Mining Tree Queries in a Graph

Bart Goethals
Universiteit Antwerpen

Eveline Hoekx
Universiteit Hasselt

Jan Van den Bussche
Universiteit Hasselt

ABSTRACT

We present an algorithm for mining tree-shaped patterns in a large graph. Novel about our class of patterns is that they can contain constants, and can contain existential nodes which are not counted when determining the number of occurrences of the pattern in the graph. Our algorithm has a number of provable optimality properties, which are based on the theory of conjunctive database queries. We propose a database-oriented implementation in SQL, and report upon some initial experimental results obtained with our implementation on graph data about food webs, about protein interactions, and about citation analysis.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms

Algorithms, Performance, Experimentation

Keywords

Canonical form, conjunctive query, equivalence checking, graph, levelwise, redundancy checking, SQL, tree query

1. INTRODUCTION

The problem of mining patterns in graph-structured data has received considerable attention in recent years, as it has many interesting applications in such diverse areas as biology, the life sciences, the World Wide Web, or social sciences. Past work in this area falls in two major categories, not to be confused:

1. In the "transactional" category, e.g., [1, 2, 3, 4, 5, 6], the dataset is a set of small graphs which we call transactions, and the task is to discover patterns that occur at least once in a sufficient number of transactions or examples. (Approaches from machine learning or inductive logic programming usually call the small graphs "examples" instead of transactions.)

2. In the "single graph" category, e.g., [7, 8, 9, 10, 11], the dataset is a single large graph, and the task is to discover patterns that occur sufficiently often in the dataset. Note that the transactional case can be simulated by the single-graph case, but the converse is not obvious.

The present work falls in the single-graph category. Past work in this category has mainly focused on patterns in the form of subgraphs that occur in sufficiently many edge-disjoint isomorphic copies in the graph that is being mined. We propose a rather different notion of pattern, which is still a graph, but with the following features:

- Patterns may have "existential" nodes: any occurrence of the pattern must have a copy of such a node, but existential nodes are not counted when determining the number of occurrences.

- Moreover, patterns may have "selected" nodes, labeled by constants, which must map to fixed designated nodes of the graph. Past work has dealt with node labels, but only with non-unique ones: such labels are easily simulated by constants, but the converse is not obvious. Also edge labels can be simulated using constants. (To simulate a node label a, add a special node a, and express that node x has label a by drawing an edge from x to a. For an edge $x \to y$ labeled b, introduce an intermediate node $x.y$ with $x \to x.y \to y$, and label node $x.y$ by b.)

- An "occurrence" of the pattern in G is defined as any homomorphism from the pattern in G. When counting the number of occurrences, two occurrences that differ only on existential nodes are identified.

A simple example of a pattern is shown in Figure 1; when applied to a food web ("who eats who"), it describes all organisms x that compete with organism #8 for some organism as food, that itself feeds on organism #0. This pattern has one existential node, two selected nodes, and one distinguished node x.

Effectively, what we want to mine are what is known in database research as *conjunctive queries* [12, 13]; these are the queries we could pose to the graph (stored as a two-column table) in the core fragment of SQL where we do not use aggregates or subqueries, and use only conjunctions of

KDD'05, August 21–24, 2005, Chicago, Illinois, USA.

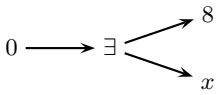


Figure 1: Simple example of a pattern with two selected nodes, one existential node, and one distinguished node x.

equality comparisons as where-conditions. For example, the pattern of Figure 1 amounts to the following SQL query on a table `G(from,to)`:

```
select distinct G3.to as x
from G G1, G G2, G G3
where G1.from=0 and G1.to=G2.from
  and G2.to=8 and G3.from=G2.from
```

We will present an algorithm for mining conjunctive-query patterns that return sufficiently many answers on a given graph. Our algorithm has the following properties:

1. We restrict to patterns that are trees, such as the example in Figure 1. Tree patterns have formed an important special case in the transactional literature [4, 14], but have not yet received special attention in the single-graph literature. Note that the graph that is being mined is not restricted in any way.

2. The algorithm is incremental in the number of nodes of the pattern. We generate unordered, rooted trees of increasing sizes, avoiding the generation of isomorphic duplicates. It is well known how to do this efficiently [15, 16, 4, 14]. Note that this generation of trees is in no way "levelwise" [17]. Indeed, under the way we count pattern occurrences, a subgraph of a pattern might be less frequent than the pattern itself (this was already pointed out by Kuramochi and Karypis [10]). So, our algorithm systematically considers ever larger trees, and can be stopped any time it has run long enough or has produced enough results. Our algorithm does not need any space beyond what is needed to store the mining results.

3. For each tree, all conjunctive queries based on that tree are generated. Here, we do work in a levelwise fashion. This aspect of our algorithm has clear similarities with "query flocks" [18]. A query flock is a user-specified conjunctive query, in which some constants are left unspecified and viewed as parameters. A levelwise algorithm was proposed for mining all instantiations of the parameters under which the resulting query returns enough answers. We push that approach further by also mining the query flocks themselves. Consequently, the specialization relation on queries used to guide the levelwise search is quite different in our approach.

4. A query based on some tree may be equivalent to a query based on a previously seen tree. Furthermore, two queries based on the same tree may be equivalent. We carefully and efficiently avoid the counting of equivalent queries, by using and adapting what is known from the theory of conjunctive database queries.

5. Last but not least, our algorithm naturally suggests a database-oriented implementation in SQL. This is useful for several reasons. First, the number of discovered patterns can be quite large, and it is important to keep them available in a persistent and structured manner, so that they can be browsed easily, and so that *association rules* can be derived efficiently. Moreover, we will show how the use of SQL allows us to generate and check large numbers of similar patterns in parallel, taking advantage of the query processing optimizations provided by modern relational database systems. Third, a database-oriented implementation does not require us to move the dataset out of the database before it can be mined. In classical itemset mining, database-oriented implementations have received serious attention [18, 19], but less so in graph mining, a recent exception being an implementation in SQL of the seminal SUBDUE algorithm [20].

The primary purpose of this paper is to present our algorithm; concrete applications to discover new knowledge about real-life scientific datasets is a topic of planned future work. Yet, using a prototype implementation, we will already demonstrate here that our approach is feasible, by showing some concrete results mined from a food web, a protein interactions graph, and a citation graph. We also give performance figures on random graphs.

2. MINING TREE-QUERY PATTERNS

In this section, we define the problem formally, and describe our overall approach.

Graph-theoretic preliminaries. Let N be any finite set of *nodes*; nodes can be any data objects such as numbers or strings. For our purposes, we define a (directed) *graph* on N as a subset of N^2, i.e., as a finite set of ordered pairs of nodes. These pairs are called *edges*. We assume familiarity with the notion of a *tree* as a special kind of graph, and with standard graph-theoretic concepts such as *root* of a tree; *children*, *descendants*, *parent*, and *ancestors* of a node; and *path* in a graph. Any good algorithms textbook will supply the necessary background.

Tree queries. A *tree query* is a tree Q whose nodes are called *variables*, where additionally:

1. Some variables may be marked as being *existential*;

2. Some other variables may be labeled with a data constant. These variables are called *selected nodes*.

The nodes of Q that are neither existential nor selected are called the *distinguished variables* of Q.

When we draw a tree query, as in Figure 1, we write each distinguished variable down as x, with different subscripts when there are multiple such variables; we depict existential nodes by the symbol $\exists$; and we depict selected nodes by writing down their label.

Matchings. Recall that a *homomorphism* from a graph G_1 to a graph G_2 is a mapping h from the nodes of G_1 to the nodes of G_2 that preserves edges, i.e., if $(i, j) \in G_1$ then $(h(i), h(j)) \in G_2$.

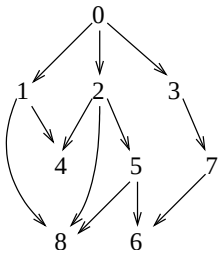


Figure 2: In this example graph, the pattern of Figure 1 has frequency three, as the distinguished node x can be mapped to the three different nodes 4, 5, and 8.

We now define a *matching* of a query Q in a graph G simply as a homomorphism h from the underlying tree of Q to G, with the constraint that for any selected node z, if z is labeled a, then $h(z) = a$.

Furthermore, we define the *frequency* of Q in G as the number of matchings of Q in G, with the important provision that *we identify any two matchings that agree on the distinguished variables*. Indeed, two matchings that differ only on the existential nodes need not be distinguished, as this is precisely the intended semantics of existential nodes. Note that we do not need to worry about selected nodes, as all matchings will agree on those by definition.

Example. Take again the query Q shown in Figure 1. Let us name the existential node by y; let us name the selected node labeled 0 by z_1; and the selected node labeled 8 by z_2. The distinguished node already has the name x. Now let us apply Q to the simple example graph shown in Figure 2. The following table lists all matchings of Q in G:

	z_1	y	z_2	x
h_1	0	1	8	4
h_2	0	1	8	8
h_3	0	2	8	4
h_4	0	2	8	5
h_5	0	2	8	8

As required by the definition, all matchings match z_1 to 0 and z_2 to 8. Although there are five matchings, we only look at their value on x to distinguish them, as y is existential. So, h_1 and h_3 are identified, as are h_2 and h_4. In conclusion, the frequency of Q in G is three, as x can be matched to the three different nodes 4, 5, and 8. □

Mining tree queries. We are now in a position to define the mining task: *given a graph G and a threshold k, find all tree queries that have frequency at least k in G; these queries are called* frequent.

In theory, however, there are infinitely many frequent queries, and even if we set an upper bound on the size of the queries, there may be exponentially many. As an extreme example, if G is the complete graph on the set of nodes $\{1, \ldots, n\}$, and $k \leq n$, then *any* query with constants in $\{1, \ldots, n\}$, and at least one distinguished variable, is frequent.

Hence, in practice, we want an algorithm that runs incrementally, and that can be stopped any time it has run long enough or has produced enough results.

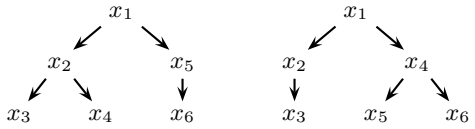


Figure 3: Two orderings of the same tree. The left one is canonical.

3. OVERALL APPROACH

An overall outline of our algorithm is the following:

Outer loop: Generate, incrementally, all possible trees of increasing sizes. Avoid trees that are isomorphic to previously generated ones.

Inner loop: For each new generated tree, generate all queries based on that tree, and test their frequency.

The inner loop is organized in a levelwise manner [17], and will be described in the next Section. Different queries can be equivalent, however, and we must take precautions to avoid generating and testing queries that are equivalent to an earlier seen query. This will be discussed in Section 5.

As for the outer loop, it is already well known how to efficiently generate all trees uniquely up to isomorphism, in increasing number of nodes [15, 16, 4, 14]. These procedures typically generate trees that are *canonically ordered* in the following sense. Given a tree T, we can order the children of every node in some way, and call this an *ordering* of T. For each such possible ordering of T, we can write down the *level sequence* of the resulting tree: if the tree has n nodes then this is a sequence of n numbers, where the ith number is the depth of the ith node in preorder. Here, the depth of the root is 0, the depth of its children is 1, and so on. The canonical ordering of T is then the one that yields the lexicographically maximal level sequence among all possible orderings of T.

For example, Figure 3 shows two orderings of the same tree; the left one is the canonical one.

4. LEVELWISE SEARCH FOR FREQUENT QUERIES

Let G be the graph being mined, and let U be its set of nodes. In this section, we fix a tree T, and we want to find all queries based on T whose frequency in G is at least k.

This tasks lends itself naturally to a levelwise approach [17]. A natural choice for the specialization relation is suggested by an alternative notation for the queries under consideration. Concretely, since the underlying tree is fixed, any query Q is characterized by three parameters:

1. The set Π_Q of existential nodes;
2. The set Σ_Q of selected nodes;
3. The labeling $\lambda_Q : \Sigma_Q \to U$ of the selected nodes by constants.

We now say that $Q_1 = (\Pi_1, \Sigma_1, \lambda_1)$ *specializes* $Q_2 = (\Pi_2, \Sigma_2, \lambda_2)$ if $\Pi_1 \supseteq \Pi_2$; $\Sigma_1 \supseteq \Sigma_2$; and λ_1 agrees with λ_2 on Σ_2. We also say that Q_2 *generalizes* Q_1.

4.1 Candidate generation

Clearly, if Q_1 specializes Q_2, then the frequency of Q_1 is at most that of Q_2, so we can use this relation to guide a levelwise search for the frequent queries. Starting with the most general query $T = (\emptyset, \emptyset, \emptyset)$, we progressively consider more specific queries. The search has the typical property that, in each new iteration, new *candidate* queries are generated: queries whose frequency has not yet been determined, but all whose generalizations are already known to be frequent. Then the frequency of all newly discovered candidate queries is determined, and the process repeats.

There are a great many queries to handle, and in particular, there will be many queries that differ only in λ. Hence, to generate candidate queries in an efficient manner, we propose the use of *candidacy tables* and *frequency tables*. Let Π and Σ be disjoint sets of nodes, as above. Then we define:

$$CanTab_{\Pi,\Sigma} = \{\lambda \mid (\Pi, \Sigma, \lambda) \text{ is a candidate query}\}$$
$$FreqTab_{\Pi,\Sigma} = \{\lambda \mid (\Pi, \Sigma, \lambda) \text{ is a frequent query}\}$$

In practice, the frequency table would also have an additional column to hold the actual frequencies, but for simplicity of presentation we will ignore that column.

Note that when $\Sigma = \emptyset$ these tables are zero-column tables, which formally still make sense and can be interpreted as boolean values; for example, if $FreqTab_{\Pi,\emptyset}$ contains the empty tuple, then the query $(\Pi, \emptyset, \emptyset)$ is frequent; if the table is empty, the query is not frequent.

To populate the tables efficiently, we use the notion of a *parent*. We say that (Π', Σ') is a parent of (Π, Σ) if either (i) $\Pi = \Pi'$ and Σ has precisely one node more than Σ'; or (ii) $\Sigma = \Sigma'$ and Π has precisely one node more than Π'. We then have:

Join Lemma. *A labeling λ is in $CanTab_{\Pi,\Sigma}$ if and only if the following conditions are satisfied for every parent (Π', Σ') of (Π, Σ):*

(i) If $\Pi = \Pi'$, then $\lambda|_{\Sigma'} \in FreqTab_{\Pi',\Sigma'}$;

(ii) If $\Sigma = \Sigma'$, then $\lambda \in FreqTab_{\Pi',\Sigma'}$.

The Join Lemma has its name because, viewing the tables as relational database tables, it can be phrased as follows:

> *Each candidacy table can be computed by taking the natural join of its parent frequency tables.*

The only exception is when $\Pi = \emptyset$ and $\Sigma = \{z\}$ is a singleton; this is the initial iteration of the search process, when there are no constants in the parent tables to start from. In that case, we define $CanTab_{\emptyset,\{z\}}$ as the table with a single column z, holding all nodes of the graph G being mined.

4.2 Frequency counting using SQL

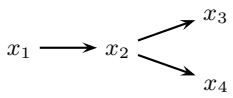
The search process starts by determining the frequency of the underlying tree $T = (\emptyset, \emptyset, \emptyset)$; indeed, formally this amounts to computing $FreqTab_{\emptyset,\emptyset}$. Similarly, for each query $Q = (\Pi, \emptyset, \emptyset)$ with $\Pi \neq \emptyset$, all we can do is determine its frequency, except that here, we do this only on condition that its parent queries are frequent.

We have seen above that, if the frequency tables are viewed as relational database tables, we can compute each candidacy table by a single database query, using the Join Lemma. Now suppose the graph G that is being mined is stored in the relational database system as well, in the form of a table `G(from,to)`. Then also each frequency table can be computed by a single SQL query.

Indeed, in the cases where $\Sigma = \emptyset$ this simply amounts to formulating the query in SQL, and determining its count (eliminating duplicates). But also when $\Sigma \neq \emptyset$, we can compute $FreqTab_{\Pi,\Sigma}$ by a single SQL query. Note that we thus compute the frequency of a large number of patterns in parallel! We proceed as follows. First, we formulate the query $(\Pi, \emptyset, \emptyset)$ in SQL; call the resulting expression E. We then take the natural join of E and $CanTab_{\Pi,\Sigma}$, group by Σ, and count each group. The join with the candidacy table ensures that only candidate queries are counted.

It goes without saying that, whenever the frequency table of a query is found to be empty, the search for more specialized queries is pruned at that point.

Example. Take again the example pattern (query) of Figure 1. This query is based on the following tree T:



Comparing Figure 1, we see that $\Pi = \{x_2\}$, $\Sigma = \{x_1, x_3\}$, and $\lambda = (x_1 \colon 0, x_3 \colon 8)$. The join expression that computes the candidacy table is:

$$FreqTab_{\{x_2\},\{x_1\}} \bowtie FreqTab_{\{x_2\},\{x_3\}} \bowtie FreqTab_{\emptyset,\{x_1,x_3\}}.$$

The SQL expression E for $(\{x_2\}, \emptyset, \emptyset)$ is:

```
select distinct G1.from as x1, G2.to as x3,
                G3.to as x4
from G G1, G G2, G G3
where G1.to=G2.from and G3.from=G2.from
```

The SQL expression that computes the frequency table then is:

```
select E.x1, E.x3, count(E.x4)
from (E) E, CanTab_{x2},{x1,x3} CT
where E.x1=CT.x1 and E.x3=CT.x3
group by E.x1, E.x3
having count(E.x4) >= k
```

4.3 The algorithm

Putting everything together so far, the algorithm is given in Figure 4. In outline it is a double Apriori algorithm [21], where the sets Π form one dimension of itemsets, and the sets Σ another.

5. EQUIVALENT QUERIES

In this section, we make a number of modifications to the algorithm described so far, so as to avoid duplicate work on equivalent queries.

Specifically, suppose we are given two queries with the same number of distinguished variables. There is indeed a natural and purely semantical notion of *equivalence* for such queries, which we define next:

- We begin by defining a *result* of a query Q on a graph G as any matching of Q in G, restricted to the distinguished variables.
- We now say that Q_1 and Q_2 are equivalent if for all graphs G, their result sets on G are the same, up to a renaming of the distinguished variables of Q_1 to those of Q_2.

For each unordered, rooted tree T do:
  $X :=$ set of nodes of T
  $p := 0$; $\mathcal{P}_0 := \{\emptyset\}$
  Repeat:
    For each $\Pi \in \mathcal{P}_p$ do:
      Compute $FreqTab_{\Pi,\emptyset}$
      If $FreqTab_{\Pi,\emptyset} \neq \emptyset$ then:
        $s := 1$; $\mathcal{S}_1 := \{\{z\} \mid z \in X - \Pi\}$
        Repeat:
          For each $\Sigma \in \mathcal{S}_s$ do:
            If $p = 0$ and $s = 1$ then:
              $CanTab_{\Pi,\Sigma} :=$ set of nodes of G
            Else:
              $CanTab_{\Pi,\Sigma} := \bowtie \{CanTab_{\Pi',\Sigma'} \mid (\Pi', \Sigma')$ parent of $(\Pi, \Sigma)\}$
            Compute $FreqTab_{\Pi,\Sigma}$
            If $FreqTab_{\Pi,\Sigma} = \emptyset$ then remove Σ from $\mathcal{S}_s$
          Done
          $\mathcal{S}_{s+1} := \{\Sigma \subseteq X - \Pi \mid \#\Sigma = s + 1$ and each s-subset of Σ is in $\mathcal{S}_s\}$
          $s := s + 1$
        Until $\mathcal{S}_s = \emptyset$
      Else remove Π from $\mathcal{P}_p$
    Done
    $\mathcal{P}_{p+1} := \{\Pi \subseteq X \mid \#\Pi = p + 1$ and each p-subset of Π is in $\mathcal{P}_p\}$
    $p := p + 1$
  Until $\mathcal{P}_p = \emptyset$
Done

Figure 4: Levelwise search for frequent queries.

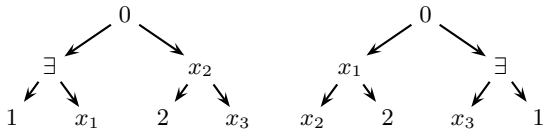


Figure 5: Two equivalent queries.

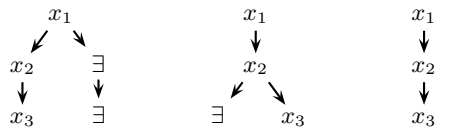


Figure 6: Three equivalent queries.

For example, the two queries shown in Figure 5 are equivalent, as are the three queries shown in Figure 6.

Of course, we want to avoid that our algorithm considers some query Q_2 if it is equivalent to an earlier considered query Q_1. Since our algorithm generates trees in increasing sizes, there are two cases to consider:

Case A: Q_1 has fewer nodes than Q_2.

Case B: Q_1 and Q_2 have the same number of nodes.

We can analyze the situation using a well-known result from the theory of conjunctive queries [12, 13]. In order to state that result, we need the notion of *containment mapping*. A containment mapping from Q_1 to Q_2 is a homomorphism from Q_1 to Q_2 that maps the distinguished variables of Q_1 one-to-one to the distinguished variables of Q_2, and that maps selected nodes of Q_1 to selected nodes of Q_2, preserving labels. From the theory of conjunctive database queries [12, 13] we recall the following:

Equivalence Theorem. *Two queries are equivalent if and only if there are containment mappings between them in both directions.*

Armed with this theorem, we now analyze the above two cases.

5.1 Case A: Redundancy checking

Using the equivalence theorem, it can be seen that this case can only happen if Q_2 contains *redundant subtrees*: subtrees such that removing them yields an equivalent query. (For example, the first two queries in Figure 6 indeed contain a redundancy.) In other words, if we can avoid queries with redundancies, then case A will not occur.

The following lemma provides us with an efficient check for redundancies. The proof is given in the Appendix.

Redundancy Lemma. *Let Q be a tree query without selected nodes. Then Q has a redundancy if and only if contains a subtree C in the form of a linear chain of existential nodes (possibly just a single node), such that the parent of C has another subtree that is at least as deep as C.*

This lemma ignores selected nodes, but this is justified. Indeed, from the equivalence theorem and the redundancy lemma it follows that, if all selected nodes in a query Q are labeled with distinct constants, then Q has a redundancy if and only if Q' has a redundancy, where Q' is obtained from Q by turning all selected nodes back into distinguished variables. Since, as explained in Section 4.1, we handle all queries that differ only in the labeling of selected nodes together in a single table, it is indeed harmless to treat selected nodes as distinguished variables.

As we have seen in Section 4, our algorithm introduces existential nodes levelwise, one by one. This makes the redundancy test provided by the redundancy lemma particularly easy to perform. Indeed, if Q is a query of which we already know it has no redundancies, and we make one additional node i existential, then it suffices to test whether i thus becomes part of a subtree C as in the Redundancy Lemma. If so, we will prune the entire search at $\Pi_Q \cup \{i\}$.

5.2 Case B: Canonical forms

We may now assume that Q_1 and Q_2 do not contain redundancies, for if they would, they would have been dismissed already. Again using the isomorphism theorem, it can then be seen that case B can only happen if Q_1 and Q_2 are actually *isomorphic*, i.e., there is a containment mapping from Q_1 to Q_2 that is one-to-one, and whose inverse is a containment mapping from Q_2 to Q_1. (For example, the two queries in Figure 5 are indeed isomorphic.) In particular, Q_1 and Q_2 have the same underlying tree. So, in our algorithm, we need an efficient way to avoid isomorphic queries based on the same tree T.

Fortunately, there is a standard way to do this, by working with *canonical forms* of queries. Consider a pair (Π, Σ) over T, as in Section 4. We can view this pair as a labeling of T: all nodes in Π get the same generic label '$\exists$'; all nodes in Σ get 'c'; and all remaining nodes get 'x'. We then say that two pairs (Π_1, Σ_1) and (Π_2, Σ_2) are *isomorphic* if there is a tree isomorphism between the corresponding labeled versions of T that respects the labels.

It is sufficient to work on the level of pairs (Π, Σ) to capture all isomorphic queries. Indeed, if $(\Pi_1, \Sigma_1, \lambda_1)$ and $(\Pi_2, \Sigma_2, \lambda_2)$ are isomorphic queries, then certainly the pairs (Π_1, Σ_1) and (Π_2, Σ_2) are equivalent as well. And conversely, if (Π_1, Σ_1) and (Π_2, Σ_2) are equivalent, then for each query $(\Pi_1, \Sigma_1, \lambda_1)$ there is an isomorphic query $(\Pi_2, \Sigma_2, \lambda_2)$.

In order to represent each pair (Π, Σ) uniquely up to isomorphism, we can rather straightforwardly refine the canonical ordering of the underlying unlabeled tree, which we already have (Section 3), to take into account the node labels. Furthermore, the classical linear-time algorithm to canonize a tree [22] generalizes straightforwardly to labeled trees. A nice review of these generalizations has been given by Chi, Yang and Muntz [14].

We will omit the details of the canonical form; in fact, there are several ways to realize it. All that is important is that we can check in linear time whether a pair is canonical; that a pair can be canonized in linear time; and that two pairs are isomorphic if and only if their canonical forms are identical.

Armed by the canonical form, we are now in a position to describe how the algorithm of Section 4 must be modified to avoid equivalent queries. First of all, we only work with pairs (Π, Σ) in canonical form; the others are dismissed. The problem then arises, however, that a parent pair (Π', Σ'), where we omit a variable from either Π or Σ as described above, might be non-canonical. In that case the frequency table for (Π', Σ') will not exist. We can solve this by canonizing (Π', Σ') to its canonical version (Π'', Σ''), and remembering the renaming of variables this entails. The table $FreqTab_{\Pi'', \Sigma''}$ can then serve in place of $FreqTab_{\Pi', \Sigma'}$, after we have applied the inverse renaming to its column headings.

6. RESULT MANAGEMENT AND ASSOCIATION RULES

When our algorithm is terminated, its final output consists of a set of frequency tables for each tree T that was investigated. These tables together form a database that provides an ideal data platform for browsing the mining results. We envisage a browsing tool in which the user draws a tree shape, marks some nodes as existential, and marks some others as selected, annotating some of these by constants, but possibly also leaving some of the selected nodes open for instantiation. By consulting the appropriate frequency table in our results database, we can immediately report to the user those annotations of the selected nodes that make the pattern frequent.

Our platform also supports a simple form of *association rules*. While a thorough treatment of association rules over tree queries is beyond the scope of the present paper, we can easily support a simple but useful kind of such rules. Let Q_1 be a frequent tree query, and let Q_2 be another frequent tree query obtained from Q_1 by making some additional nodes selected. Clearly, the frequency f_2 of Q_2 will be at most the frequency f_1 of Q_1, and we can define the *confidence* of the rule $Q_1 \Rightarrow Q_2$ as the fraction f_2/f_1. With all the frequency tables in place, it is a simple matter to compute confidences. Thus, given a Q_1 and a threshold c, an Apriori-like algorithm can be used to compute all rules $Q_1 \Rightarrow Q_2$ of the above kind with confidence at least c. The "itemsets" are here, of course, the possible sets of extra selected nodes in Q_2 as compared to Q_1.

We will give a real-life example of an association rule in the next Section.

7. EXPERIMENTAL RESULTS

In this section, we report on some preliminary experiments performed using our prototype implementation applied to both real-life and synthetic datasets. The results show that our approach is indeed workable. In the near future we plan to pay more attention to performance tuning, and to work with domain experts to see if new scientific facts can be discovered.

The experiments were performed on a Pentium IV (2.8GHz) architecture with 1GB of internal memory, running under Linux 2.6. The program was written in C++ with embedded SQL, with DB2 UDB v8.2 as the relational database system.

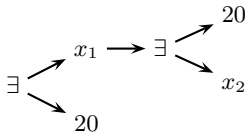
7.1 Real-life datasets

We have worked with a food web, a protein interactions graph, and a citation graph. For each dataset, the table below lists the number of nodes, number of edges, frequency threshold k, and maximum size of trees considered in the run. As we set rather generous limits on the maximum size of trees, or on the minimum frequency threshold, each run took several hours.

	#nodes	#edges	k	size
food web	154	370	25	6
proteins	2114	4480	10	5
citations	2500	350000	5	4

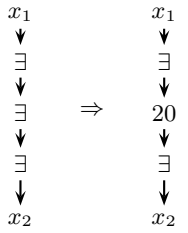
The **food web** [23] comprises 154 species that are all directly or indirectly dependent on the Scotch Broom (a

kind of shrub). One of the patterns that was mined with frequency 176 is the following:



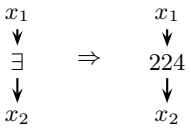
This is really a rather arbitrary example, just to give an idea of the kind of complex patterns that can be mined. Note also that, thanks to the constant 20 appearing twice, this is really a non-tree shaped pattern: we could equally well draw both arrows to a single node labeled 20.

While we were thus browsing through the results, we quickly noticed that the constant 20 actually occurs quite predominantly, in many different frequent patterns. This constant denotes the species *Orthotylus adenocarpi*, an omnivorous plant bug. To confirm our hypothesis that this species plays a central role in the food web, we asked for all association rules with the following left-hand side:

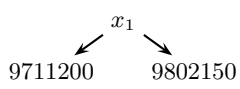


Indeed, the rule shown above turned up with 89% confidence! For 89% of all pairs of species that are linked by a path of length four, *Orthotylus adenocarpi* is involved in between.

The **protein interaction graph** [24] comprises molecular interactions (symmetric) among 1870 proteins occurring in the yeast *Saccharomyces cerevisiae*. In such interaction networks, typically a small number of highly connected nodes occur. Indeed, we discovered the following association rule with 10% confidence, indicating that protein #224 is highly connected:



The **citation graph** comes from the KDD cup 2003, and contains around 2500 papers about high-energy physics taken from arXiv.org, with around 350 000 cross-references. One of the discovered patterns is the following, with frequency 1655, showing two papers that are frequently cited together (by 6% of all papers).



7.2 Performance

While our current prototype implementation has not yet been tuned for performance, we still conducted some preliminary performance measurements, with encouraging results. We have used two types of synthetic datasets.

Random Web graphs. Naturally occurring graphs (as found in biology, sociology, or the WWW) have a number of typical characteristics, such as sparseness and a skewed degree distribution [25]. Various random graph models have been proposed in this respect, of which we have used the "copy model" for Web graphs [26, 27]. We use degree 5 and probability $\alpha = 10\%$ to link to a random node (thus 90% to copy a link).

On these graphs, we have measured the total running time as a function of the size (number of edges) of the graph, where we mine up to tree size 5, with varying minimum frequency thresholds of 4, 10, and 25. The results, depicted in Figure 7, show that the performance of these runs is quite adequate.

Uniform random graphs. We have also experimented with the well-known Erdös–Rényi random graphs, where one specifies a number n of nodes and gives each of the possible n^2 edges a uniform probability (we used 10%) of actually belonging to the graph. In contrast to random Web graphs, these graphs are quite dense and uniform, and they serve well as a worst-case scenario to measure the performance as a function of the number of discovered patterns, which will be huge.

We have run on graphs with 47, 264, and 997 edges, with minimum frequency thresholds of 10 and 25. The results, depicted in Figure 8, show, first, that huge numbers of patterns are mined within a reasonable time, and second, that the overhead per discovered pattern is constant (all six lines have the same slope).

8. REFERENCES

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

[2] A. Inokuchi, T. Washio, and H. Motoda. An Apriori-based algorithm for mining frequent substructures from graph data. In D.A. Zighed, H.J. Komorowski, and J.M. Zytkow, editors, *PKDD*, volume 1910 of *Lecture Notes in Computer Science*, pages 13–23. Springer, 2000.

[3] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In Cercone et al. [28], pages 313–320.

[4] M.J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80. ACM Press, 2002.

[5] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)* [29], pages 721–724.

[6] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM 2003)*, pages 549–552. IEEE Computer Society Press, 2003.

[7] D.J. Cook and L.B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[8] N. Vanetik, E. Gudes, and S.E. Shimony. Computing frequent graph patterns from semistructured data. In

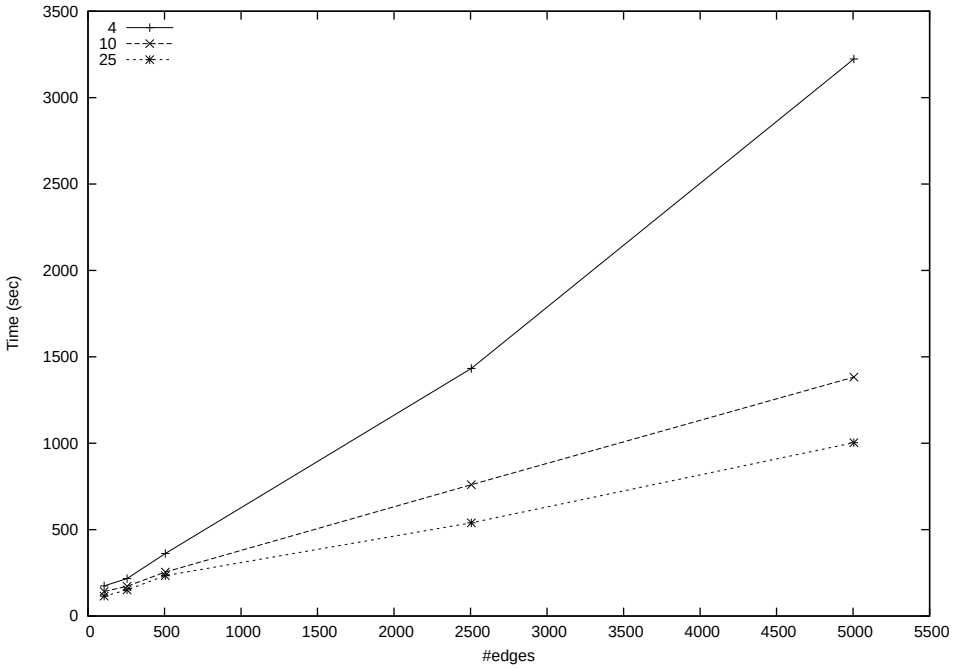


Figure 7: Performance on Web graphs.

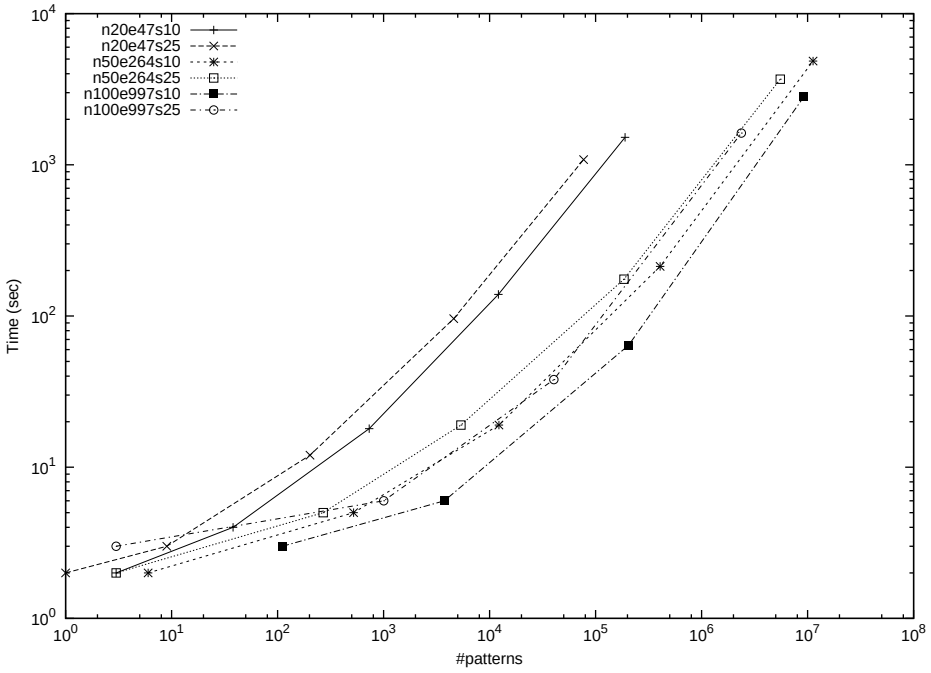


Figure 8: Performance in terms of number of discovered patterns.

Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002) [29], pages 458–465.

[9] S. Ghazizadeh and S. Chawathe. SEuS: Structure extraction using summaries. In S. Lange, K. Satoh, and C.H. Smith, editors, *Discovery Science*, volume 2534 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2002.

[10] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In M.W. Berry, U.Dayal, C.Kamath, and D.B. Skillicorn, editors, *Proceedings of the Fourth SIAM International Conference on Data Mining*. SIAM, 2004.

[11] G. Jeh and J. Widom. Mining the space of graph properties. In W. Kim, R. Kohavi, J. Gehrke, and W. DuMouchel, editors, *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 187–196. ACM Press, 2004.

[12] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings 9th ACM Symposium on the Theory of Computing*, pages 77–90. ACM Press, 1977.

[13] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[14] Y. Chi, Y. Yang, and R.R. Muntz. Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowledge and Information Systems*, 2004. Published online, 31 August.

[15] H.I. Scions. Placing trees in lexicographic order. In D. Michie, editor, *Machine Intelligence 3*, pages 43–62. Edinburgh University Press, 1968.

[16] G. Li and F. Ruskey. The advantages of forward thinking in generating rooted and free trees. In *Proceedings 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 939–940, 1999.

[17] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

[18] S. Tsur, J.D. Ullman, et al. Query flocks: A generalization of association-rule mining. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, volume 27:2 of *SIGMOD Record*, pages 1–12. ACM Press, 1998.

[19] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. *Data Mining and Knowledge Discovery*, 4(2–3):89–125, 2000.

[20] S. Chakravarthy, R. Beera, and R. Balachandran. DB-Subdue: Database approach to graph mining. In H. Dai, R. Srikant, and C. Zhang, editors, *Advances in Knowledge Discovery and Data Mining, Proceedings 8th PAKDD Conference*, volume 3056 of *Lecture Notes in Computer Science*, pages 341–350. Springer, 2004.

[21] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. MIT Press, 1996.

[22] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[23] J. Memmott, N.D. Martinez, and J.E. Cohen. Predators, parasites and pathogens: species richness, trophic generality, and body sizes in a natural food web. *Journal of Animal Ecology*, 69:1–15, 2000.

[24] H. Jeong, S.P. Mason, et al. Lethality and centrality in protein networks. *Nature*, 411(3 May 2001).

[25] M. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[26] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Random graph models for the web graph. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 57–65. IEEE Computer Society, 2000.

[27] K. Bharat, B.-W. Chang, M. Henzinger, and M. Ruhl. Who links to whom: Mining linkage between Web sites. In Cercone et al. [28], pages 51–58.

[28] N. Cercone, T.Y. Lin, and X. Wu, editors. *Proceedings of the 2001 IEEE International Conference on Data Mining, 29 November - 2 December 2001, San Jose, California, USA*. IEEE Computer Society Press, 2001.

[29] *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002), 9-12 December 2002, Maebashi City, Japan*. IEEE Computer Society Press, 2002.

APPENDIX

Proof of the Redundancy Lemma. Let us refer to a subtree C as described in the lemma as an "eliminable path". An eliminable path is clearly redundant, so we only need to prove the 'only-if' direction. Let T be a redundant subtree of Q that is maximal, in the sense that it is not the subtree of another redundant subtree. Then there must be a containment mapping h of Q to $Q - T$. All distinguished variables of Q must already be in $Q - T$, since containment mappings are one-to-one on the distinguished variables. Hence, T consists entirely of existential nodes. Also, note that h must fix the root of Q, since the height of Q is at least that of $Q - T$.

Any iteration h^n of h is a containment mapping of Q to $Q - T$ as well. Moreover, each h^n induces a permutation on the set V of distinguished variables. Since V is finite, there are only a finite number of possible permutations of V. A standard argument then shows that there is an iteration $h' = h^m$ that is the identity on V.

There are now two possible cases.

First, T itself may be a linear chain. Then the parent p of T must either be distinguished, or must have at least two children. Indeed, if p would be existential with T as only subtree, then the subtree T' rooted at p would be redundant as well, contradicting the maximality of T. If $h'(p) = p$, then T is mapped by h' to another subtree of p; since h' is a homomorphism, that subtree must be at least as deep as T. Hence, T is an eliminable path and we are done. If $h'(p) \neq p$, then the subtree rooted at p consists entirely of existential nodes, since h' is the identity on distinguished nodes. This brings us in the second case.

Second, T is not a linear chain. An easy induction on the height shows that any non-linear tree consisting entirely of existential nodes must contain an eliminable path. Hence, T, and thus also Q, contains an eliminable path as desired. □