mathcal{D}, \mathcal{L}) \subseteq Th(Q_0, \mathcal{D}, \mathcal{L})$. So, we can find out the answer $Th(Q_1, \mathcal{D}, \mathcal{L})$ by simply performing a filtering operation on $Th(Q_0, \mathcal{D}, \mathcal{L})$. If the frequencies of all patterns in $Th(Q_0, \mathcal{D}, \mathcal{L})$ were stored with the results, we can even do this filtering without scanning the database! On the other hand, when $\theta_2 \leq \theta_1$, a simple filtering does not work. In this case, we can make use of an incremental update algorithm [4, 13], with adaptations, to efficiently compute $Th(Q_1, \mathcal{D}, \mathcal{L})$ by using the cached results of $Th(Q_0, \mathcal{D}, \mathcal{L})$.

The problem boils down to recognizing the parts of queries Q_0 and Q_1 which are related this way. For simple queries, such recognition is simple. However, for more complicated queries, it becomes challenging to recognize such related parts of the queries. Our algebraic framework allows one to break down a complicated query into a disjunctive normal form, making it easier to recognize such related parts for optimization. Moreover, it is possible to rewrite the queries Q_0 and Q_1 into many equivalent forms, and we can then devise algorithms to recognize the related parts in these forms. This provides further opportunities for optimizations in data mining.

5. Generalized Version Space Trees

We have extended our previous data structure “Version Space Trees” [7] for mining $\mathcal{VS}^1$ of strings to handle the general case of $\mathcal{VS}^Z$. The extended data structure is called *Generalized Version Space Trees* (GVS Tree).

The GVS Tree maintains a set of strings which are patterns we are discovering from the database. Each such string is represented by a node in the tree. For any given node n , we denote the string pattern it represents by $s(n)$. The organization of the tree is based on suffix tries. A *suffix trie* is a trie with the following properties:

- For each node n in the trie, 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 as a *suffix link*, $\text{suffix}(n) = n'$, such that $s(n')$ is obtained from $s(n)$ by dropping the first character. The root node is special because it represents ϵ , which has no suffixes. We define $\text{suffix}(\text{root}) = \Omega$, where Ω denotes a unique fake node.

Unlike the common approach in the literature on suffix trees [16, 17], we use suffix tries in two very different ways from the main stream. The first one is that instead of building a suffix tree on all the suffixes of a *single* string, we are indexing all the suffixes of a *set of strings patterns* for a string database $\mathcal{D}$. This means multiple strings are stored in the tree. Moreover, in parts of our algorithms, we even keep a count of occurrences of each such substring in the corresponding node.

We label each node n of the GVS Tree with either a “ $\oplus$ ” or a “ $\ominus$ ”, according to whether or not the $s(n) \in Th(Q, \mathcal{D}, \Sigma^*) \in \mathcal{VS}^Z(\Sigma^*)$. In our previous publication [7], the VS Tree had a restriction that there can be at most one sign change the root to any leaf. This is because VS Tree was designed to model sets in $\mathcal{VS}^1$ only. As a generalization in this current work, we have removed this restriction, allowing complete freedom on the assignment of the labels “ $\oplus$ ” or a “ $\ominus$ ” to any node. As a result, a GVS Tree can represent sets of string patterns in $\mathcal{VS}^Z$. Observe that if $\dim(V) = n$, then there exists a path (exploiting both child and suffix links, and ignoring the link directions) in T with alternating signs so that the number of sign changes from $\ominus$ to $\oplus$ is n (Lemma 4).

Figure 1 shows a labelled GVS Tree for the set $Th(Q, \mathcal{D}, \Sigma^*)$ where $\Sigma = \{a, b, c, d\}$ and $Q(\varphi, \mathcal{D}) = (bc \preceq \varphi \preceq abcd) \vee (a \preceq \varphi \preceq acd)$. The dashed, curved arrows show the suffix links. The suffix links of the nodes immediate below the root node all points back to the root node, and are omitted for clarity. The $\oplus$ nodes represent the seven members of the GVS, namely a , abc , $abcd$, ac , acd , bc and bcd . Note that Q above is already in a minimal disjunctive normal form. So, the GVS has a dimension of 2, and the path through the nodes representing ϵ , a , ab , abc with exactly 2 sign changes.

An important property of the GVS Tree is that checking for membership is very efficient. Given any string $\varphi \in \Sigma^*$, we just need to follow the symbols on φ and descend through the tree accordingly. We will then end up at a node n so that $s(n) = \varphi$. This node has an $\oplus$ mark if and only if φ is in the GVS. The time complexity is $O(|\varphi|)$. The space complexity of a GVS Tree and the time required to build it is the same of that of suffix tries—quadratic in the size of the input strings.

The *TreeMerge* algorithm is basically an algorithm for merging two ordinary trees. However, we have to combine the flags ($\ominus$ or $\oplus$) from both trees, too. The combining algorithm is presented in the pseudocode in Algorithm 1 as an abstract Boolean operation $\odot$. When the operation is “and”,

pears in multiple groups. However, within each group, a sequence appears at most once. Three of these groups were used in the experiments. They are given in Table 1(b).

For both databases, we used a query of the following form and ran our programs to mine the patterns satisfying the constraints.

$$\begin{aligned} Q &= (Q_1 \vee Q_2) \wedge Q_A \text{ where} \\ Q_1 &= \text{min_freq}(\varphi, D_1, |D_1| \times \theta_1) \wedge \text{len_lb}(\varphi, 3) \\ Q_2 &= \text{min_freq}(\varphi, D_2, |D_2| \times \theta_2) \wedge \text{len_lb}(\varphi, 2) \end{aligned}$$

and Q_A is an anti-monotonic constraint. The monotonic predicate $\text{len_lb}(\varphi, n)$ evaluates to true iff the length of φ is at least n . Here, D_1 and D_2 are subsets of the database being used, θ_1 and θ_2 are the corresponding minimum frequency thresholds, also set to 10%, and Q_A is an anti-monotonic constraint.

Note that Q is neither monotonic nor anti-monotonic. So, the solution set is in $\mathcal{VS}^Z$. However, Q_1 and Q_2 are each a conjunction of an anti-monotonic constraint (minimum frequency) and a monotonic one (minimum length). So, $\dim(Q_1) = \dim(Q_2) = \dim(Q_A) = 1$. Thus, one (straight-forward) strategy to find the set of patterns satisfying Q is to use the query plan

$$\left(\underbrace{Q_1 \vee Q_2} \right) \wedge \underbrace{Q_A}$$

This involves 3 invocations of our frequent pattern miner VST. However, the VST algorithm (or any other frequent pattern discovery algorithm such as Apriori [1]) is the most time-consuming part of the whole processes. We would like to minimize this cost. This is possible now with our algebra on generalized version spaces and theorems on the dimension. By Theorem 8, we know that $\dim(Q_1 \vee Q_2) \leq \dim(Q_1) + \dim(Q_2) = 1 + 1 = 2$. Now, applying Theorem 7, we have $\dim(Q) \leq \dim(Q_1 \vee Q_2) + \dim(Q_A) - 1 \leq 2 + 1 - 1 = 2$. Thus, Q has a dimension of at most 2. Thus, one can express Q as the union of two version spaces.

Indeed, we can obtain a different query plan for Q , which we denote by Q' :

$$\begin{aligned} Q' &= \underbrace{Q'_1 \vee Q'_2} \text{ where} \\ Q'_1 &= Q_1 \wedge Q_A \\ Q'_2 &= Q_2 \wedge Q_A \end{aligned}$$

This query plan involves only 2 invocations of algorithm VST. It is thus expected to be faster. Moreover, having pushed the anti-monotonic constraint Q_A deeper into the query evaluation, we expect the levelwise algorithm VST to prune more effectively.

To verify this, we have run experiments on the two databases with the values for D_1 , D_2 and Q_A as shown in Table 2. On the unix command database, this query translates to “What sequences of unix commands are used often

	Unix command database	Yeast database
D_1	experienced programmers	cat30
D_2	computer scientists	cat40
θ_1	10%	20%
θ_2	10%	20%
Q_A	$\text{min_freq}(\varphi, \text{non}, \text{non} \times 10\%) \wedge \text{min_freq}(\varphi, \text{nov}, \text{nov} \times 10\%)$	$\text{min_freq}(\varphi, \text{cat2}, \text{cat2} \times 20\%)$

Table 2. Queries used for the experiments

Query Strategy	No. of patterns		Time (sec.)	
	Unix	Yeast	Unix	Yeast
Q (total)	41	404	2.15	4.03
Q_1	110	638	0.74	0.42
Q_2	212	122	1.18	3.34
Q_A	67	434	0.23	0.27
Q' (total)	41	404	1.60	3.65
Q'_1	16	403	0.65	0.55
Q'_2	40	38	0.95	3.10

Table 3. Experimental Results

by experienced programmers with a length of at least 3 or by computer scientists with a length of at least 2, and are also frequently used by the other two groups of users?”. With our algebraic framework, it is possible to perform data mining with such complicated constraints. The query for the protein database translates to “What amino acid sequences occur frequently in function category ‘cat30’ with a length of at least 3 or in function category ‘cat40’ with a length of at least 2, and is at the same time frequent among the function category ‘cat2’?”.

The queries Q and Q' are evaluated as described above using our implementation of the VST and TreeMerge algorithms. For each database the resulting patterns for both queries are compared and found to be identical. This verifies the correctness of our theory and implementation.

The time taken are noted and given in Table 3. With the unix command database, it took 2.15 seconds to evaluate the query as Q and only 1.60 seconds to evaluate as Q' . With the yeast database, it took 4.03 and 3.65 seconds, respectively. It is thus 9–26% faster to use strategy Q' than Q to find out the set of patterns. The table also shows a breakdown of the time taken for evaluating the queries Q_1 , Q_2 , Q_A , Q'_1 and Q'_2 . The pattern sets for these are all in $\mathcal{VS}^1$, and are computed by the VST algorithm. It should be noted that the time taken for the TreeMerge algorithm is negligible (less than 1 ms). This confirms our claim that invoca-

Query Strategy	No. of patterns		Heap memory (bytes)	
	Unix	Yeast	Unix	Yeast
Q (total)	41	404	128619	119988
Q' (total)	41	404	66740	106476

Table 4. Memory usage

tions of algorithm VST is the most time-consuming part of the whole process.

Another important observation in Table 3 is the number of patterns found for each query strategy and subqueries. In query strategy Q' , the constraint Q_A is pushed down to the subqueries Q'_1 and Q'_2 , effectively pruning the number of patterns that needs to be processed by the programs. This accounts for the improved speed and memory usage.

Not only is time saved, but also is memory more efficiently used when we use strategy Q' instead of Q to compute the set of patterns in question. The amount of heap memory used by our programs were recorded. The maximum amount of heap memory usage is shown in Table 4. Using query evaluation strategy Q' , we save 11–48% of memory. Thus, it saves both time and memory to evaluate the query using Q' .

7. Conclusions

We have generalized the notion of convex sets or version spaces to represent sets of higher dimensions. These generalized version spaces are useful for representing the solution sets to boolean inductive queries. Furthermore, we have studied the effect of algebraic operations on such generalized version spaces and shown that these generalized version spaces are closed under the set operations. This generalizes Hirsh's results on traditional version spaces (sets of dimension 1).

We have also shown how the resulting algebraic framework can be employed for query planning and optimization. The framework has been implemented for the pattern domain of strings and experimental results that illustrate the use of the framework have been presented.

Nevertheless, there are many remaining opportunities for further research. Most important is the development of effective and realistic cost functions for inductive query evaluation and their use in query optimization.

Acknowledgements This work was partly supported by the EU IST project cInQ (Consortium on Inductive Querying). The authors would also like to thank Manfred Jaeger and Heikki Mannila, with whom they developed the original theory of inductive querying [7] that is extended in the present paper, as well as Ross D. King and Amanda Clare for providing the yeast data.

References

- [1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th VLDB*, pages 487–499. Morgan Kaufmann, 1994.
- [2] E. Baralis and G. Psaila. Incremental refinement of mining queries. In *Proc. 1st DaWaK*, LNCS 1676, pages 173–182, Florence, Italy, August 30–September 1 1999. Springer.
- [3] C. Bucila, J. Gehrke, D. Kifer, and W. White. DualMiner: A dual-pruning algorithm for itemsets with constraints. In *Proc. 8th ACM SIGKDD*, Edmonton, Alberta, Canada, July 23–26 2002.
- [4] D. W. L. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. 5th DASFAA*, pages 185–194, Melbourne, Australia, 1–4 Apr. 1997.
- [5] A. Clare and R. D. King. Machine learning of functional class from phenotype data. *Bioinformatics*, 18(1):160–166, 2002.
- [6] L. De Raedt. A perspective on inductive databases. *SIGKDD Explorations*, 4(2):69–77, January 2003.
- [7] L. De Raedt, M. Jaeger, S. D. Lee, and H. Mannila. A theory of inductive query answering (extended abstract). In *Proc. 2nd IEEE ICDM*, pages 123–130, Maebashi, Japan, December 9–12 2002.
- [8] L. De Raedt and S. Kramer. The levelwise version space algorithm and its application to molecular fragment finding. In *Proc. 17th IJCAI*, August 4–10 2001.
- [9] S. Greenberg. Using unix: Collected traces of 168 users. Research Report 88/333/45, Department of Computer Science, University of Calgary, Canada, 1988.
- [10] H. Hirsh. Theoretical underpinnings of version spaces. In *Proc. 12th IJCAI*, pages 665–670. Morgan Kaufmann Publishers, 1991.
- [11] H. Hirsh. Generalizing version spaces. *Machine Learning*, 17(1):5–46, 1994.
- [12] S. Kramer, L. De Raedt, and C. Helma. Molecular feature mining in hiv data. In *Proc. 7th ACM SIGKDD*, Association for Computing Machinery, August 26–29 2001. ISBN: 158113391X.
- [13] S. D. Lee and D. Cheung. *Maintenance of Discovered Association Rules*, volume 600 of *The Kluwer International Series in Engineering and Computer Science*, chapter 8. Kluwer Academic Publishers, Boston, November 2000. ISBN-0-7923-7243-3.
- [14] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.
- [15] T. M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, March 1982.
- [16] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [17] P. Weiner. Linear pattern matching algorithm. In *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.