

DOCTORAATSPROEFSCHRIFT

2007 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Mining Tree-Query Associations in Graphs

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen: Informatica, te verdedigen door:

Eveline HOEKX

Promotor: prof. dr. Jan Van den Bussche

37

selt



Universiteit Maastricht

universiteit
hasselt

BIBLIOTHEEK UNIVERSITEIT HASSELT



03 04 0087325 0

071185



27 NOV 2007

681.37
HOEK
2007

uhasselt

DOCTORAATSPROEFSCHRIFT

2007 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

071185

Mining Tree-Query Associations in Graphs

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen: Informatica, te verdedigen door:

Eveline HOEKX



Promotor: prof. dr. Jan Van den Bussche

27 NOV 2007



Universiteit Maastricht

universiteit
▶▶ hasselt

Acknowledgements

A PhD thesis is not possible without the contributions of many people.

First and foremost, I am grateful to my advisor Jan Van den Bussche for his guidance, patience, and never ending enthusiasm the past four years. As a result of all the effort he put in proofreading my texts, I am now able to write a decent scientific text. I enjoyed his enthusiasm and knowledge about birds, which resulted in naming our tree-query browser after a bird that crawls trees, *Certhia*.

I thank Bart Goethals for his contribution in the initial conception of the ideas presented in this thesis, and I also thank Jan Hidders and Dries Van Dyck for their help with our results on graph isomorphism.

Special thanks to my office-mate Dieter Van de Craen for his interest in my research, encouragement and support, and for all the pleasant moments we shared in D6.

Also many thanks go out to the other members of our research group, the department and the administrative staff for creating a stimulating environment.

Although I did not manage to beat him before the end of my PhD, I am grateful to Kurt for all the pleasant squash games, which were a well-needed distraction.

I am much in debt with my parents, Jeroen and Sarah, other family members, and friends for their support and encouragement during my career as a student.

Finally, I am very grateful to Alpha, for his patience, understanding, listening to my complaints and never ending support.

Diepenbeek, November 2007

Contents

Acknowledgements	i
1 Introduction	1
1.1 Related Work	5
1.2 Outline	9
2 Tree Queries and Tree-Query Associations	11
2.1 Tree Pattern	11
2.2 Tree Query	14
2.2.1 Containment of Tree Queries	14
2.3 Tree-Query Association	17
2.4 Mining Problems	18
2.4.1 Mining Tree Queries	18
2.4.2 Association Rule Mining	18
3 Mining Tree Queries	19
3.1 Problem Reduction	19
3.2 Overall Approach	20
3.3 Outer Loop	21
3.4 Inner Loop	21
3.4.1 Candidate Generation	23
3.4.2 Frequency Counting using SQL	24
3.4.3 The Algorithm	25
3.4.4 Example Run	25
3.5 Equivalence among Tree Patterns	30
3.5.1 Equivalency	31
3.5.2 Case A: Redundancy Checking	34
3.5.3 Case B: Canonical Forms	37
3.5.4 The Algorithm	42
3.5.5 Example Run	42
3.6 Result Management: Pattern Database	46

4 Mining Tree-Query Associations	47
4.1 Problem Reduction	47
4.2 Overall Approach	51
4.3 Generation of Containment Mappings	52
4.4 Generation of Parameter Assignments	55
4.5 Example Run	56
4.6 Equivalent Association Rules	59
4.6.1 Testing for Equivalence	61
4.6.2 Hardness Argument	63
4.6.3 Polynomial Case	66
4.6.4 The Algorithm	66
5 Experimental Results	67
5.1 Certhia: Pattern and Association Browsing	67
5.2 Smaller Experiments	68
5.2.1 Real-Life Datasets	71
5.2.2 Performance	73
5.3 Ecology Experiment	75
5.3.1 Natal-Dispersal Dataset	76
5.3.2 Graph Construction	77
5.3.3 Tree-Query Browsing	80
5.3.4 Conclusion	88
6 Conclusions and Future Work	89
Bibliography	95
Notations	99
Samenvatting (Dutch Summary)	101

1

Introduction

Data mining is a new research area that attracted a lot of attention the past decade. A well-known textbook [21] on data mining motivates this new research area as follows:

Progress in digital data acquisition and storage technology has resulted in the growth of huge databases. This has occurred in all areas of human endeavour, from the mundane (such as supermarket transaction data, credit card usage records, telephone call details, and government statistics) to the more exotic (such as images of astronomical bodies, molecular databases, and medical records). Little wonder, then, that interest has grown in the possibility of tapping these data, of extracting from them information that might be of value to the owner of the database.

Data mining is concerned with the task of automatically extracting information from huge datasets that might be interesting for the owners of the datasets.

In the beginning, data mining was usually only applied to rather simple datasets such as transaction databases. However, recently interest grew to apply data mining to more complex datasets such as data streams, graphs, trees and xml-files.

In this thesis we focus on the case where the dataset is a graph. Graphs become more and more important in modeling complicated structures, such as circuits, images, chemical compounds, protein structures, biological networks, social networks, the Web, workflows, and XML documents. *Graph mining* has become an active and important theme in data mining since there is an increasing demand for analyzing large amounts of graph-structured data.

Among the various kinds of graph patterns, *frequent substructures* are the very basic patterns that can be discovered in a (collection of) data graph(s). They are useful for characterizing graph sets, classifying and clustering graphs, and facilitating similarity search in graph databases. Although, graph mining may include mining

frequent subgraph patterns, graph classification, clustering, and other analysis tasks, we focus in this thesis on mining frequent tree-shaped patterns and rules over these patterns.

We refer the interested reader for more information on graph mining in general, to a chapter of a book that was recently published on this topic [20].

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. In the present work we introduce a novel class of patterns, called tree queries, and we present algorithms for mining these tree queries and tree-query associations in a large unlabeled directed data graph.

Most parts of this text are based on two earlier conference papers [16, 22].

Tree queries are powerful tree-shaped patterns, inspired by conjunctive database queries [17]. In comparison to the kinds of patterns used in most other graph mining approaches, tree queries have some extra 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 “parameterized” nodes, labeled by constants (node identifiers), which must map to fixed designated nodes of the data graph.
- An “occurrence” of the pattern in a data graph 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.

Past work in graph mining has dealt with node labels, but only with non-unique ones: as we will show in Section 1.1 such labels are easily simulated by constants, but the converse is not obvious. It is also possible to simulate edge labels using constants.

A simple example of a tree query is shown in Figure 1.1(a); when applied to a food web: a data graph of organisms, where there is an edge $x \rightarrow y$ if y feeds on x , 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 parameters, and one distinguished node x . Figure 1.1(b) shows another example of a tree query; when applied to a food web, it describes all organisms x that have a path of length four beneath them that ends in organism #8.

Effectively, tree queries are what is known in database research as *conjunctive queries* [8, 44, 1]; these are the queries we could pose to the data 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 equality comparisons as where-conditions. For example, the pattern of Figure 1.1(a) amounts to the following SQL query on a table $G(\text{from}, \text{to})$:

```
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
```

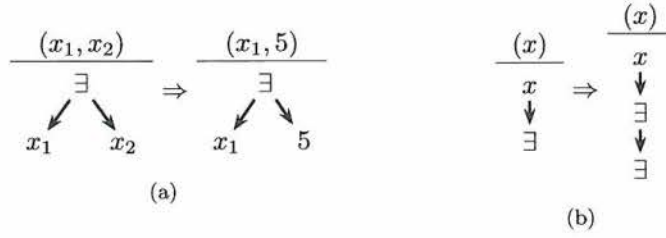



Figure 1.2: Simple examples of association rules over tree queries.

often in the single data graph. We will refer to this group of algorithms as the single graph category. There is also a second category of graph mining algorithms, called the transactional category, which is explained in Section 1.1.

2. We restrict to patterns that are trees, such as the examples in Figure 1.1. Tree patterns have formed an important special case in the transactional category (Section 1.1), but have not yet received special attention in the single-graph literature. Note that the data graph that is being mined is not restricted in any way.
3. The tree-query-mining algorithm is incremental in the number of nodes of the pattern. 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. Thanks to the restriction to tree shapes the duplicate-free generation of trees can be done efficiently.
4. For each tree, all conjunctive queries based on that tree are generated in the tree-query-mining algorithm. Here, we work in a levelwise fashion in the sense of Mannila and Toivonen [33].
5. As in classical association rules over itemsets [2], our association rule generation phase comes after the generation of frequent patterns and does not require access to the original dataset.
6. We apply the theory of conjunctive database queries [8, 44, 1] to formally define and to correctly generate association rules over tree queries. The conjunctive-query approach to pattern matching allows for an efficiently checkable notion of frequency, whereas in the subgraph-based approach, determining whether a pattern is frequent is NP-complete (in that approach the frequency of a pattern is the maximal number of disjoint subgraphs isomorphic to the pattern [18, 31]).
7. There is a notion of equivalence among tree queries and association rules over tree queries. We carefully and efficiently avoid the generation of equivalent tree queries and associations, by using and adapting what is known from the theory of conjunctive database queries. Due to the restriction to tree shapes, equivalence and redundancy (which are normally NP-complete) are efficiently checkable.

8. Last but not least, our algorithms naturally suggest 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 [43, 40], but less so in graph mining, a recent exception being an implementation in SQL of the seminal SUBDUE algorithm [7].

Note that if we would define an occurrence of a tree query in the data graph G , as an isomorphism from the tree query in G , instead of a homomorphism from the tree query in G , we could still use the theory of conjunctive queries to check for equivalence, and we could also use SQL to compute the frequency. However, both tasks would not be that straightforward as they are for homomorphisms, since for isomorphisms our tree queries are actually conjunctive queries with inequalities. For that kind of conjunctive queries the notion of equivalence is more complicated, and computing their frequency with SQL is also harder.

1.1 Related Work

Past work in graph mining has dealt with graphs where the nodes and the edges are labeled. In our work, we mine patterns and rules from the most simple kind of directed graphs, where nodes and edges are not labeled². But labeled graphs are easily simulated by unlabeled graphs: 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 $x \rightarrow y$ labeled b , introduce an intermediate node $x.y$ with $x \rightarrow x.y \rightarrow y$, and give node $x.y$ label b . As an illustration consider the labeled graph in Figure 1.3(a) and its unlabeled version in Figure 1.3(b).

Approaches to graph mining, especially mining for frequent patterns or association rules, can be divided in two major categories which are not to be confused.

1. In transactional graph mining, e.g., [11, 23, 24, 25, 30, 45, 46], the dataset consists of many small data graphs which we call transactions, and the task is to discover patterns that occur at least once in a sufficient number of transactions. (Approaches from machine learning or inductive logic programming usually call the small data graphs “examples” instead of transactions.)
2. In single-graph mining the dataset is a single large data graph, and the task is to discover patterns that occur sufficiently often in the dataset.

²Note that unlabeled graphs are in fact a special case of labeled graphs where all edges and all nodes have the same label.

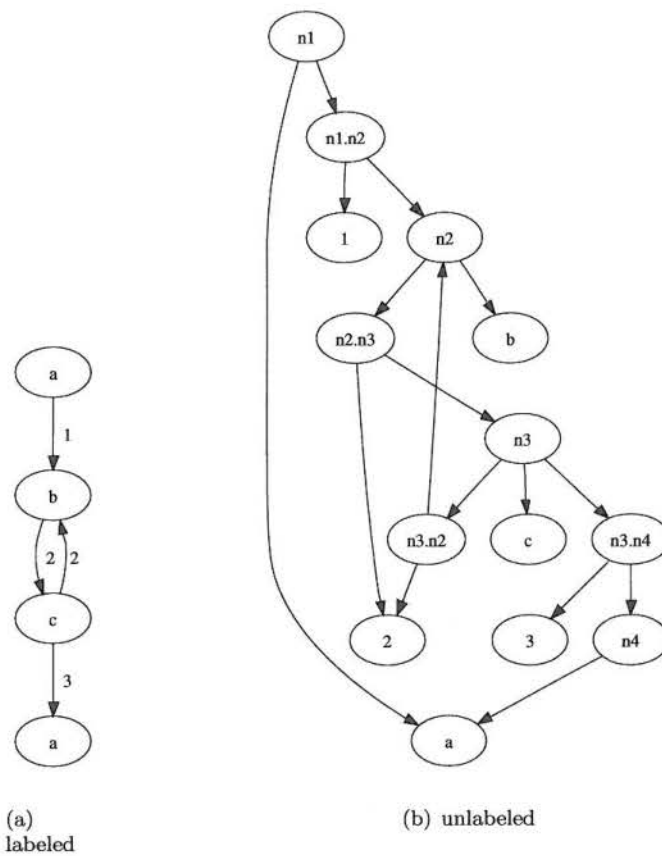


Figure 1.3: The labeled graph in (a) is simulated by the unlabeled graph in (b).

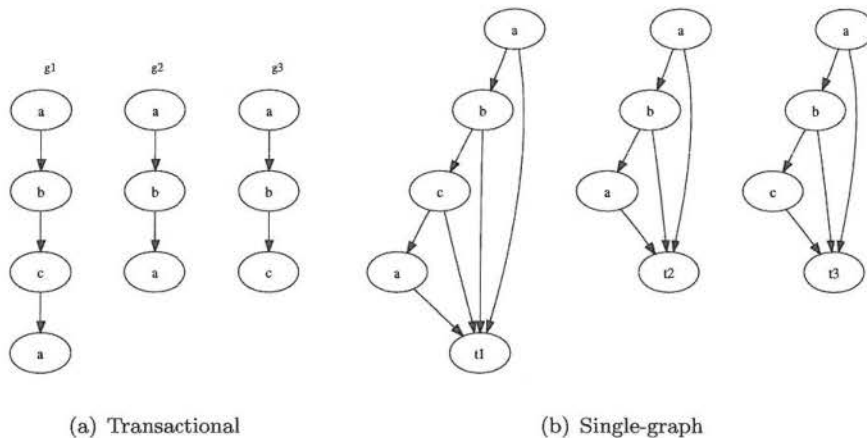


Figure 1.4: The graph transaction database in (a) is simulated by the single-graph database in (b).

Note that single-graph mining is more difficult than transactional mining, in the sense that transactional graph mining can be simulated by single-graph mining, but the converse is not obvious. To simulate transactional mining by single-graph mining, create a single large graph G from a set of many small graphs $\{g_1, g_2, \dots, g_n\}$ by simply taking the union of the smaller graphs, so $G = g_1 \cup g_2 \cup \dots \cup g_n$. To distinguish between the different smaller graphs, add to each graph g_i a node t_i that is labeled with the transaction id of g_i . Furthermore, add edges from each node of g_i to t_i . Now we can apply algorithms from the single-graph category to this larger graph G , but when determining the frequency of a pattern, we must now count the transaction ids instead of simply the number of matchings. As an illustration consider the graph transaction database in Figure 1.4(a), and its single graph simulation in Figure 1.4(b). Note that it is even possible to simulate a labeled graph transaction database with an unlabeled single graph, if we combine this simulation with the previous simulation to go from a labeled graph to an unlabeled graph. As an illustration, the labeled graph transaction database in Figure 1.4(a) is simulated by the unlabeled single graph in Figure 1.5.

Since our approach falls squarely within the single-graph category, we will focus on that category in this Section. Most work in this category has been done on frequent pattern mining, and less attention has been spent on association rules. We briefly review the work in this category next:

- Cook and Holder [10] apply in their SUBDUE system the minimum description length (MDL) principle to discover substructures in a labeled data graph. The MDL principle states that the best pattern, is that pattern that minimizes the description length of the complete data graph. Hence, in SUBDUE a pattern is evaluated on how well it can compress the entire dataset. The input for the SUBDUE system is a labeled data graph; nodes and edges are labeled with non-unique labels. This is in contrast with the unique labels ('constants') in our system. But as we already noted, non-unique node labels and edge-labels

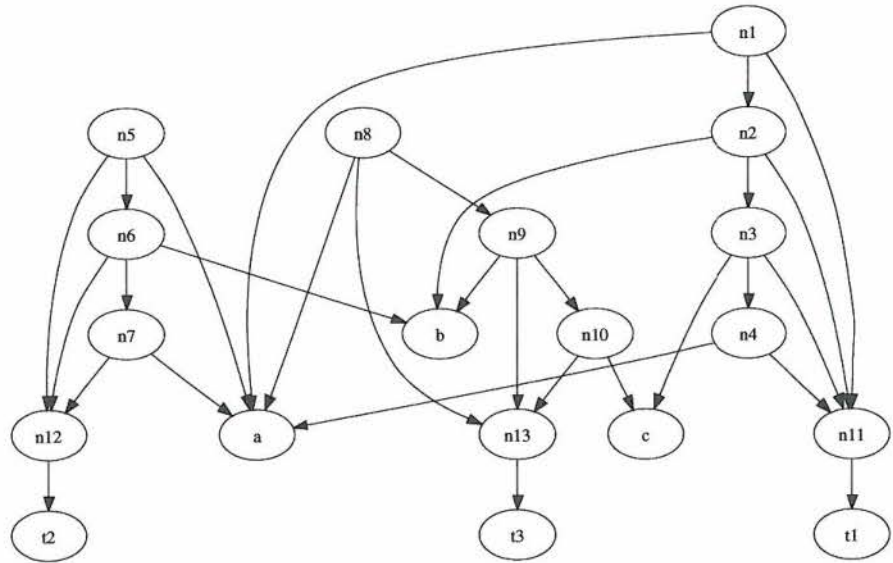


Figure 1.5: The labeled graph transaction database from Figure 1.4(a) simulated by a single unlabeled graph.

can easily be simulated by constants, but the converse is not obvious. The SUBDUE system only mines patterns, no association rules.

- Ghazizadeh and Chawathe [15] mine in their SEuS system for connected subgraphs in a labeled, directed data graph, as in the SUBDUE system. Instead of generating candidate patterns using the input data graph, SEuS uses a summary of the data graph. This summary gives an upper bound for the support of the patterns, and the user can then select those patterns of which he wants to know the exact support. SEuS also only mines for frequent patterns and not for associations.
- Vanetik, Gudes, and Shimony [18] propose an Apriori-like [2] algorithm for mining subgraphs from a labeled data graph. The support of a graph pattern is defined as the maximal number of edge-disjoint instances of the pattern in the data graph. By reducing the support counting problem to the maximal independent set problem on graphs, they show that in worst case, computing the support of a graph pattern is NP-hard. They propose an Apriori-like algorithm to minimize the number of patterns for which the support needs to be computed. The major idea of their approach is using edge-disjoint paths as building blocks instead of items in classical itemset mining. Vanetik, Gudes, and Shimony also only mine for frequent patterns in the data graph.
- Kuramochi and Karypis [31] use the same support measure for graph patterns as Vanetik, Gudes and Shimony [18]. They also note that computing the support of a graph pattern is NP-hard in worst case, since it can be reduced to finding the

maximum independent set (MIS) in a graph. Kuramochi and Karypis quickly compute the support of a graph pattern using approximate MIS-algorithms. The number of candidate patterns is restricted using canonical labeling. As the majority of algorithms, Kuramochi and Karypis only mine for frequent patterns.

- Jeh and Widom [26] consider patterns that are, like our tree queries, inspired by conjunctive database queries, and they also emphasize the tree-shaped case. A severe restriction, however, is that their patterns can be matched by single nodes only, rather than by tuples of nodes. Still their work is interesting in that it presents a rather non-standard approach to graph mining, quite different from the standard incremental, levelwise approach, and in that it incorporates ranking. Jeh and Widom mention association rules as an example of an application of their mining framework.

The related work that was most influential for us is Warmr [11, 12], although it belongs to the transactional category. Based on inductive logic programming, patterns in Warmr also feature existential variables and parameters. While not restricted to tree shapes, the queries in Warmr are restricted in another sense so that only transactional mining can be supported. Association rules in Warmr are defined in a naive manner through pattern extension, rather than being founded upon the theory of conjunctive query containment. The Warmr system is also Prolog-oriented, rather than database-oriented, which we believe is fundamental to mining of single large data graphs, and which allows a more uniform and parallel treatment of parameter instantiations, as we will show in this paper. Finally, Warmr does not seriously attempt to avoid the generation of duplicates. Yet, Warmr remains a pathbreaking work, which did not receive sufficient follow-up in the data mining community at large. We hope our present work represents an improvement in this respect. Many of the improvements we make to Warmr were already envisaged (but without concrete algorithms) in 2002 by Goethals and Van den Bussche [17].

Finally, we note that parameterized conjunctive database queries have been used in data mining quite early, e.g., [43, 42], but then in the setting of “data mining query languages”, where a *single* such query serves to specify a family of patterns to be mined or queried for, rather than the mining for such queries themselves, let alone associations among them.

1.2 Outline

This thesis is further organized as follows:

- In **Chapter 2** we formally define a novel class of patterns, called tree queries. We introduce the notion of containment among tree queries, and define it formally. Furthermore, association rules over tree queries are defined, and we conclude the chapter by defining the mining problems that we solve in this thesis.
- In **Chapter 3** we present an algorithm for mining tree queries in a large data graph. We start by showing that we do not need to tackle the problem in its

full generality. Then, we give an overall approach, basically two loops, of the presented algorithm, and discuss it in more detail. Furthermore, we discuss equivalent tree queries, and show how the algorithm must be tuned to avoid the generation of them. We conclude the chapter with some notes on how the results of this algorithm are stored and why that is useful.

- In **Chapter 4** we present an algorithm for mining tree-query associations. Again, we show that we do need to tackle the problem in its full generality. We give an overview of the presented algorithm and show that the tree-query mining algorithm, discussed in Chapter 3, is an ideal preprocessing step. Furthermore, we discuss the remaining steps of the algorithm in more detail. We conclude the chapter by defining equivalent association rules, and by showing how we must tune the presented algorithm to avoid the generation of them.
- In **Chapter 5** we first introduce an interactive tool, called Certhia, for browsing the mined patterns and generating association rules. Next, we give results of some smaller experiments we performed using a prototype implementation. Furthermore, we explain how are algorithms can be used to find interesting patterns and rules in data from ecology, and give examples of interesting patterns and rules we mined.
- In **Chapter 6** we give conclusions on the presented work and give some points that need to be improved in the future.

2

Tree Queries and Tree-Query Associations

Tree queries are powerful tree-shaped patterns, inspired by conjunctive database queries [17], that have some extra features in comparison to the kinds of patterns used in most other graph mining approaches:

- 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 “parameterized” nodes, labeled by constants, which must map to fixed designated nodes of the data graph.
- An “occurrence” of the pattern in a data graph 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.

In this Chapter we define tree queries formally and introduce the notion of equivalence among tree queries. We also introduce tree-query associations. Tree-query associations can be used to discover quite subtle properties of the data graph. This Chapter is concluded by the definitions of the mining problems we solve in this thesis.

An overview of all notations used in this Chapter and the rest of this thesis is given on page 99.

2.1 Tree Pattern

Graph-theoretic concepts We basically assume a set U of *data constants* from which the nodes of the data graph to be mined will be taken.



Figure 2.1: (a) is a parameterized tree pattern, and (b) is an instantiation of (a).

Let $N \subseteq U$ 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.

In this thesis all trees we consider are rooted and ordered, unless stated otherwise.

Tree Patterns A *parameterized tree pattern* P is a tree whose nodes are called *variables*, and where additionally:

- Some variables may be marked as being *existential*;
- Some other variables may be marked as *parameters*;
- The variables of P that are neither existential nor parameters are called *distinguished*.

We will denote the set of existential variables by Π , the set of parameters by Σ , and the set of distinguished variables by Δ . To make clear that these sets belong to some parameterized tree pattern P we will use a subscript as in Π_P , Σ_P , or Δ_P .

A *parameter assignment* α , for a parameterized tree pattern P , is a mapping $\Sigma \rightarrow U$ which assigns data constants to the parameters.

An *instantiated tree pattern* is a pair (P, α) , with P a parameterized tree pattern and α a parameter assignment for P . We will also denote this by P^α .

When depicting parameterized tree patterns, existential nodes are indicated by labeling them with the symbol ' \exists ' and parameters are indicated by labeling them with the symbol ' σ '. When depicting instantiated tree patterns, parameters are indicated by directly writing down their parameter assignment.

Figure 2.1 gives an illustration.

Matching Recall that a *homomorphism* from a graph G_1 to a graph G_2 is a mapping μ from the nodes of G_1 to the nodes of G_2 that preserves edges, i.e., if $(i, j) \in G_1$ then $(\mu(i), \mu(j)) \in G_2$. We now define a *matching* of an instantiated tree pattern P^α

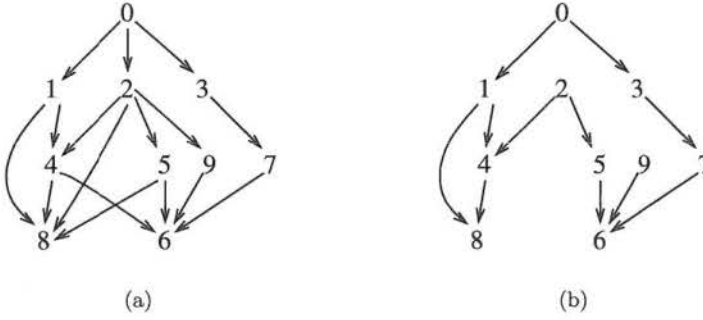


Figure 2.2: Two data graphs.

in a data graph G as a homomorphism μ from the underlying tree of P to G , with the constraint that for any parameter σ , if $\alpha(\sigma) = a$, then $\mu(\sigma)$ must be the node a . We denote the set $\{\mu|_{\Delta} : \mu \text{ is a matching of } P^{\alpha} \text{ in } G\}$ by $P^{\alpha}(G)$.

Frequency of a tree pattern The *frequency* of an instantiated tree pattern P^{α} in a data graph G , is formally defined as the cardinality of $P^{\alpha}(G)$. So, we count the number of matchings of P^{α} 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. For a given threshold k (a natural number) we say that P^{α} is *k-frequent* if its frequency is at least k . Often the threshold is understood implicitly, and then we talk simply about “frequent” patterns and denote the threshold by *minsup*.

Example. Take again the instantiated tree pattern P^{α} shown in Figure 2.1(b). Let us name the existential node by y ; let us name the parameter labeled 0 by z_1 ; the parameter labeled 8 by z_2 ; and the parameter labeled 6 by z_3 . The distinguished node already has the name x . Now let us apply P^{α} to the simple example data graph G shown in Figure 2.2(a). The following table lists all matchings of P^{α} in G :

	z_1	x	y	z_2	z_3
h_1	0	1	4	8	6
h_2	0	2	4	8	6
h_3	0	2	5	8	6

As required by the definition, all matchings match z_1 to 0, z_2 to 8, and z_3 to 6. Although there are three matchings, when determining the frequency of P^{α} in G , we only look at their value on x to distinguish them, as y is existential. So, h_2 and h_3 are identified as identical matchings when counting the number of matchings. In conclusion, the frequency of P^{α} in G is two, as x can be matched to the two different nodes 1 and 2. \square

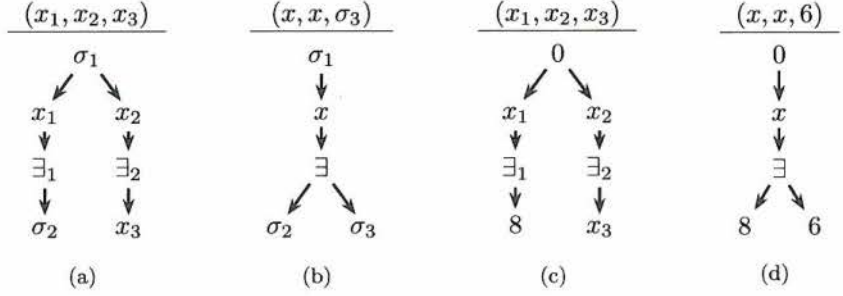


Figure 2.3: (a) and (b) are parameterized tree queries; (c) is an instantiation of (a); (d) is an instantiation of (b); and query (b) is ρ -contained in query (a)

2.2 Tree Query

Tree Queries A *parameterized tree query* Q is a pair (H, P) where:

1. P is a parameterized tree pattern, called the *body* of Q ;
2. H is a tuple of distinguished variables and parameters coming from P . All distinguished variables of P must appear at least once in H . We call H the *head* of Q .

A parameter assignment for Q is simply a parameter assignment for its body, and an *instantiated* tree query is then again a pair (Q, α) with Q a parameterized tree query and α a parameter assignment for Q . We will again denote this by Q^α .

When depicting tree queries, the head is given above a horizontal line, and the body below it. Illustrations are given in Figure 2.3.

Frequency of a tree query The *frequency* of an instantiated tree query $Q^\alpha = ((H, P), \alpha)$ in a data graph G , is defined as the frequency of the body P^α in G . When G is understood, we denote the frequency by $\text{Freq}(P^\alpha)$. For a given threshold k (a natural number) we say that Q^α is *k-frequent* if its frequency is at least k . Again, this threshold is often understood implicitly, and then we talk simply about “frequent” queries and denote the threshold by *minsup*.

2.2.1 Containment of Tree Queries

An important step towards our formal definition of tree-query association is the notion of *containment* among queries. Since queries are parameterized, a variation of the classical notion of containment [8, 44, 1] is needed in that we now need to specify a parameter correspondence.

First, we define the *answer set* of an instantiated tree query Q^α , with $Q = (H, P)$, in a data graph G as follows:

$$Q^\alpha(G) := \{\mu(H) \mid \mu \text{ is a matching of } P^\alpha \text{ in } G\}$$

Consider two parameterized tree queries Q_1 and Q_2 , with $Q_i = (H_i, P_i)$ for $i = 1, 2$. A *parameter correspondence* from Q_1 to Q_2 is any mapping $\rho : \Sigma_1 \rightarrow \Sigma_2$. We then say that a parameterized tree query Q_2 is ρ -*contained* in a parameterized tree query Q_1 , if for every α_2 , a parameter assignment for Q_2 , $Q_2^{\alpha_2}(G) \subseteq Q_1^{\alpha_2 \circ \rho}(G)$ for all data graphs G . In shorthand notation we write this as $Q_2 \subseteq_\rho Q_1$.

Containment as just defined is a semantical property, referring to all possible data graphs, and it is not immediately clear how one could decide this property syntactically. The required syntactical notion for this, is that of ρ -*containment mapping*, which we next define in two steps. For the tree queries Q_1 and Q_2 as above, and ρ a parameter correspondence from Q_1 to Q_2 :

1. A ρ -containment mapping from P_1 to P_2 is a homomorphism f from the underlying tree of P_1 to the underlying tree of P_2 , with the properties:
 - (a) f maps the distinguished nodes of P_1 to distinguished nodes or parameters of P_2 ; and
 - (b) $f|_{\Sigma_1} = \rho$, i.e., for each $z \in \Sigma_1$ we have $f(z) = \rho(z)$.
2. Finally, a ρ -containment mapping from Q_1 to Q_2 is a ρ -containment mapping f from P_1 to P_2 such that $f(H_1) = H_2$.

For later use, we note:

Lemma 1. *Consider three parameterized tree patterns P_1 , P_2 , and P_3 , a parameter correspondence $\rho_1 : \Sigma_1 \rightarrow \Sigma_2$, a parameter correspondence $\rho_2 : \Sigma_2 \rightarrow \Sigma_3$, a ρ_1 -containment mapping f_1 from P_1 to P_2 , and a ρ_2 -containment mapping f_2 from P_2 to P_3 . Then $f_2 \circ f_1$ is a $(\rho_2 \circ \rho_1)$ -containment mapping from P_1 to P_3 .*

Proof. We will show that:

1. $f_2 \circ f_1$ is homomorphism;
2. $f_2 \circ f_1$ maps distinguished nodes of P_1 to distinguished nodes or parameters of P_3 ; and
3. $(f_2 \circ f_1)|_{\Sigma_1} = \rho_2 \circ \rho_1$.

(1) Clearly $f_2 \circ f_1$ is a homomorphism since both f_1 and f_2 are homomorphisms, and it is already known that a composition of homomorphisms is a homomorphism.

(2) Consider $x_1 \in \Delta_1$, then there are two possibilities for $f_1(x_1)$:

1. $f_1(x_1) = x_2$, with $x_2 \in \Delta_2$. Then we know, since f_2 is a ρ_2 -containment mapping, that $f_2(x_2)$ is either a distinguished node $x_3 \in \Delta_3$, or a parameter $z_3 \in \Sigma_3$.
2. $f_1(x_1) = z_2$, with $z_2 \in \Sigma_2$. Then we know, since $f_2|_{\Sigma_2} = \rho_2$, that $f_2(z_2) = z_3$, with $z_3 \in \Sigma_3$.

Hence, we can conclude that $f_2 \circ f_1$ maps distinguished nodes of P_1 to distinguished nodes or parameters of P_3 .

(3) For each $z_1 \in \Sigma_1$, we have $f_2(f_1(z_1)) = \rho_2(\rho_1(z_1))$. Hence, $(f_2 \circ f_1)|_{\Sigma_1} = \rho_2 \circ \rho_1$. \square

From the theory of conjunctive database queries [8, 44, 1] we can derive the following:

Lemma 2. *Consider two parameterized tree queries Q_1 and Q_2 , with $Q_1 = (H_1, P_1)$ and $Q_2 = (H_2, P_2)$, and a parameter correspondence $\rho : \Sigma_1 \rightarrow \Sigma_2$. Then Q_2 is ρ -contained in Q_1 ($Q_2 \subseteq_\rho Q_1$), if and only if there exists a ρ -containment mapping from Q_1 to Q_2 .*

Proof. Let us start with the ‘only if’ direction. We first introduce the concept of a *freezing* of a parameterized tree query $Q = (H, P)$. Recall that U is the set of data constants from which the nodes of the data graph to be mined will be taken. A freezing β of P is then a one-to-one mapping from the nodes of P to U . We denote by $\text{freeze}_\beta(P)$ the data graph constructed from P by replacing each node n of P by $\beta(n)$, and we denote by $\text{freeze}_\beta(H)$ the tuple constructed from H by replacing each node n in H by the data constant $\beta(n)$.

For example, consider the parameterized tree query $Q = (H, P)$ in Figure 2.4(a). Then Figure 2.4(b) shows $\text{freeze}_\beta(P)$ and $\text{freeze}_\beta(H)$ for the freezing β given as follows: $x_1 \rightarrow c_1$; $x_2 \rightarrow c_2$; $\exists_3 \rightarrow c_3$; $x_4 \rightarrow c_4$; $x_5 \rightarrow c_5$; $\sigma_6 \rightarrow c_6$.

We can now continue with the proof of the ‘only if’ direction. Consider a freezing β from the nodes of P_2 to U . Note that $\beta|_{\Sigma_2}$ is a parameter assignment for Q_2 , and $\text{freeze}_\beta(H_2) \in Q_2^{\beta|_{\Sigma_2}}(\text{freeze}_\beta(P_2))$. Since $Q_2 \subseteq_\rho Q_1$, also $\text{freeze}_\beta(H_2) \in Q_1^{\beta|_{\Sigma_2} \circ \rho}(\text{freeze}_\beta(P_2))$. Hence, there must be a matching μ from $P_1^{\beta|_{\Sigma_2} \circ \rho}$ in $\text{freeze}_\beta(P_2)$ such that $\mu(H_1) = \text{freeze}_\beta(H_2)$. Now consider the function $g : \beta^{-1} \circ \mu$. We show that g is ρ -containment mapping from Q_1 to Q_2 :

1. Clearly, g is a homomorphism from P_1 to P_2 since μ is a homomorphism and β^{-1} is an isomorphism. Also the following properties hold for g :
 - (a) g maps distinguished nodes of P_1 to distinguished nodes or parameters of P_2 since $g(H_1) = H_2$ (as shown in (2)); and
 - (b) for each $z \in \Sigma_1$: $g(z) = \beta^{-1}(\mu(z)) = \beta^{-1}(\beta(\rho(z))) = \rho(z)$, hence $g|_{\Sigma_1} = \rho$
2. $g(H_1) = \beta^{-1}(\mu(H_1)) = \beta^{-1}(\text{freeze}_\beta(H_2)) = H_2$.

Hence, we conclude that g is a ρ -containment mapping from Q_1 to Q_2 .

Let us then look at the ‘if’ direction. Let h be the ρ -containment mapping from Q_1 to Q_2 . Consider an arbitrary parameter assignment α_2 for Q_2 . We must prove that for every data graph G , if $a \in Q_2^{\alpha_2}(G)$, then also $a \in Q_1^{\alpha_1 \circ \rho}(G)$. Consider such an arbitrary data graph G . Since, $a \in Q_2^{\alpha_2}(G)$, we know that there exists a matching μ of $P_2^{\alpha_2}$ in G such that $a = \mu(H_2)$. Now consider the function $g = \mu \circ h$. We show that g is a matching from $P_1^{\alpha_1 \circ \rho}$ in G and $a = g(H_1)$:

1. g is a homomorphism since both μ and h are homomorphisms; and
2. for each $z \in \Sigma_1$ we have $g(z) = \mu(h(z)) = \mu(\rho(z)) = \alpha_2(\rho(z))$.

So, g is indeed a matching of $P_1^{\alpha_1 \circ \rho}$ in G . Finally, we observe that $g(H_1) = \mu(h(H_1)) = \mu(H_2) = a$, as desired. \square

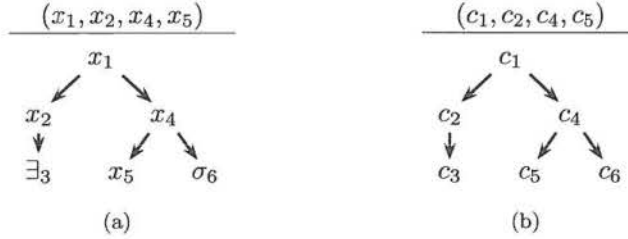


Figure 2.4: (b) is a freezing of the parameterized tree query in (a)

Checking for a containment mapping is evidently computable, and although the problem for general database conjunctive queries is NP-complete, our restriction to tree shapes allows for efficient checking, as we will see later.

Example. Consider the parameterized and instantiated tree queries shown in Figure 2.3. In the example data graph in Figure 2.2(a) the frequency of query (c) is 10 and that of query (d) is 2. Let Σ_a be the set of parameters of query (a), and let Σ_b be the set of parameters of query (b); then let the parameter correspondence $\rho : \Sigma_a \rightarrow \Sigma_b$ be as follows: $\sigma_1 \rightarrow \sigma_1$; $\sigma_2 \rightarrow \sigma_2$. A moment's reflection should convince the reader that (b) is ρ -contained in (a), and indeed a ρ -containment mapping f from (a) to (b) can be found as follows:

f	
σ_1	σ_1
x_1	x
x_2	x
\exists_1	\exists
\exists_2	\exists
σ_2	σ_2
x_3	σ_3

2.3 Tree-Query Association

Association Rules A *parameterized association rule* (pAR) is of the form $Q_1 \Rightarrow_\rho Q_2$, with Q_1 and Q_2 parameterized tree queries and ρ a parameter correspondence from Σ_1 to Σ_2 . We call a pAR *legal* if $Q_2 \subseteq_\rho Q_1$. We call Q_1 the left-hand side (lhs), and Q_2 the right-hand side (rhs). A *parameter assignment* α , for a pAR, is a mapping $\Sigma_2 \rightarrow U$ which assigns data constants to the parameters. An *instantiated association rule* (iAR) is a pair $(Q_1 \Rightarrow_\rho Q_2, \alpha)$, with $Q_1 \Rightarrow_\rho Q_2$ a pAR and α a parameter assignment for $Q_1 \Rightarrow_\rho Q_2$. Note that while α is only defined on the rhs, we can also apply it to the lhs by using ρ first.

Confidence The *confidence* of an iAR in a data graph G is defined as the frequency of $Q_2^{\alpha_2}$ divided by the frequency of $Q_1^{\alpha_2 \circ \rho}$. If the AR is legal, we know that the answer

set of $Q_2^{\alpha_2}$ is a subset of the answer set of $Q_1^{\alpha_2 \circ \rho}$, and hence the confidence equals precisely the proportion that the $Q_2^{\alpha_2}$ answer set takes up in the $Q_1^{\alpha_2 \circ \rho}$ answer set. Thus, our notions of a legal pAR and confidence are very intuitive and natural.

For a given threshold c (a rational number, $0 \leq c \leq 1$) we say that the iAR is *c-confident* in G if its confidence in G is at least c . Often the threshold is understood implicitly, and then we talk simply about “confident” iARs and denote the threshold by *minconf*.

Furthermore, the iAR is called *frequent* in G if $Q_2^{\alpha_2}$ is frequent in G . Note that if the iAR is legal and frequent, then also $Q_1^{\alpha_2 \circ \rho}$ is frequent, since the rhs is ρ -contained in the lhs.

Example. Continuing the previous example, we can see that we can form a legal pAR from the queries of Figure 2.3, with (a) the lhs and (b) the rhs and ρ as follows: $\sigma_1 \rightarrow \sigma_1$; $\sigma_2 \rightarrow \sigma_2$. We can also form an iAR with the tree queries in Figure 2.3(c) and Figure 2.3(d); the confidence of this iAR in the data graph of Figure 2.2(a) is $2/10$. Many more examples of ARs are given in Chapter 4 and Chapter 5.

2.4 Mining Problems

We are now finally ready to define the graph mining problems we want to solve.

2.4.1 Mining Tree Queries

Input: A data graph G ; a threshold *minsup*.

Output: All frequent instantiated tree queries $Q = ((H, P), \alpha)$.

In theory, however, there are infinitely many k -frequent tree queries, and even if we set an upper bound on the size of the patterns, there may be exponentially many. As an extreme example, if G is the complete graph on the set of nodes $\{1, \dots, n\}$, and $k \leq n$, then *any* instantiated pattern with all parameters assigned to values in $\{1, \dots, n\}$, and with 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. We introduce such an algorithm in Chapter 3.

2.4.2 Association Rule Mining

Input: A data graph G ; a threshold *minsup*; a parameterized tree query Q_{left} ; and a threshold *minconf*.

Output: All iARs $(Q_{\text{left}} \Rightarrow_{\rho} Q_{\text{right}}, \alpha)$ that are legal, frequent and confident in G .

In theory, however, there are infinitely many legal, frequent and confident association rules for a fixed lhs, and even if we set an upper bound on the size of the rhs, there may be exponentially many. 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. We introduce such an algorithm in Chapter 4.

3

Mining Tree Queries

In this Chapter we present an algorithm for mining frequent instantiated tree queries in a large data graph. We start by showing that we do not need to tackle the problem in its full generality in Section 3.1. Next, we give an overall approach of the presented algorithm in Section 3.2. The outer loop of the algorithm is explained in Section 3.3, and the inner loop in Section 3.4. We discuss equivalent tree queries, and show how to avoid generating them in Section 3.5. Finally, we explain how the results of the presented algorithm are managed in Section 3.6.

3.1 Problem Reduction

In this Section we show that, without loss of generality, we can focus on parameterized tree queries that are ‘pure’.

Pure Tree Queries To define this formally, assume that all possible variables (nodes of tree patterns) have been arranged in some fixed but arbitrary order. We then call a parameterized tree query $Q = (H, P)$ *pure* when H consists of the enumeration, in order and without repetitions, of all the distinguished variables of P . In particular H cannot contain parameters. We call H the *pure head* for P . As an illustration, the parameterized tree query in Figure 2.3(a) is pure, while the parameterized tree query in Figure 2.3(b) is not pure.

A parameterized tree query that is not pure can always be rewritten to a parameterized tree query that is pure, in such a way that all instantiations of the impure query correspond to instantiations of the pure query, with the same frequency. Indeed, take a parameterized tree query $Q = (H, P)$. We can purify Q by removing all parameters and repetitions of distinguished variables from H , and sorting H by the order on the variables. An illustration of this is given in Figure 3.1.

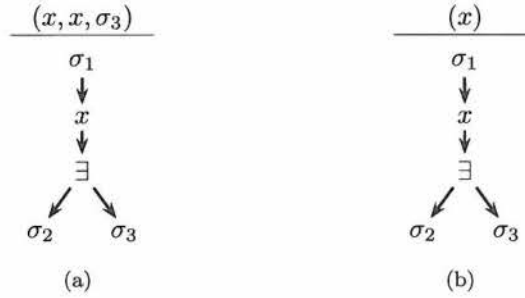


Figure 3.1: The parameterized tree query in (a) is an impure parameterized tree query, and the parameterized tree query in (b) is the purification of the parameterized tree query in (a), that expresses precisely the same information.

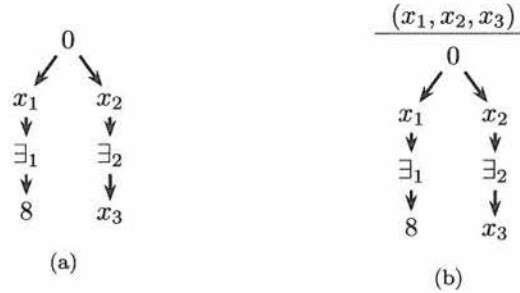


Figure 3.2: (b) is the pure instantiated tree query constructed from the instantiated tree pattern in (a)

We can conclude that it is sufficient to only consider pure instantiated tree queries. As a consequence, rather than mining tree *queries*, it suffices to mine for tree *patterns*, because the frequency of a query is nothing else then the frequency of his body, i.e., a pattern. An illustration is given in Figure 3.2.

3.2 Overall Approach

An overall outline of our tree-query mining algorithm is the following:

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

Inner loop: For each T , generate all instantiated tree patterns P^α based on T , and test their frequency.

The algorithm is incremental in the number of nodes of the pattern. We generate canonically ordered rooted trees of increasing sizes, avoiding the generation of isomorphic duplicates. It is well known how to do this efficiently [41, 32, 46, 9]. Note

that this generation of trees is in no way “levelwise” [33]. 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 [31]). 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. The outer loop of our algorithm will be explained in more detail in Section 3.3.

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” [43]. 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. The inner loop of our algorithm will be explained in more detail in Section 3.4.

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. This will be discussed in Section 3.5.

3.3 Outer Loop

In the outer loop we generate all possible trees of increasing sizes and we avoid trees that are isomorphic to previously generated ones. In fact, it is well known how to do this [41, 32, 46, 9]. What these procedures typically do is generating trees that are *canonically ordered* in the following sense. Given an (unordered) tree T , we can order the children of every node in some way, and call this an *ordering* of T . For example, Figure 3.3 shows two orderings of the same tree. From the different orderings of a tree T , we want to uniquely select one, to be the canonical ordering of T . For each such possible ordering of T , we can write down the *level sequence* of the resulting tree. This is actually a string representation of the resulting tree. This level sequence is as follows: if the tree has n nodes then this is a sequence of n numbers, where the i th number is the depth of the i th 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 ordering of T that yields the lexicographically maximal level sequence among all possible orderings of T .

As an example, in Figure 3.3, the left one is the canonical one.

3.4 Inner Loop

Let G be the data 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 instantiated tree patterns P^α based on T whose frequency in G is at least *minsup*.



Figure 3.3: Two orderings of the same tree. The left one is canonical.

This task lends itself naturally to a levelwise approach [33]. A natural choice for the specialization relation is suggested by an alternative notation for the patterns under consideration. Concretely, since the underlying tree T is fixed, any parameterized tree pattern P based on T is characterized by two parameters:

1. The set Π of existential nodes;
2. The set Σ of parameters.

Note that Π and Σ are disjoint.

Thus, a parameterized tree pattern P is completely characterized by the pair (Π, Σ) . An instantiation P^α of P is then represented by the triple (Π, Σ, α) . For two parameterized tree patterns $P_1 = (\Pi_1, \Sigma_1)$ and $P_2 = (\Pi_2, \Sigma_2)$ we now say that P_1 *specializes* P_2 if $\Pi_1 \supseteq \Pi_2$ and $\Sigma_1 \supseteq \Sigma_2$; and $\alpha_2 = \alpha_1|_{\Sigma_2}$. We also say that P_2 *generalizes* P_1 .

Parent An immediate generalization of a tree pattern is called a *parent*. Formally, let $P = (\Pi, \Sigma)$ and $P' = (\Pi', \Sigma')$ be parameterized tree patterns based on T . We say that P' is a *parent* of P if:

- (i) $\Sigma = \Sigma'$ and $\Pi = \Pi' \cup \{y\}$ for some node $y \notin \Pi'$; or
- (ii) $\Pi = \Pi'$ and $\Sigma = \Sigma' \cup \{z\}$ for some node $z \notin \Sigma'$.

From the following lemma, it follows that specialized patterns have a lower frequency, as expected for a specialization relation:

Lemma 3. *Let P and P' be parameterized tree patterns such that P' is a parent of P . Let P^α be an instantiation of P , and let $\alpha' = \alpha|_{\Sigma'}$. Then $\text{Freq}(P^\alpha) \leq \text{Freq}(P'^{\alpha'})$.*

Proof. We will show that $\#P^\alpha(G) \leq \#P'^{\alpha'}(G)$ by defining an injection $I : P^\alpha(G) \rightarrow P'^{\alpha'}(G)$.

Since P' is a parent of P , we know that $\Delta' = \Delta \cup \{u\}$ where u is either an existential node or a parameter of P . Note that each $\mu \in P^\alpha(G)$ is of the form $\bar{\mu}|_\Delta$ for some matching $\bar{\mu}$ of $P^\alpha \in G$. For each μ in $P^\alpha(G)$, we fix arbitrarily $\bar{\mu}$. Now we define $I(\mu) := \bar{\mu}|_{\Delta'}$. To see that I is an injection, let $\mu_1, \mu_2 \in P^\alpha(G)$ and suppose that $I(\mu_1) = I(\mu_2)$. In other words, $\bar{\mu}_1|_{\Delta'} = \bar{\mu}_2|_{\Delta'}$. In particular, $\mu_1 = \bar{\mu}_1|_\Delta = \bar{\mu}_2|_\Delta = \mu_2$, as desired.

Hence, we can conclude that $\#P^\alpha(G) \leq \#P'^{\alpha'}(G)$ and that $\text{Freq}(P^\alpha) \leq \text{Freq}(P'^{\alpha'})$. \square

The above lemma suggests the following definition of specialization among *instantiated* tree patterns: we say that $(\Pi_1, \Sigma_1, \alpha_1)$ is a specialization of $(\Pi_2, \Sigma_2, \alpha_2)$ if the parameterized tree pattern (Π_1, Σ_1) is a specialization of the parameterized tree pattern (Π_2, Σ_2) , and $\alpha_2 = \alpha_1|_{\Sigma_2}$.

Intuitively, the previous lemma then expresses that the frequency of an instantiated tree pattern is always at most the frequency of any of its instantiated parents.

3.4.1 Candidate Generation

Candidate pattern A *candidate pattern* is an instantiated tree pattern whose frequency is not yet determined, but all whose generalizations are known to be frequent.

Using the specialization relation and the definition for a candidate pattern we explain how the levelwise search for frequent instantiated tree patterns will go.

Levelwise search We start with the most general instantiated tree pattern $P = (\emptyset, \emptyset, \emptyset)$, and we progressively consider more specific patterns. The search has the typical property that, in each new iteration, new *candidate* patterns are generated; the frequency of all newly discovered candidate patterns is determined; and the process repeats.

There are many different instantiations to consider for each parameterized tree pattern. Hence, to generate candidate patterns in an efficient manner, we propose the use of *candidacy tables* and *frequency tables*. These candidacy and frequency tables allow us to generate all frequent instantiations for a particular parameterized tree pattern in parallel. A frequency table contains all frequent instantiations for a particular parameterized tree pattern.

Formally, for any parameterized pattern $P = (\Pi, \Sigma)$, we define:

$$CanTab_{\Pi, \Sigma} = \{\alpha \mid P^\alpha \text{ is a candidate instantiated tree pattern}\}$$

$$FreqTab_{\Pi, \Sigma} = \{\alpha \mid P^\alpha \text{ is a frequent instantiated tree pattern}\}$$

Technically, the table has columns for the different parameters, plus a column **freq**. Note that when $\Sigma = \emptyset$, i.e., P has no parameters, this is a single-column, single-row table containing just the frequency of P . This still makes sense and can be interpreted as boolean values; for example, if $FreqTab_{\Pi, \emptyset}$ contains the empty tuple, then the pattern $(\Pi, \emptyset, \emptyset)$ is frequent; if the table is empty, the pattern is not frequent. Of course in practice, all frequency tables for parameterless patterns can be combined into a single table. All frequency tables are kept in a relational database.

The following crucial lemma shows that these tables can be populated efficiently.

Join Lemma. A parameter assignment α 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 $\alpha|_{\Sigma'} \in FreqTab_{\Pi', \Sigma'}$;
- (ii) If $\Sigma = \Sigma'$, then $\alpha \in FreqTab_{\Pi', \Sigma'}$.

Proof. For the ‘only-if’ direction: By definition of a candidacy table, if $\alpha \in CanTab_{\Pi, \Sigma}$, then all generalizations of (Π, Σ, α) are frequent. In particular, for all parents (Π', Σ') of (Π, Σ) , we know that $(\Pi', \Sigma', \alpha|_{\Sigma'})$ is frequent, since parents are generalizations.

For the ‘if’ direction, we must show that all generalizations of (Π, Σ, α) are frequent. Consider such a generalization $(\Pi_{g_1}, \Sigma_{g_1}, \alpha|_{\Sigma_{g_1}})$. Let us denote the parent relation by \geq_p . Then there is a sequence of parent patterns: $(\Pi_{g_1}, \Sigma_{g_1}) \geq_p (\Pi_{g_2}, \Sigma_{g_2}) \geq_p \dots \geq_p (\Pi', \Sigma')$. And we have: $\text{Freq}(\Pi_{g_1}, \Sigma_{g_1}, \alpha|_{\Sigma_{g_1}}) \geq \text{Freq}(\Pi_{g_2}, \Sigma_{g_2}, \alpha|_{\Sigma_{g_2}}) \geq \dots \geq \text{Freq}(\Pi', \Sigma', \alpha|_{\Sigma'}) \geq \text{minsup}$. The last inequality is given by (i) or (ii), the other inequalities are given by Lemma 3. \square

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 $\text{CanTab}_{\emptyset, \{z\}}$ as the table with a single column z , holding all nodes of the data graph G being mined.

3.4.2 Frequency Counting using SQL

The search process starts by determining the frequency of the underlying tree $T = (\emptyset, \emptyset)$; indeed, formally this amounts to computing $\text{FreqTab}_{\emptyset, \emptyset}$. Similarly, for each parameterized tree pattern $P = (\Pi, \emptyset)$ with $\Pi \neq \emptyset$, all we can do is determine its frequency, except that here, we do this only on condition that its parent patterns 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 data graph G that is being mined is stored in the relational database system as well, in the form of a table $G(\text{from}, \text{to})$. 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 pattern in SQL, and determining its count (eliminating duplicates). Since our patterns are in fact conjunctive queries (or datalog rules) known from database research [8, 44, 1]. They can easily be translated in SQL:

- The FROM-clause consists of all table references of the form G as G_{ij} , for all edges $x_i \rightarrow x_j$ in T .
- The WHERE-clause consists of all equalities of the form $G_{ij}.\text{from} = G_{ik}.\text{from}$ as well of equalities of the form $G_{ij}.\text{to} = G_{jh}.\text{from}$.
- The SELECT-clause is of the form SELECT DISTINCT and consists of all column references of the form $G_{ij}.\text{to}$ when x_{ij} is a distinguished node in P , plus one reference of the form $G_{ik}.\text{from}$ if the root node is distinguished.

The SQL query for the tree in Figure 3.4 with $\Pi = \{x_2\}$ and $\Sigma = \emptyset$ is as follows:

```
E = SELECT G12.from, G23.to, G24.to
FROM G as G12, G as G23, G as G24
WHERE G12.to = G23.from AND G12.to = G24.from
```

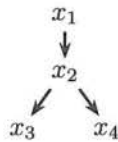


Figure 3.4: Illustration on translating a tree pattern without parameters in SQL.

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 instantiated tree patterns in parallel! We proceed as follows:

1. We formulate the pattern (Π, \emptyset) in SQL, and call the resulting expression E ;
2. 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 patterns are counted.

For instance, the SQL query to compute the frequency table for the tree in Figure 3.4, with $\Pi = \{x_2\}$ and $\Sigma = \{x_1, x_3\}$, with E as above, is as follows:

```

SELECT E.x1, E.x3, COUNT(*)
FROM E, CanTab_{x2}, {x1, x3} CT
WHERE E.x1 = CT.x1 AND E.x3 = CT.x3
GROUP BY E.x1, E.x3 HAVING COUNT(*) >= minsup

```

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

3.4.3 The Algorithm

Putting everything together so far, the algorithm is given in Algorithm 1. In outline it is a double Apriori algorithm [2], where the sets Π form one dimension of itemsets, and the sets Σ another. A graphical illustration of the algorithm is given in Figure 3.5. In this illustration we use *tries* (or prefix-trees) to store the itemsets. A trie [4, 6, 28] is commonly used in implementations of the Apriori algorithm.

3.4.4 Example Run

In this Section we give an example run of the proposed algorithm. Consider the example data graph G in Figure 3.6(a); the tree T in Figure 3.6(b); and let the minimum support threshold be 3.

The example run is then given in Table 3.1.

Algorithm 1 Levelwise search for frequent tree patterns.

```

1: for each unordered, rooted tree  $T$  do
2:    $X :=$  set of nodes of  $T$ 
3:    $p := 0$ ;  $\mathcal{P}_0 := \{\emptyset\}$ 
4:   repeat
5:     for each  $\Pi \in \mathcal{P}_p$  do
6:       Compute  $FreqTab_{\Pi, \emptyset}$  in SQL
7:       if  $FreqTab_{\Pi, \emptyset} \neq \emptyset$  then
8:          $s := 1$ 
9:          $\mathcal{S}_1 := \{\{z\} \mid z \in X - \Pi\}$ 
10:        repeat
11:          for each  $\Sigma \in \mathcal{S}_s$  do
12:            if  $p = 0$  and  $s = 1$  then
13:               $CanTab_{\Pi, \Sigma} :=$  set of nodes of  $G$ 
14:            else
15:               $CanTab_{\Pi, \Sigma} := \bowtie \{FreqTab_{\Pi', \Sigma'} \mid (\Pi', \Sigma') \text{ parent of } (\Pi, \Sigma)\}$ 
16:            end if
17:            Compute  $FreqTab_{\Pi, \Sigma}$  in SQL
18:            if  $FreqTab_{\Pi, \Sigma} = \emptyset$  then
19:              remove  $\Sigma$  from  $\mathcal{S}_s$   $\{\Sigma \text{ is pruned away}\}$ 
20:            end if
21:          end for
22:           $\mathcal{S}_{s+1} := \{\Sigma \subseteq X - \Pi \mid \#\Sigma = s + 1$ 
23:                                and each  $s$ -subset of  $\Sigma$  is in  $\mathcal{S}_s\}$ 
24:           $s := s + 1$ 
25:        until  $\mathcal{S}_s = \emptyset$ 
26:      else
27:        remove  $\Pi$  from  $\mathcal{P}_p$   $\{\Pi \text{ is pruned away}\}$ 
28:      end if
29:    end for
30:     $\mathcal{P}_{p+1} := \{\Pi \subseteq X \mid \#\Pi = p + 1 \text{ and each } p\text{-subset of } \Pi \text{ is in } \mathcal{P}_p\}$ 
31:     $p := p + 1$ 
32:  until  $\mathcal{P}_p = \emptyset$ 
33: end for
  
```

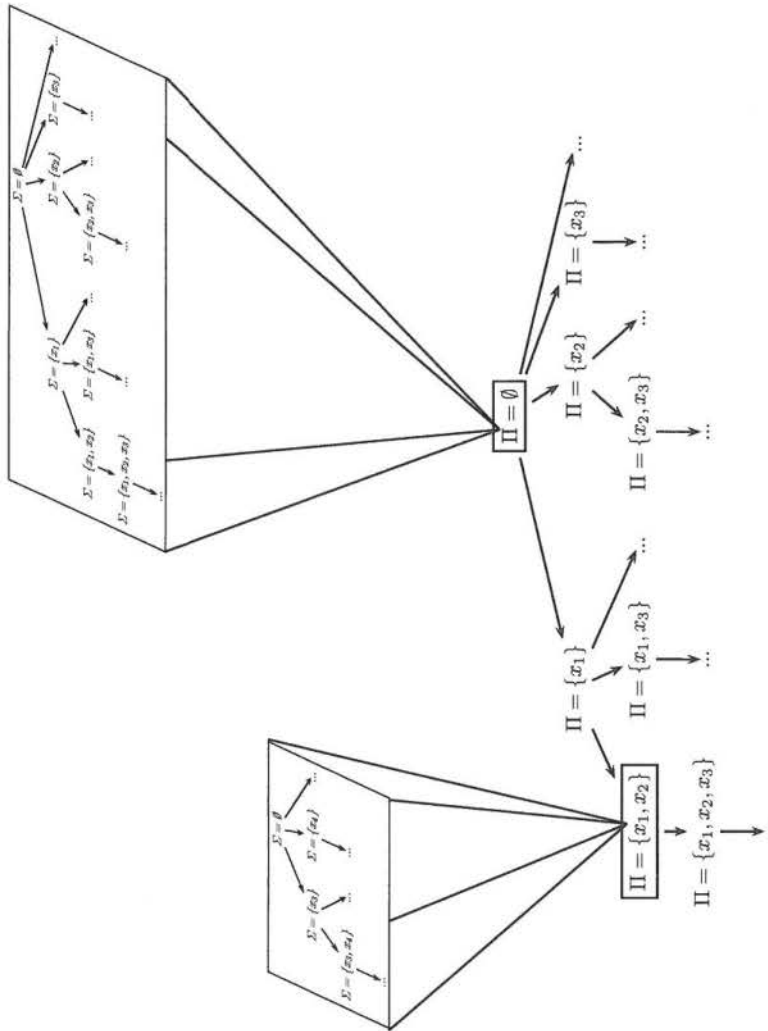


Figure 3.5: Illustration of the algorithm in Algorithm 1

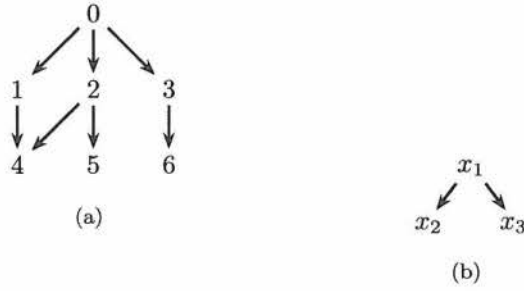
Figure 3.6: Data graph G and tree T for the example run in Section 3.4.4.

Table 3.1: Example run of the inner loop for the tree and data graph in Figure 3.6

Π	Σ	P	$CanTab$	$FreqTab$												
\emptyset	\emptyset	<div>x_1 $\swarrow \quad \searrow$ $x_2 \quad x_3$</div>		<table><tr><th>$Freq$</th></tr><tr><td>15</td></tr></table>	$Freq$	15										
$Freq$																
15																
\emptyset	$\{x_1\}$	<div>σ_1 $\swarrow \quad \searrow$ $x_2 \quad x_3$</div>	nodes of G	<table><tr><th>σ_1</th><th>$Freq$</th></tr><tr><td>0</td><td>9</td></tr><tr><td>2</td><td>4</td></tr></table>	σ_1	$Freq$	0	9	2	4						
σ_1	$Freq$															
0	9															
2	4															
\emptyset	$\{x_2\}$	<div>x_1 $\swarrow \quad \searrow$ $\sigma_2 \quad x_3$</div>	nodes of G	<table><tr><th>σ_2</th><th>$Freq$</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr><tr><td>4</td><td>3</td></tr></table>	σ_2	$Freq$	1	3	2	3	3	3	4	3		
σ_2	$Freq$															
1	3															
2	3															
3	3															
4	3															
\emptyset	$\{x_3\}$	<div>x_1 $\swarrow \quad \searrow$ $x_2 \quad \sigma_3$</div>	nodes of G	<table><tr><th>σ_3</th><th>$Freq$</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr><tr><td>4</td><td>3</td></tr></table>	σ_3	$Freq$	1	3	2	3	3	3	4	3		
σ_3	$Freq$															
1	3															
2	3															
3	3															
4	3															
\emptyset	$\{x_1, x_2\}$	<div>σ_1 $\swarrow \quad \searrow$ $\sigma_2 \quad x_3$</div>	$FreqTab_{\emptyset, \{x_1\}}$ $\bowtie FreqTab_{\emptyset, \{x_2\}}$	<table><tr><th>σ_1</th><th>σ_2</th><th>$Freq$</th></tr><tr><td>0</td><td>1</td><td>3</td></tr><tr><td>0</td><td>2</td><td>3</td></tr><tr><td>0</td><td>3</td><td>3</td></tr></table>	σ_1	σ_2	$Freq$	0	1	3	0	2	3	0	3	3
σ_1	σ_2	$Freq$														
0	1	3														
0	2	3														
0	3	3														
\emptyset	$\{x_1, x_3\}$	<div>σ_1 $\swarrow \quad \searrow$ $x_2 \quad \sigma_3$</div>	$FreqTab_{\emptyset, \{x_1\}}$ $\bowtie FreqTab_{\emptyset, \{x_3\}}$	<table><tr><th>σ_1</th><th>σ_3</th><th>$Freq$</th></tr><tr><td>0</td><td>1</td><td>3</td></tr><tr><td>0</td><td>2</td><td>3</td></tr><tr><td>0</td><td>3</td><td>3</td></tr></table>	σ_1	σ_3	$Freq$	0	1	3	0	2	3	0	3	3
σ_1	σ_3	$Freq$														
0	1	3														
0	2	3														
0	3	3														
\emptyset	$\{x_2, x_3\}$	<div>x_1 $\swarrow \quad \searrow$ $\sigma_2 \quad \sigma_3$</div>	$FreqTab_{\emptyset, \{x_2\}}$ $\bowtie FreqTab_{\emptyset, \{x_3\}}$	\emptyset												

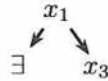
\emptyset	$\{x_1, x_2, x_3\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \searrow \\ \sigma_2 \quad \sigma_3 \end{array}$	Pruned									
$\{x_1\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \searrow \\ x_2 \quad x_3 \end{array}$		<table><tr><th>Freq</th></tr><tr><td>14</td></tr></table>	Freq	14						
Freq												
14												
$\{x_1\}$	$\{x_2\}$	$\begin{array}{c} \exists \\ \swarrow \searrow \\ \sigma_2 \quad x_3 \end{array}$	$FreqTab_{\emptyset, \{x_2\}}$ $\bowtie FreqTab_{\{x_1\}, \emptyset}$	<table><tr><th>σ_2</th><th>Freq</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr></table>	σ_2	Freq	1	3	2	3	3	3
σ_2	Freq											
1	3											
2	3											
3	3											
$\{x_1\}$	$\{x_3\}$	$\begin{array}{c} \exists \\ \swarrow \searrow \\ x_2 \quad \sigma_3 \end{array}$	$FreqTab_{\emptyset, \{x_3\}}$ $\bowtie FreqTab_{\{x_1\}, \emptyset}$	<table><tr><th>σ_3</th><th>Freq</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr></table>	σ_3	Freq	1	3	2	3	3	3
σ_3	Freq											
1	3											
2	3											
3	3											
$\{x_1\}$	$\{x_2, x_3\}$	$\begin{array}{c} \exists \\ \swarrow \searrow \\ \sigma_2 \quad \sigma_3 \end{array}$	Pruned									
$\{x_2\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \searrow \\ \exists \quad x_3 \end{array}$		<table><tr><th>Freq</th></tr><tr><td>7</td></tr></table>	Freq	7						
Freq												
7												
$\{x_2\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \searrow \\ \exists \quad x_3 \end{array}$	$FreqTab_{\emptyset, \{x_1\}}$ $\bowtie FreqTab_{\{x_2\}, \emptyset}$	<table><tr><th>σ_1</th><th>Freq</th></tr><tr><td>0</td><td>3</td></tr></table>	σ_1	Freq	0	3				
σ_1	Freq											
0	3											
$\{x_2\}$	$\{x_3\}$	$\begin{array}{c} x_1 \\ \swarrow \searrow \\ \exists \quad \sigma_3 \end{array}$	$FreqTab_{\emptyset, \{x_3\}}$ $\bowtie FreqTab_{\{x_2\}, \emptyset}$	\emptyset								
$\{x_2\}$	$\{x_1, x_3\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \searrow \\ \exists \quad \sigma_3 \end{array}$	Pruned									
$\{x_3\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \searrow \\ x_2 \quad \exists \end{array}$		<table><tr><th>Freq</th></tr><tr><td>7</td></tr></table>	Freq	7						
Freq												
7												
$\{x_3\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \searrow \\ x_2 \quad \exists \end{array}$	$FreqTab_{\emptyset, \{x_1\}}$ $\bowtie FreqTab_{\{x_3\}, \emptyset}$	<table><tr><th>σ_1</th><th>Freq</th></tr><tr><td>0</td><td>3</td></tr></table>	σ_1	Freq	0	3				
σ_1	Freq											
0	3											
$\{x_3\}$	$\{x_2\}$	$\begin{array}{c} x_1 \\ \swarrow \searrow \\ \sigma_2 \quad \exists \end{array}$	$FreqTab_{\emptyset, \{x_2\}}$ $\bowtie FreqTab_{\{x_3\}, \emptyset}$	\emptyset								

$\{x_3\}$	$\{x_1, x_2\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \searrow \\ \sigma_2 \quad \exists \end{array}$	Pruned			
$\{x_1, x_2\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \searrow \\ \exists \quad x_3 \end{array}$		<table><tr><td><i>Freq</i></td></tr><tr><td>6</td></tr></table>	<i>Freq</i>	6
<i>Freq</i>						
6						
$\{x_1, x_2\}$	$\{x_3\}$	$\begin{array}{c} \exists \\ \swarrow \searrow \\ \exists \quad \sigma_3 \end{array}$	Pruned			
$\{x_1, x_3\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \searrow \\ x_2 \quad \exists \end{array}$		<table><tr><td><i>Freq</i></td></tr><tr><td>6</td></tr></table>	<i>Freq</i>	6
<i>Freq</i>						
6						
$\{x_1, x_3\}$	$\{x_2\}$	$\begin{array}{c} \exists \\ \swarrow \searrow \\ \sigma_2 \quad \exists \end{array}$	Pruned			
$\{x_2, x_3\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \searrow \\ \exists \quad \exists \end{array}$		<table><tr><td><i>Freq</i></td></tr><tr><td>4</td></tr></table>	<i>Freq</i>	4
<i>Freq</i>						
4						
$\{x_2, x_3\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \searrow \\ \exists \quad \exists \end{array}$	$FreqTab_{\{x_2, x_3\}, \emptyset}$ $\bowtie FreqTab_{\{x_2\}, \{x_1\}}$ $\bowtie FreqTab_{\{x_3\}, \{x_1\}}$	\emptyset		

3.5 Equivalence among Tree Patterns

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

As an example of duplicate work, consider the parameterized tree pattern P_1 from the example run in Section 3.4.4 ($\Pi = \{x_2\}$ and $\Sigma = \emptyset$):



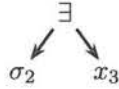
and the parameterized tree pattern P_2 :



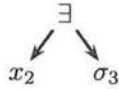
Clearly, P_1 and P_2 have the same answer set for all data graphs G , up to renaming of the distinguished variables (x_2 by x_3). However, these patterns have different

underlying trees, and hence Algorithm 1 will compute the answer set for both patterns (line 6). The answer set of P_2 is computed before the answer set of P_1 , since our algorithm is incremental in the number of nodes of T . Hence, we can conclude that our algorithm performs some duplicate work which we want to avoid.

Another example of duplicate work our algorithm performs: Consider the parameterized tree pattern P_3 from the example run in Section 3.4.4 ($\Pi = \{x_1\}$ and $\Sigma = \{x_2\}$):



and the parameterized tree pattern P_4 also from the example run in Section 3.4.4 ($\Pi = \{x_1\}$ and $\Sigma = \{x_3\}$):



As one can see in Section 3.4.4, these two parameterized patterns have the same instantiations for all data graphs G , up to renaming of the parameters (σ_2 by σ_3), and for each instantiation, the same answer set for all data graphs G , up to renaming of the distinguished variables (x_3 by x_2). However, when we look at the outline of our algorithm in Algorithm 1, we see that for both patterns the candidacy and frequency tables are computed between line 11 and line 17. Hence, we can conclude again that our algorithm performs duplicate work that we want to avoid.

In the rest of this Section we formalize the duplicate work our algorithm performs, and we make a number of modifications to the algorithm described so far, so as to avoid the duplicate work.

3.5.1 Equivalency

Intuitively we call two parameterized tree patterns *equivalent* if they have the same instantiations, and for each instantiation the same answer set for all data graphs G , up to renaming of the parameters and the distinguished variables. For instance, the parameterized tree patterns P_1 and P_2 from above we call equivalent, as the tree patterns P_3 and P_4 from above.

To define equivalent parameterized tree patterns formally we introduce the notion of (δ, ρ) -equivalence.

(δ, ρ) -Equivalence Let P_1 and P_2 be two parameterized tree patterns and ρ a parameter correspondence from P_1 to P_2 (recall Section 2.2). We define an *answer set correspondence* from P_1 to P_2 as any mapping $\delta : \Delta_1 \rightarrow \Delta_2$. Furthermore, assume that δ and ρ are bijections. We then say that P_1 and P_2 are (δ, ρ) -equivalent, denoted by $P_1 \equiv_{\rho}^{\delta} P_2$, if for all data graphs G , and all parameter assignments $\alpha_2 : \Sigma_2 \rightarrow U$, we have $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$, where $P_2^{\alpha_2}(G) \circ \delta$ denotes the set $\{f \circ \delta : f \in P_2^{\alpha_2}(G)\}$.

For example, consider the two parameterized tree patterns in Figure 3.7, and let $\rho : \Sigma_a \rightarrow \Sigma_b$ be as follows:

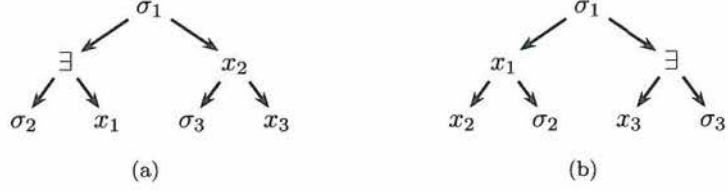


Figure 3.7: Two equivalent parameterized tree patterns.

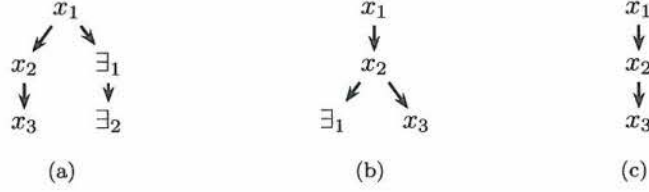


Figure 3.8: Three equivalent parameterized tree patterns.

ρ	
σ_1	σ_1
σ_2	σ_3
σ_3	σ_2

and let $\delta : \Delta_a \rightarrow \Delta_b$ be as follows:

δ	
x_1	x_3
x_2	x_1
x_3	x_2

The two parameterized tree patterns are clearly (δ, ρ) -equivalent, as are the three parameterized tree patterns shown in Figure 3.8 with an empty parameter correspondence ρ and δ the identity.

The parameter correspondence ρ is a bijection in the definition of ρ -equivalence, since intuitively we want equivalent parameterized tree patterns to have essentially the same set of instantiations. Hence it is necessary that the tree patterns have the same number of parameters. Intuitively we also want equivalent tree patterns to have the same answer sets up to renaming of the distinguished variables. That is the reason why an answer set correspondence is introduced that is a bijection.

We then define equivalent parameterized tree patterns as follows:

Equivalent parameterized tree patterns We call two parameterized tree patterns P_1 and P_2 *equivalent* if P_1 is (δ, ρ) -equivalent with P_2 for some bijective parameter correspondence ρ and some bijective answer set correspondence δ .



Figure 3.9: Two equivalent parameterized tree patterns with more than one possibility for a parameter and answer set correspondence.

Note that there can exist more than one parameter correspondence ρ and more than one answer set correspondence δ for which the two parameterized tree patterns are (δ, ρ) -equivalent. An illustration of this is given in Figure 3.9. Let $\rho_1 : \Sigma_a \rightarrow \Sigma_b$, $\delta_1 : \Delta_a \rightarrow \Delta_b$, $\rho_2 : \Sigma_a \rightarrow \Sigma_b$ and $\delta_2 : \Delta_a \rightarrow \Delta_b$ be as follows: ρ_1 is the identity; δ_1 is the identity and

ρ_2	
σ_1	σ_2
σ_2	σ_1

δ_2	
x_1	x_1
x_2	x_4
x_3	x_5
x_4	x_2
x_5	x_3

Then the two tree patterns in Figure 3.9 are clearly (δ_1, ρ_1) -equivalent and (δ_2, ρ_2) -equivalent.

Equivalence as just defined is a semantical property, referring to all possible data graphs, and it is not immediately clear how one could decide this property syntactically. The required syntactical notion is given by the following Lemma and Corollary.

Lemma 4. Consider two parameterized tree patterns P_1 and P_2 , $\delta : \Delta_1 \rightarrow \Delta_2$ a bijective answer set correspondence, and $\rho : \Sigma_1 \rightarrow \Sigma_2$ a bijective parameter correspondence. Then $P_1 \equiv_{\rho}^{\delta} P_2$ if and only if we have the following containment relations among the tree queries (H_1, P_1) and $(\delta(H_1), P_2)$, with H_1 the pure head of P_1 (cfr. Section 3.1):

1. $(\delta(H_1), P_2) \subseteq_{\rho} (H_1, P_1)$; and
2. $(H_1, P_1) \subseteq_{\rho^{-1}} (\delta(H_1), P_2)$

Proof. Let us start with the if direction. We need to prove that for every parameter assignment α_2 for P_2 , and every data graph G that $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$. We know that $(\delta(H_1), P_2)^{\alpha_2}(G) \subseteq (H_1, P_1)^{\alpha_2 \circ \rho}(G)$ since $(\delta(H_1), P_2) \subseteq_{\rho} (H_1, P_1)$. We may rewrite this as: $P_2^{\alpha_2}(G) \circ \delta \subseteq P_1^{\alpha_2 \circ \rho}(G)$ since H_1 is an enumeration of Δ_1 .

We also know that $(H_1, P_1)^{\alpha_1}(G) \subseteq (\delta(H_1), P_2)^{\alpha_1 \circ \rho^{-1}}(G)$ for every parameter assignment α_1 for P_1 since $(H_1, P_1) \subseteq_{\rho^{-1}} (\delta(H_1), P_2)$. Now take $\alpha_1 = \alpha_2 \circ \rho$. We then have $(H_1, P_1)^{\alpha_2 \circ \rho}(G) \subseteq (\delta(H_1), P_2)^{\alpha_2}(G)$. Again since H_1 is an enumeration of Δ_1 we may rewrite this as: $P_1^{\alpha_2 \circ \rho}(G) \subseteq P_2^{\alpha_2}(G) \circ \delta$. Hence we can conclude that $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$.

Let us then look at the only-if direction. To prove that $(\delta(H_1), P_2) \subseteq_\rho (H_1, P_1)$, we will show that for every α_2 parameter assignment for P_2 , and every data graph G , we have $(\delta(H_1), P_2)^{\alpha_2}(G) \subseteq (H_1, P_1)^{\alpha_2 \circ \rho}(G)$. Since $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$, we have $(\delta(H_1), P_2)^{\alpha_2}(G) = (H_1, P_1)^{\alpha_2 \circ \rho}(G)$, and hence clearly $(\delta(H_1), P_2)^{\alpha_2}(G) \subseteq (H_1, P_1)^{\alpha_2 \circ \rho}(G)$.

To prove that $(H_1, P_1) \subseteq_{\rho^{-1}} (\delta(H_1), P_2)$, we will show that for every α_1 parameter assignment for P_1 , and every data graph G , we have

$$(H_1, P_1)^{\alpha_1}(G) \subseteq (\delta(H_1), P_2)^{\alpha_1 \circ \rho^{-1}}(G)$$

We know that for every α_2 parameter assignment for P_2 , we have $P_2^{\alpha_2}(G) \circ \delta = P_1^{\alpha_2 \circ \rho}(G)$. Now take $\alpha_2 = \alpha_1 \circ \rho^{-1}$. We then have $P_2^{\alpha_1 \circ \rho^{-1}}(G) \circ \delta = P_1^{\alpha_1}(G)$, hence $(\delta(H_1), P_2)^{\alpha_1 \circ \rho^{-1}}(G) = (H_1, P_1)^{\alpha_1}(G)$, hence clearly

$$(H_1, P_1)^{\alpha_1}(G) \subseteq (\delta(H_1), P_2)^{\alpha_1 \circ \rho^{-1}}(G)$$

□

Corollary 1. *Consider two parameterized tree patterns P_1 and P_2 . Then P_1 is equivalent with P_2 if and only if there exist:*

1. a bijective answer set correspondence $\delta : \Delta_1 \rightarrow \Delta_2$;
2. a bijective parameter correspondence $\rho : \Sigma_1 \rightarrow \Sigma_2$;
3. a ρ -containment mapping $f_1 : (H_1, P_1) \rightarrow (\delta(H_1), P_2)$; and
4. a ρ^{-1} -containment mapping $f_2 : (\delta(H_1), P_2) \rightarrow (H_1, P_1)$,

with H_1 the pure head for P_1 .

Of course, we want to avoid that our algorithm considers some parameterized tree pattern P_2 if it is equivalent to an earlier considered parameterized tree pattern P_1 . Since our algorithm generates trees in increasing sizes, there are two cases to consider:

Case A: P_1 has fewer nodes than P_2 .

Case B: P_1 and P_2 have the same number of nodes.

Armed with the above Lemma and Corollary, we can now analyze the above two cases.

3.5.2 Case A: Redundancy Checking

Let us start by defining the notion of a redundancy.

Redundant subtree A *redundant subtree* R , is a subtree of a parameterized tree pattern P , such that removing R from P yields a parameterized tree pattern P' that is equivalent with P .

For example, the first two parameterized tree patterns in Figure 3.8 indeed contain a redundant subtree.

The following lemma shows that two parameterized tree patterns with different numbers of nodes can only be equivalent if the largest one contains redundant subtrees:

Lemma 5. *Consider two parameterized tree patterns P and P' , and the number of nodes of P' is smaller than the number of nodes of P . Then P can only be equivalent with P' if P contains redundant subtrees.*

Proof. Since P and P' are equivalent we know from Corollary 1 that the following exist:

1. an answer set correspondence $\delta : \Delta \rightarrow \Delta'$ that is a bijection;
2. a parameter correspondence $\rho : \Sigma \rightarrow \Sigma'$ that is a bijection;
3. a ρ -containment mapping $f_1 : (H, P) \rightarrow (\delta(H), P')$; and
4. a ρ^{-1} -containment mapping $f_2 : (\delta(H), P') \rightarrow (H, P)$,

with H the pure head for P . Since the number of nodes of P' is smaller than the number of nodes of P , we know that some subtrees R of P are not in the range of f_2 . We will prove that these subtrees R are redundant subtrees, by showing that P and $P - R$ are equivalent.

Since the containment mappings f_1 and f_2 exist, we know that in particular the following containment mappings will exist:

1. $g_1 = f_1|_{P-R}$, a ρ -containment mapping from $(H, P - R)$ to $(\delta(H), P')$, and
2. $g_2 = f_2$, a ρ^{-1} -containment mapping from $(\delta(H), P')$ to $(H, P - R)$.

with δ and ρ as above.

Let us now look at the following mappings: $h_1 = g_2 \circ f_1$ and $h_2 = f_2 \circ g_1$. By Lemma 1, $h_1 = g_2 \circ f_1$ and $h_2 = f_2 \circ g_1$ are identity-containment mappings.

Using Corollary 1 we can now conclude that P and $P - R$ are (identity, identity)-equivalent and thus R is a redundant subtree. □

From this lemma follows that Case A can only happen if P_2 contains redundant subtrees. Hence, if we can avoid redundancies, Case A will never occur.

The following lemma provides us with an efficient check for redundancies.

Redundancy Lemma. *Let P be a parameterized tree pattern. Then P has a redundancy if and only if it 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 .*

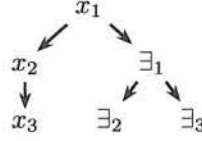


Figure 3.10: A tree pattern that contains a linear chain of existential nodes that is redundant.

Before we prove this Lemma, let us see some examples. For instance the parameterized tree patterns in Figure 3.8(a) and Figure 3.8(b) contain a linear chain of existential nodes that is redundant. In both tree patterns this linear chain is rooted in \exists_1 . Another example of such a redundancy is given in Figure 3.10. Here the linear chain is rooted in \exists_3 . Note that when we remove the linear chain rooted in \exists_3 , we have a new linear chain rooted in \exists_1 that is redundant.

Proof. 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 P that is maximal, in the sense that it is not a subtree of another redundant subtree. Then by Corollary 1, there must be a ρ -containment mapping h from (H, P) to $(\delta(H), P - T)$ with ρ and δ bijections and H the pure head for P . All distinguished variables of P must be in $P - T$, since δ is a bijection. Also all parameters of P must be in $P - T$, since ρ is also a bijection. So T consists entirely of existential nodes.

Furthermore, note that h must fix the root of P , since the height of P is at least that of $P - T$.

Any iteration h^n of h is a ρ^n -containment mapping from (H, P) to $(\delta^n(H), P - T)$ by Lemma 1. Moreover, each $h^n|_{\Delta \cup \Sigma}$ induces a permutation on the set $\Delta \cup \Sigma$ of distinguished variables and parameters. Since $\Delta \cup \Sigma$ is finite, there are only a finite number of possible permutations of $\Delta \cup \Sigma$, namely $|\Delta \cup \Sigma|!$. Hence, there will be an iteration $h^k|_{\Delta \cup \Sigma}$ and an iteration $h^{(k+l)}|_{\Delta \cup \Sigma}$ such that $h^k|_{\Delta \cup \Sigma} = h^{(k+l)}|_{\Delta \cup \Sigma}$. Thus, $h^l|_{\Delta \cup \Sigma}$ is the identity on $\Delta \cup \Sigma$, because

$$\begin{aligned} \text{id}|_{\Delta \cup \Sigma} &= (h^{-1})^k|_{\Delta \cup \Sigma} \circ h^k|_{\Delta \cup \Sigma} \\ &= (h^{-1})^k|_{\Delta \cup \Sigma} \circ h^{(k+l)}|_{\Delta \cup \Sigma} \\ &= h^l|_{\Delta \cup \Sigma} \end{aligned}$$

There are now two possible cases.

1. T itself is a linear chain. Let us then look at the parent p of T in P . Again there are two possibilities:
 - (a) $h^l(p) = p$: Since h^l is a ρ^l -containment mapping from (H, P) to $(\delta^l(H), P - T)$ and T is redundant, we know that T must be mapped to another subtree of p, T' , that is at least as deep as T . Hence, T is an eliminable path. An illustration is given in Figure 3.11(a).

(b) $h^l(p) \neq p$: Then p can only be an existential node. We now have two possibilities:

- i. T is the only subtree of p . We will show that the subtree T' , rooted in p is redundant as well. Clearly we have the following containment relations:
 - $h_1 = h^l$, a ρ^l -containment mapping from (H, P) to $(\delta^l(H), P - T')$; and
 - $h_2 = h^{-l}$, a ρ^{-l} -containment mapping from $(\delta^l(H), P - T')$ to (H, P) .

with δ and ρ as above. By Corollary 1, T' is then a redundant subtree. This is in contraction with the assumption that T is maximal. Hence, it is impossible that p has only one subtree and p is existential. An illustration of this is in given in Figure 3.11(b).

- ii. p has more than one subtree. Consider such another subtree T' . We will show that all subtrees T' of p consist entirely of existential nodes. Suppose a node $n \in T'$ is not an existential node. We then know that $h^l(n) = n$. However, since h^l is a homomorphism and p is an ancestor of n , $h^l(p)$ must be p . But this is in contradiction with the assumption that $h^l(p) \neq p$. So T' must consist entirely of existential nodes. Hence this brings us to the second case where T is not a linear chain. An illustration is given in Figure 3.11(c).

2. T is not a linear chain. An easy induction on the height of T , shows that any non-linear tree consisting entirely of existential nodes must contain an eliminable path. If the height of T is 1, there is an eliminable path of a single node: just choose one of the children of the root. If the height of T is $n > 1$, consider the subtree S of the root of T with the smallest height, at most $n - 1$. Then we have two possibilities: If S is a linear chain, we found our eliminable path. And if S is a non-linear chain we know by induction that S will contain an eliminable path. Hence, T , and thus also P , contains an eliminable path as desired.

□

As we have seen in Section 3.4, 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 (Π, Σ) is a parameterized tree patterns of which we already know it has no redundancies, and we make one additional node n existential, then it suffices to test whether n thus becomes part of a subtree C as in the Redundancy Lemma. If so, we will prune the entire search at $\Pi \cup \{n\}$.

3.5.3 Case B: Canonical Forms

We may now assume that P_1 and P_2 do not contain redundancies, for if they would, they would have been dismissed already.

Let us start by defining isomorphic parameterized tree patterns.

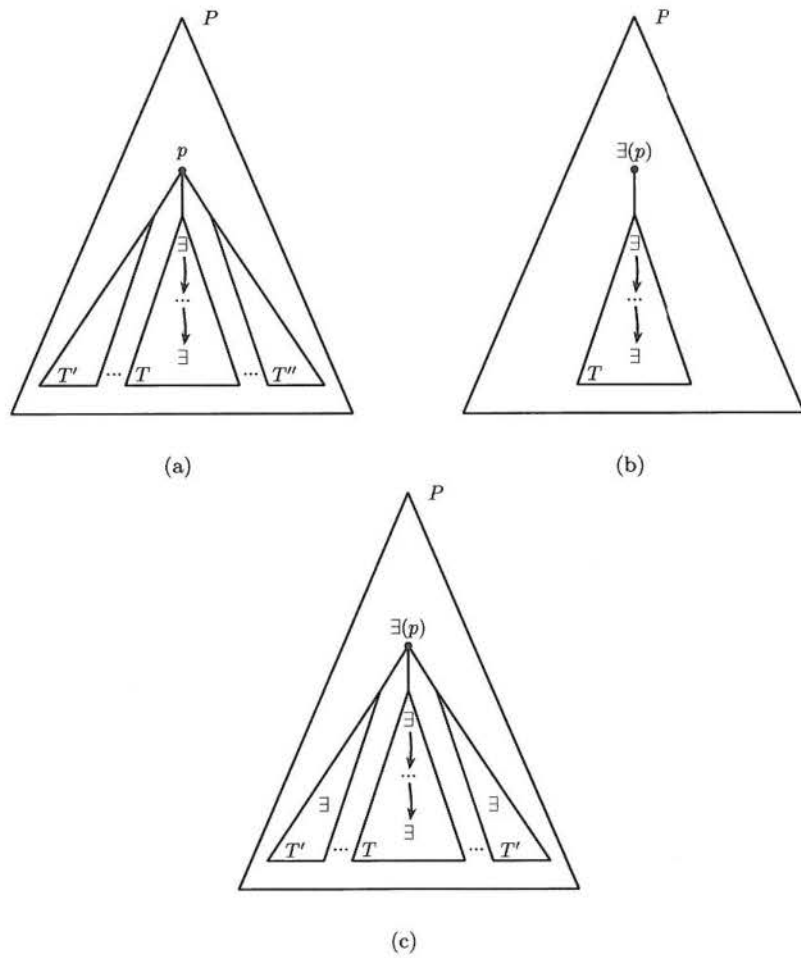


Figure 3.11: Figures to illustrate the proof of the Redundancy Lemma.

Isomorphic Parametrized Tree Patterns We call two parameterized tree patterns P_1 and P_2 *isomorphic* if there exists a homomorphism $f : P_1 \rightarrow P_2$ that is a bijection and that maps distinguished nodes to distinguished nodes, parameters to parameters and existential nodes to existential nodes. We call f an *isomorphism*. Since we are working with trees, f^{-1} is also a homomorphism.

For example, the two parameterized tree patterns in Figure 3.7 are indeed isomorphic with f as follows:

f	
σ_1	σ_1
\exists	\exists
σ_2	σ_3
x_1	x_3
x_2	x_1
σ_3	σ_2
x_3	x_2

Clearly, we have the following:

Property 1. *Any two isomorphic parameterized tree patterns P_1 and P_2 are equivalent.*

Proof. Using Corollary 1 we have to show that the following exists:

1. a bijective answer set correspondence $\delta : \Delta_1 \rightarrow \Delta_2$;
2. a bijective parameter correspondence $\rho : \Sigma_1 \rightarrow \Sigma_2$;
3. a ρ -containment mapping $f_1 : (H_1, P_1) \rightarrow (\delta(H_1), P_2)$; and
4. a ρ^{-1} -containment mapping $f_2 : (\delta(H_1), P_2) \rightarrow (H_1, P_1)$,

with H_1 the pure head for P_1 .

Since P_1 and P_2 are isomorphic, there exists a homomorphism $f : P_1 \rightarrow P_2$ that is a bijection and that maps distinguished nodes to distinguished nodes, parameters to parameters and existential nodes to existential nodes.

We now take $\delta = f|_{\Delta_1}$ and $\rho = f|_{\Sigma_1}$. Then δ and ρ are bijections since f is a bijection.

For (3) we will show that f is ρ -containment mapping from (H_1, P_1) to $(\delta(H_1), P_2)$, with H_1 the pure head for P_1 :

- f is a homomorphism;
- f maps distinguished nodes to distinguished nodes and $f|_{\Delta_1} = \delta$;
- f maps parameters to parameters and $f|_{\Sigma_1} = \rho$; and
- $f(H_1) = \delta(H_1)$.

For (4) we will show that f^{-1} is ρ^{-1} -containment mapping from $(\delta(H_1), P_2)$ to (H_1, P_1) , with H_1 the pure head for P_1 :

- f^{-1} is a homomorphism since f is a bijection and we are working with trees;
- f^{-1} maps distinguished nodes to distinguished nodes and $f^{-1}|_{\Delta_2} = \delta^{-1}$;
- f^{-1} maps parameters to parameters and $f^{-1}|_{\Sigma_2} = \rho^{-1}$; and
- $f^{-1}(\delta(H_1)) = H_1$.

□

The following lemma shows that two parameterized tree patterns without redundancies and with the same number of nodes can only be equivalent if they are isomorphic.

Isomorphism Lemma. *Consider two parameterized tree patterns P_1 and P_2 without redundancies, and with the same number of nodes. Then P_1 and P_2 are equivalent if and only if P_1 and P_2 are isomorphic.*

Proof. We only need to show the only-if direction.

Since P_1 and P_2 are equivalent we know that the following exists by Corollary 1:

1. a bijective answer set correspondence $\delta : \Delta_1 \rightarrow \Delta_2$;
2. a bijective parameter correspondence $\rho : \Sigma_1 \rightarrow \Sigma_2$;
3. a ρ -containment mapping $f_1 : (H_1, P_1) \rightarrow (\delta(H_1), P_2)$; and
4. a ρ^{-1} -containment mapping $f_2 : (\delta(H_1), P_2) \rightarrow (H_1, P_1)$,

with H_1 a pure head for P_1 .

We also know that P_1 and P_2 have the same number of existential nodes since P_1 and P_2 have the same number of nodes and ρ and δ are bijections.

Hence to prove that P_1 and P_2 are isomorphic, we only need to show that:

1. f_1 maps existential nodes to existential nodes; and
2. $f_1|_{\Pi_1}$ is a bijection.

There to, it suffices to prove that f_1 is surjective on the existential nodes of P_2 , because f_1 is already a bijection from $\Delta_1 \cup \Sigma_1$ to $\Delta_2 \cup \Sigma_2$.

Assume that $f_1|_{\Pi_1}$ is not surjective. Hence, there will be some existential nodes $p \in \Pi_2$ that are not in the range of f_1 . Note that these existential nodes p can never have descendants that are parameters or distinguished nodes since f_1 is a homomorphism and δ and ρ bijections. Now fix some p as high as possible in the tree. Then the entire subtree R rooted in p consists entirely of existential nodes, that are not in the range of f_1 . We will now show that this subtree R is a redundant subtree in P_2 . We then have a contradiction since we assume that P_2 is redundancy free.

Since the containment mappings f_1 and f_2 exist, we know that in particular the following containment mappings will exist:

1. $g_1 = f_1$, a ρ -containment mapping from (H_1, P_1) to $(\delta(H_1), P_2 - R)$; and
2. $g_2 = f_2|_{P_2 - R}$, a ρ^{-1} -containment mapping from $(\delta(H_1), P_2 - R)$ to (H_1, P_1) ,

with δ and ρ as above.

Let us now look at the following mappings $h_1 = f_1 \circ g_2$ and $h_2 = g_1 \circ f_2$. By Lemma 1, h_1 and h_2 are identity-containment mappings. Using Corollary 1, we can now conclude that P_2 and $P_2 - R$ are equivalent, and thus R is a redundant subtree. \square

From the above lemma it follows that Case B can only happen if P_1 and P_2 are actually isomorphic. In particular, P_1 and P_2 have the same underlying tree.

So, in our algorithm, we need an efficient way to avoid isomorphic parameterized tree patterns based on the same tree T .

Fortunately, there is a standard way to do this, by working with *canonical forms* of parameterized tree patterns. Consider a pair (Π, Σ) , as in Section 3.4. 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 ' σ '; and all distinguished nodes get ' x '. We then observe that the patterns (Π_1, Σ_1) and (Π_2, Σ_2) are isomorphic iff there is a tree isomorphism between the corresponding labeled versions of T that respects the labels.

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 T , which we already have (Section 3.3), to take into account the node labels. Furthermore, the classical linear-time algorithm to canonize a tree [3] generalizes straightforwardly to labeled trees. A nice review of these generalizations has been given by Chi, Yang and Muntz [9].

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.

Example. We can refine the level sequence introduced in Section 3.3 to a *refined level sequence* for parameterized tree patterns P as follows: if the tree pattern P consists of n nodes, then the *refined level sequence* is now a sequence of n elements, where the i th element is the depth of the i th node in preorder in the pattern, followed by a 'd' if the node is distinguished; followed by a 'e' if the node is existential and followed by a 'p' if the node is a parameter. The canonical ordering of a parameterized tree pattern P , is then the ordering of P that yields the lexicographically maximal refined level sequence, among all orderings of P . Then the refined level sequences for the parameterized tree patterns in Figure 3.7 are:

(a) 0p1e2p2d1d2p2d

(b) 0p1d2d2p1e2d2p

and (a) is the canonical one.

Armed by the canonical form, we are now in a position to describe how the algorithm of Section 3.4 must be modified to avoid equivalent parameterized tree patterns. First of all, we only work with patterns (Π, Σ) in canonical form; the others are dismissed. However, the problem then arises that a parent pattern (Π', Σ') , where we omit a variable from either Π or Σ as described in Section 3.4, 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

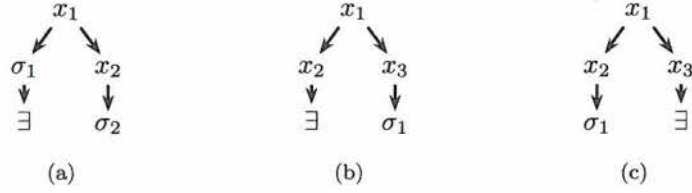


Figure 3.12: The tree pattern in (b) is a parent of a tree pattern in (a), and (c) is the canonical version of (b).

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.

This does not completely solve the problem, however. Indeed the frequency table of (Π'', Σ'') might not yet have been computed. For example, consider the parameterized tree pattern in Figure 3.12(a), and one of its parents in Figure 3.12(b). The canonical version of this parent, using the canonical ordering from the previous example, is shown in Figure 3.12(c). Using the current order for computing the frequency tables as in Algorithm 1, the frequency table for the pattern in Figure 3.12(c) is not yet computed, when we want to compute the frequency for the pattern in Figure 3.12(a).

We can solve this by changing the order in which we compute the frequency tables. We work with increasing levels: in level i we compute the frequency tables for all pairs (Π, Σ) , where $\#\Pi + \#\Sigma = i$. If we use this order, we are sure that when we compute the frequency table of a pair (Π, Σ) , all frequency tables of pairs (Π', Σ') with $(\#\Pi' + \#\Sigma') < (\#\Pi + \#\Sigma)$, have been computed.

3.5.4 The Algorithm

The final algorithm is now given in Algorithm 2. The outline for canonizing a parameterized tree pattern is given in Function 3.

3.5.5 Example Run

In this Section we give an example run of the final algorithm in Algorithm 2. We use the same data graph G ; tree T ; and minimum support threshold, 3, as in the example run in Section 3.4.4.

Note that there are two important differences between this run and the run in Section 3.4.4:

1. duplicate work is avoided: equivalent tree patterns are not generated; and
2. the order for computing the tree patterns is different in the sense that here, the tree patterns are generated in levels, as explained in Section 3.5.3.

The example run is then given in Table 3.2.

Algorithm 2 Levelwise search for frequent tree patterns with equivalence checking.

```

1: for each unordered, rooted tree  $T$  do
2:   level := number of nodes of  $T$ 
3:    $i := 0$ 
4:    $C_0 := \{(\emptyset, \emptyset)\}$ ;  $F := \emptyset$ 
5:   while  $i \leq \text{level}$  AND  $C_i \neq \emptyset$  do
6:     {Candidate evaluation}
7:     for each pattern  $(\Pi, \Sigma)$  in  $C_i$  do
8:       if  $\Sigma = \emptyset$  then
9:         Compute  $\text{FreqTab}_{\Pi, \emptyset}$  in SQL
10:      else
11:        if  $(\#\Sigma = 1 \text{ AND } \#\Pi = 0)$  then
12:           $\text{CanTab}_{\Pi, \Sigma} := \text{set of nodes of } G$ 
13:        else
14:           $\text{CanTab}_{\Pi, \Sigma} := \bowtie \{f^{-1}(\text{FreqTab}_{\Pi'', \Sigma''}) \mid (\Pi', \Sigma') \text{ parent of } (\Pi, \Sigma)$ 
15:                                 $\text{and } (f, (\Pi'', \Sigma'')) = \text{Canonize}(\Pi', \Sigma')\}$ 
16:        end if
17:      end if
18:      Compute  $\text{FreqTab}_{\Pi, \Sigma}$  in SQL
19:      if  $(\text{FreqTab}_{\Pi, \Sigma} \neq \emptyset)$  then
20:         $F = F \cup \{(\Pi, \Sigma)\}$ 
21:      end if
22:    end for
23:    {Candidate generation}
24:     $C_{i+1} = \{(\Pi, \Sigma) \mid \text{all parents } (\Pi', \Sigma') \text{ of } (\Pi, \Sigma) \text{ are in } F\}$ 
25:    {Equivalence Check}
26:    for each pattern  $(\Pi, \Sigma)$  in  $C_{i+1}$  do
27:      if  $((\Pi, \Sigma)$  contains a redundancy) then
28:        remove  $(\Pi, \Sigma)$  from  $C_{i+1}$ 
29:      else if  $((\Pi, \Sigma)$  is not canonical) then
30:        remove  $(\Pi, \Sigma)$  from  $C_{i+1}$ 
31:      end if
32:    end for
33:     $i := i + 1$ 
34:  end while
35: end for

```

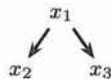
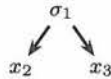
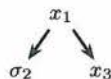
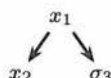
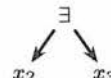
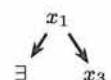
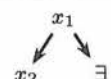
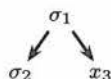
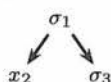
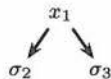

Function 3 Canonize (Π', Σ') based on T

```



1:  $(\Pi_c, \Sigma_c) := \text{canonization of } (\Pi', \Sigma') \text{ based on } T$ 
2:  $f := \text{isomorphism from } (\Pi_c, \Sigma_c) \text{ to } (\Pi', \Sigma')$ 
3: return  $(f, (\Pi_c, \Sigma_c))$ 

```

Table 3.2: Example run of the inner loop with equivalence checking for the tree and data graph in Figure 3.6

Π	Σ	P	$CanTab$	$FreqTab$												
Level 0																
\emptyset	\emptyset			<table><tr><th>$Freq$</th></tr><tr><td>15</td></tr></table>	$Freq$	15										
$Freq$																
15																
Level 1																
\emptyset	$\{x_1\}$		nodes of G	<table><tr><th>σ_1</th><th>$Freq$</th></tr><tr><td>0</td><td>9</td></tr><tr><td>2</td><td>4</td></tr></table>	σ_1	$Freq$	0	9	2	4						
σ_1	$Freq$															
0	9															
2	4															
\emptyset	$\{x_2\}$		nodes of G	<table><tr><th>σ_2</th><th>$Freq$</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr><tr><td>4</td><td>3</td></tr></table>	σ_2	$Freq$	1	3	2	3	3	3	4	3		
σ_2	$Freq$															
1	3															
2	3															
3	3															
4	3															
\emptyset	$\{x_3\}$		Equivalent with $(\emptyset, \{x_2\})$													
$\{x_1\}$	\emptyset			<table><tr><th>$Freq$</th></tr><tr><td>14</td></tr></table>	$Freq$	14										
$Freq$																
14																
$\{x_2\}$	\emptyset		Redundancy													
$\{x_3\}$	\emptyset		Equivalent with $(\{x_2\}, \emptyset)$													
Level 2																
\emptyset	$\{x_1, x_2\}$		$FreqTab_{\emptyset, \{x_1\}}$ $\bowtie FreqTab_{\emptyset, \{x_2\}}$	<table><tr><th>σ_1</th><th>σ_2</th><th>$Freq$</th></tr><tr><td>0</td><td>1</td><td>3</td></tr><tr><td>0</td><td>2</td><td>3</td></tr><tr><td>0</td><td>3</td><td>3</td></tr></table>	σ_1	σ_2	$Freq$	0	1	3	0	2	3	0	3	3
σ_1	σ_2	$Freq$														
0	1	3														
0	2	3														
0	3	3														
\emptyset	$\{x_1, x_3\}$		Equivalent with $(\emptyset, \{x_1, x_2\})$													
\emptyset	$\{x_2, x_3\}$		$FreqTab_{\emptyset, \{x_2\}}$ $\bowtie FreqTab_{\emptyset, \{x_3\}}$	\emptyset												
$\{x_1\}$	$\{x_2\}$		$FreqTab_{\emptyset, \{x_2\}}$ $\bowtie FreqTab_{\{x_1\}, \emptyset}$	<table><tr><th>σ_2</th><th>$Freq$</th></tr><tr><td>1</td><td>3</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>3</td></tr></table>	σ_2	$Freq$	1	3	2	3	3	3				
σ_2	$Freq$															
1	3															
2	3															
3	3															

$\{x_1\}$	$\{x_3\}$	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ x_2 \quad \sigma_3 \end{array}$	Equivalent with $(\{x_1\}, \{x_2\})$
$\{x_2\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \exists \quad x_3 \end{array}$	Redundancy
$\{x_2\}$	$\{x_3\}$	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \exists \quad \sigma_3 \end{array}$	Redundancy
$\{x_3\}$	$\{x_1\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ x_2 \quad \exists \end{array}$	Equivalent with $(\{x_2\}, \{x_1\})$
$\{x_3\}$	$\{x_2\}$	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad \exists \end{array}$	Equivalent with $(\{x_2\}, \{x_3\})$
$\{x_1, x_2\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \exists \quad x_3 \end{array}$	Redundancy
$\{x_1, x_3\}$	\emptyset	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ x_2 \quad \exists \end{array}$	Equivalent with $(\{x_1, x_2\}, \emptyset)$
$\{x_2, x_3\}$	\emptyset	$\begin{array}{c} x_1 \\ \swarrow \quad \searrow \\ \exists \quad \exists \end{array}$	Redundancy
Level 3			
\emptyset	$\{x_1, x_2, x_3\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad \sigma_3 \end{array}$	Pruned
$\{x_1\}$	$\{x_2, x_3\}$	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \sigma_2 \quad \sigma_3 \end{array}$	Pruned
$\{x_2\}$	$\{x_1, x_3\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \exists \quad \sigma_3 \end{array}$	Pruned
$\{x_3\}$	$\{x_1, x_2\}$	$\begin{array}{c} \sigma_1 \\ \swarrow \quad \searrow \\ \sigma_2 \quad \exists \end{array}$	Equivalent with $(\{x_2\}, \{x_1, x_3\})$
$\{x_1, x_2\}$	$\{x_3\}$	$\begin{array}{c} \exists \\ \swarrow \quad \searrow \\ \exists \quad \sigma_3 \end{array}$	Pruned

$\{x_1, x_3\}$	$\{x_2\}$		Equivalent with $(\{x_1, x_2\}, \{x_3\})$
$\{x_2, x_3\}$	$\{x_1\}$		Redundancy

3.6 Result Management: Pattern Database

When the algorithm is terminated, its final output consists of a set of frequency tables for each tree T that was investigated. All frequency tables are kept in a relational database that we call the *pattern database*. The pattern database is an ideal platform for an interactive tool for browsing the frequent queries. We developed such a browser called *Certhia* and discuss it in Section 5.1.

The pattern database is also an ideal platform for tree-query-association mining as will be described in Chapter 4.

4

Mining Tree-Query Associations

In this Chapter we present an algorithm for mining confident tree-query associations in a large data graph.

Recall from Section 2.3 that a parameterized association rule (pAR) is something of the form $Q_1 \Rightarrow_\rho Q_2$, with Q_1 and Q_2 parameterized tree queries, $\rho : \Sigma_1 \rightarrow \Sigma_2$ a parameter correspondence, and $Q_2 \subseteq_\rho Q_1$. An instantiated association rule (iAR) is a pair $(Q_1 \Rightarrow_\rho Q_2, \alpha)$, with $Q_1 \Rightarrow_\rho Q_2$ a pAR and $\alpha : \Sigma_2 \rightarrow U$ a parameter assignment for $Q_1 \Rightarrow_\rho Q_2$. Also recall that the confidence of an iAR in a data graph G is defined as $\text{Freq}(Q_2^{\alpha_2}) / \text{Freq}(Q_1^{\alpha_2 \circ \rho})$.

The algorithm presented in this Chapter finds all iARs of the form $(Q_{\text{left}} \Rightarrow_\rho Q_{\text{right}}, \alpha)$ that are confident and frequent in a given data graph G for a given lhs Q_{left} .

We start by showing that we do not need to tackle the problem in its full generality in Section 4.1. Next, we present our algorithm in Section 4.2, and discuss it in more detail in Section 4.3 and Section 4.4. We give an example run of our algorithm in Section 4.5. And finally, we discuss equivalent association rules in Section 4.6, and show how we can avoid generating them.

4.1 Problem Reduction

In this Section we show that, without loss of generality, we can focus on the case where the given lhs tree query Q_{left} is pure in the sense that was defined in Section 3.1. We will also show that this restriction can not be imposed on the rhs tree queries to be output. We also make a remark regarding “free constants” in the head of a tree query.

Pure lhs's Assume that all possible variables (nodes of tree patterns) have been arranged in some fixed but arbitrary order. Recall then from Section 3.1 that we call a parameterized tree query $Q = (H, P)$ pure when H consists of the enumeration in order and without repetitions of all distinguished variables of P . In particular H can not contain parameters. We call H the pure head for P . As an illustration, the lhs of rule (a) of Figure 4.1 is impure, while the lhs of rule (b) is pure.

Consider the pARs in Figure 4.1(a) and Figure 4.1(b), and their instantiations in Figure 4.1(c) and Figure 4.1(d). The rules in Figure 4.1(a) and Figure 4.1(c) have an impure lhs. If we apply the iARs in Figure 4.1(c) and Figure 4.1(d) to the data graph G in Figure 2.2(a), both have the same frequency, namely 2, and the same confidence, namely 33%. Indeed, since the frequency of a tree query is in fact the frequency of its body, repetitions of distinguished variables in the head and the occurrence of parameters in the head do not change the frequency of a tree query. In fact the pAR in Figure 4.1(b) is the purification of the pAR in Figure 4.1(a): the repetition of the distinguished variable x_2 is removed from the head, and the parameter σ_1 is removed from the head.

Hence, a pAR with an impure lhs can always be rewritten to an equivalent pAR with a pure lhs, in such a way that all instantiations of the pAR with the impure lhs correspond to instantiations of the pAR with pure lhs, with the same confidence and frequency. Indeed, take a legal pAR $Q_1 \Rightarrow_\rho Q_2$ with Q_1 not pure. We know that Q_1 's head is mapped to Q_2 's head by some ρ -containment mapping. Hence, we can purify Q_1 by removing all parameters and repetitions of distinguished variables from Q_1 's head, sort the head by the order on the variables, and perform the corresponding actions on Q_2 's head as prescribed by the ρ -containment mapping.

We can conclude that it is sufficient to only consider pARs with pure lhs's. The rhs, however, need not be pure; impure rhs's are in fact interesting, as we will demonstrate next.

Impure rhs's Consider the pAR in Figure 4.2(a). The rhs is impure since x_2 appears twice in the head. The pAR expresses that a sufficient proportion of the matchings of the lhs pattern, are also matchings of the rhs pattern, which is the same as the lhs pattern except that x_2 is equal to x_3 . Since the pAR has no parameters, we can identify it with its instantiation by the empty parameter assignment. The confidence is then:

$$\frac{m}{\sum_x \deg^2 x}$$

where m is the number of edges, x ranges over the nodes in the graph, and $\deg x$ is the outdegree of (number of edges leaving) x . Since $m = \sum_x \deg x$, we show by an

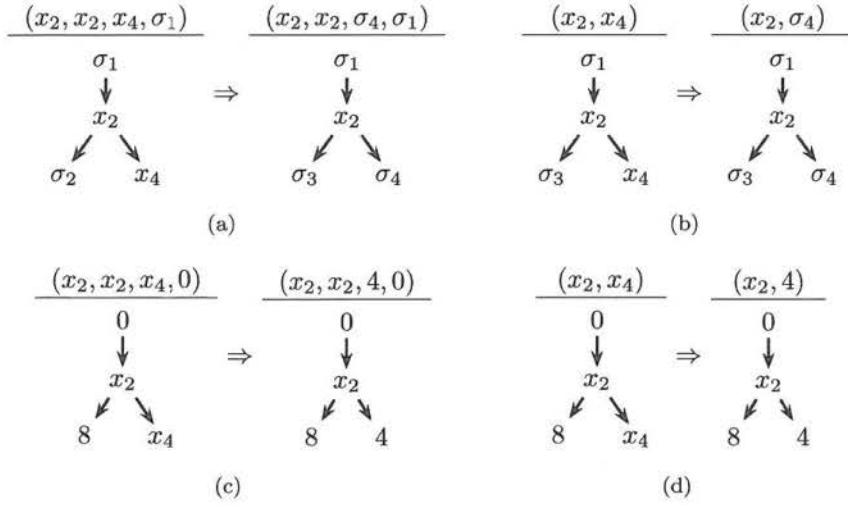


Figure 4.1: Rule (a) has a non-pure lhs. Rule (b) is the purification of rule (a), and expresses precisely the same information. Rules (c) and (d) are two example instantiations.

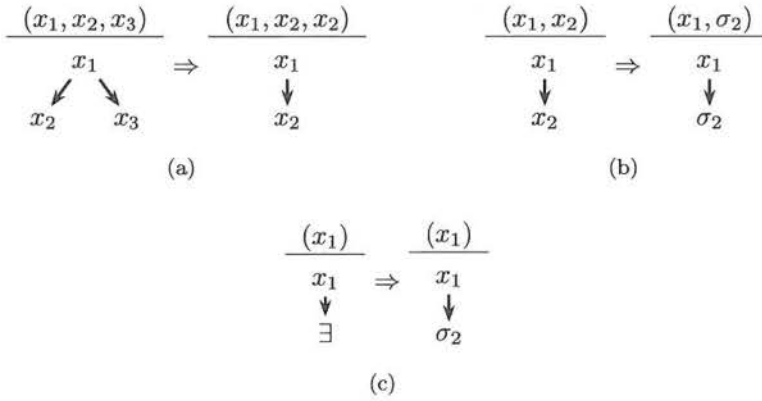


Figure 4.2: (a) and (b) are pARs with impure rhs. (c) is an ill-advised attempt to purify (b) on the rhs.

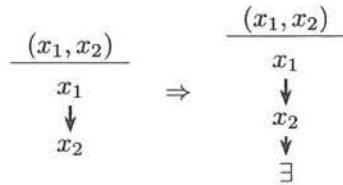


Figure 4.3: A pAR with a pure rhs.

easy calculation that this confidence is much larger than $1/m$:

$$\begin{aligned}
 \frac{m}{\sum_x \deg^2 x} &= \frac{\sum_x \deg x}{\sum_x \deg^2 x} \\
 &\geq \frac{\sum_x \deg x}{(\sum_x \deg x)^2} \\
 &= \frac{1}{\sum_x \deg x} \\
 &= \frac{1}{m}
 \end{aligned}$$

Hence, the sparser the graph (with the number of nodes remaining the same), the higher the confidence, and thus the pAR is interesting in that it tells us something about the sparsity of the graph. As an illustration, on the graph of Figure 2.2(a) the confidence is 0.4, but on the graph of Figure 2.2(b), it is 0.6.

Also consider the pAR in Figure 4.2(b). Again the rhs is impure since its head contains a parameter. Create an iAR for this pAR with $\alpha = \sigma_2 \mapsto 8$. With confidence c , this iAR then expresses that a fraction of c of all edges point to node 8, which again would be an interesting property of the graph.

The knowledge expressed by the above two example pARs cannot be expressed using pARs with pure rhs's. To illustrate, the pAR of Figure 4.2(c) may at first seem equivalent (and has a pure rhs) to that of Figure 4.2(b). On second thought, however, it says nothing about the proportion of *edges* pointing to σ_2 , but only about the proportion of *nodes* with an edge to σ_2 .

Of course, we are not implying that pARs with pure rhs's are uninteresting. But all they can express are statements about the proportion of matchings of the lhs that can be specialized or extended to a matching of the rhs (another example is in Figure 4.3, which says something about the proportion of edges that can be extended); they cannot say anything about the proportion of matchings of the lhs that satisfy certain equalities in the distinguished variables.

Free Constants Most treatments of conjunctive database queries [8, 44, 1] allow arbitrary constants in the head. In our treatment, a constant can only appear in the head as the value of a parameter. Fortunately this is enough. We do not need to consider “free” constants, i.e., constants not corresponding to a parameter value. To see this, first consider the possibility of free constants in the lhs. The same argument we already gave to assume that the lhs is pure can be used to dismiss this possibility. Next consider a free constant in the rhs of an iAR $(Q_1 \Rightarrow_\rho Q_2, \alpha)$, with $Q_1 = (H_1, P_1)$ and $Q_2 = (H_2, P_2)$ and Q_1 already pure. Then there must be a ρ -containment mapping $f : Q_1 \rightarrow Q_2$, with $f(H_1) = H_2$, for the iAR to be legal. Hence, a constant a can only appear in H_2 by one of the following two possibilities:

1. $a = \alpha(f(\sigma)) = \alpha(\rho(\sigma))$, with $\sigma \in H_1$; or
2. $a = \alpha(f(x))$, with x a distinguished variable in H_1 .

However, in both cases a is not actually free, being equal to a parameter value.

4.2 Overall Approach

Given the inputs G , $Q_{\text{left}} = (H_{\text{left}}, P_{\text{left}})$, minconf , and minsup , an outline of our algorithm for the association rule mining problem is that of four nested loops:

1. Generate, incrementally, all possible trees of increasing sizes. Avoid trees that are isomorphic to previously generated ones. The height of the generated trees must be at least the height of the tree underlying P_{left} . (When enough trees have been generated, this loop can be terminated.)
2. For each new generated tree T , generate all frequent instantiated tree patterns P^α based on that tree.

These first two loops are nothing but our algorithm for mining frequent tree queries as presented in Chapter 3.

3. For each parameterized tree pattern P , generate all containment mappings f from P_{left} to P . Here, a plain “containment mapping” is a ρ -containment mapping, as defined in Section 2.2.1, for some ρ . Note that ρ then equals $f|_{\Sigma_{\text{left}}}$.
4. For each f , generate the parameterized tree query $Q_{\text{right}} = (f(H_{\text{left}}), P)$, and all parameter assignments α such that $(Q_{\text{left}} \Rightarrow_\rho Q_{\text{right}}, \alpha)$ is frequent, and the confidence exceeds minconf . The generation of all these α 's happens in a parallel fashion.

This approach is complete, i.e., it will output everything that must be output. In proof, consider a legal, frequent and confident iAR $(Q_{\text{left}} \Rightarrow_{\rho_0} Q_{\text{right}}, \alpha_0)$, with $Q_{\text{right}} = (H_{\text{right}}, P_{\text{right}})$. The tree T is the underlying tree of P_{right} ; P_{right} is a tree pattern P in loop 2; the containment mapping f in loop 3 is the ρ_0 -containment mapping that exists since the iAR is legal; H_{right} is $f(H_{\text{left}})$; and α in loop 4 is α_0 .

The reader may wonder whether loop 3 cannot be organized in a levelwise fashion. This is not obvious, however, since any two queries of the form $((f_1(H_{\text{left}}), P), \alpha)$ and $((f_2(H_{\text{left}}), P), \alpha)$ have exactly the same frequency, namely that of P^α . Loop 4, however, is levelwise because it is based on loop 2 which is levelwise.

As already mentioned, these first two loops are nothing but our algorithm for mining frequent tree queries as presented in Chapter 3. As already explained in Section 3.6, in loop 2 we build up a structured database containing all frequency tables for all trees in loop 1. We call this database the *pattern database*. In fact these two loops should be regarded as a preprocessing step; once built up, this pattern database can be used to generate association rules.

Hence, in practice an outline for our rule-mining algorithm is the following:

1. Preprocessing step: Generate a pattern database D using the algorithm discussed in Chapter 3. Halt this algorithm when enough patterns are generated.
2. Consider, in a levelwise order, each parameterized tree pattern P that has frequent instantiations in D , and such that the height of the underlying tree of P is at least the height of the underlying tree of P_{left} .

3. For each parameterized tree pattern P , generate all containment mappings f from P_{left} to P and let ρ be $f|_{\Sigma_{\text{left}}}$.
4. For each f , generate the parameterized tree query $Q = (f(H_{\text{left}}), P)$, and all parameter assignments α such that $(Q_{\text{left}} \Rightarrow_{\rho} Q, \alpha)$ is frequent; and the confidence exceeds *minconf*. The generation of all these α 's happens in a parallel fashion.

We present loops 3 and 4 in detail in Sections 4.3 and 4.4. In Section 4.6, we will show how our overall approach must be refined so that the generation of equivalent association rules is avoided.

4.3 Generation of Containment Mappings

In this Section, we discuss loop 3, the generation of all containment mappings f from P_{left} to P . So, we need to solve the following problem: Given two parameterized tree patterns P_1 and P_2 , find all containment mappings f from P_1 to P_2 .

Since the patterns are typically small, a naive algorithm suffices. For a node x_1 of P_1 and a node x_2 of P_2 , we say that x_1 “matches” x_2 if there is a containment mapping f from the subpattern of P_1 rooted at x_1 to the subpattern of P_2 rooted at x_2 such that $f(x_1) = x_2$. In a first phase, we determine for every node y of P_2 separately whether the root r_1 of P_1 matches y . While doing so, we also determine for every other node x_1 of P_1 , and every node x_2 below y at the same distance as x_1 is from r_1 , whether x_1 matches x_2 . We store all these boolean values in a two-dimensional matrix *Map*. The function for filling in *Map* is given in Function 4. In line 2 of this function we mean by “ $x_1 \mapsto x_2$ is legal”, that if x_1 is a distinguished variable, then x_2 is a distinguished variable or a parameter; and if x_1 is a parameter then x_2 is a parameter, as prescribed by the definition of a ρ -containment mapping in Section 2.2.1.

This first phase compares every possible pair (x_1, x_2) , with x_1 a node in P_1 and x_2 a node in P_2 , at most once. Indeed, if x_1 is at distance d from r_1 , then x_1 will be compared to x_2 only during the matching of r_1 with the node y that is d steps above x_2 in P_2 (if existing). We thus have an $O(n_1 \times n_2)$ algorithm, where n_1 (n_2) is the number of nodes in P_1 (P_2).

In a second phase, we output all containment mappings. Initially, by a synchronous preorder traversal of P_1 and P_2 , we map each node of P_1 to the first matching node of P_2 . We store this first mapping in a one-dimensional matrix *Cm*. In Function 5 an outline for finding the initial containment mapping is given.

In each subsequent step, we look for the last node x_1 (in preorder) of P_1 , currently matched to some node x_2 , with the property that x_1 can also be matched to a right sibling x_3 of x_2 , and now map x_1 to the first such x_3 . The mappings of all nodes of P_1 coming after x_1 are reinitialized. Every such step takes time that is linear in n_1 and n_2 . Of course, the total number of different containment mappings may well be exponential in n_1 . An outline of this step is given in Function 6.

The complete outline for the generation of all containment mappings is given in Function 7.

Function 4 Function for filling in Map

```

1: bool FillInn( $x_1 \in P_1, x_2 \in P_2$ )
2: if  $x_1 \mapsto x_2$  is legal then
3:   Match := true;
4:   for each child  $c_1$  of  $x_1$  from left to right do
5:     MatchChild := false;
6:     for each child  $c_2$  of  $x_2$  from left to right do
7:       MatchChild := MatchChild OR FillIn( $c_1, c_2$ )
8:     end for
9:     Match := Match AND MatchChild;
10:  end for
11:  Map[ $x_1, x_2$ ] := Match;
12:  return Match;
13: else
14:   Map[ $x_1, x_2$ ] := false;
15:   return false
16: end if

```

Function 5 Function for finding the initial containment mapping

```

1: Init( $x_1 \in P_1, x_2 \in P_2$ )
2: Cm[ $x_1$ ] :=  $x_2$ ;
3: for each child  $c_1$  of  $x_1$  from left to right do
4:   for each child  $c_2$  of  $x_2$  from left to right do
5:     if Map[ $c_1, c_2$ ] then
6:       Init( $c_1, c_2$ );
7:       Break;
8:     end if
9:   end for
10: end for

```

Function 6 Function for finding the other containment mappings

```

1: bool Step( $x \in P_1$ )
2: Found := false;
3: for each child  $c$  from  $x$  from right to left do
4:   if Step( $c$ ) then
5:     Found := true;
6:     Break;
7:   end if
8: end for
9: if Found then
10:  for each right-sibling  $z$  of  $c$  from left to right do
11:     $p_2 := \text{Cm}[x]$ ;
12:    for each child  $c_2$  of  $p_2$  from left to right do
13:      if Map[ $z, c_2$ ] then
14:        Init( $z, c_2$ )
15:      end if
16:    end for
17:  end for
18:  return true;
19: else
20:  if  $x$  is the root of  $P_1$  then
21:    return false;
22:  else
23:     $p_2 := \text{Cm}[x]$ ;
24:    for each right-sibling  $s$  of  $p_2$  from left to right do
25:      if Map[ $x, s$ ] then
26:        Init( $x, s$ )
27:        Break;
28:      end if
29:    end for
30:    return true;
31:  end if
32: end if

```

Function 7 Function for generating all containment mappings from P_1 to P_2

```

1: GenerateCm( $P_1, P_2$ )
2: Initialize Map;
3:  $r_1 := \text{root of } P_1$ ;
4: for each  $x_2 \in P_2$  in preorder do
5:   FillIn( $r_1, x_2$ );
6: end for
7: for each node  $x_2 \in P_2$  in preorder do
8:   if Map[ $r_1, x_2$ ] then
9:     Initialize Cm;
10:    Init( $r_1, x_2$ )
11:    repeat
12:      Output Cm;
13:    until not Step( $r_1$ )
14:   end if
15: end for

```

We can thus easily generate all containment mappings f from P_{left} to P as required for loop 3 of our overall algorithm. Note, however, that in loop 4 these mappings are used to produce the head $f(H_{\text{left}})$ of query Q_{right} . For Q_{right} to be a legal query, this head must contain all distinguished variables of P . Hence, we only pass to loop 4 those f whose image contains all distinguished variables of P .

4.4 Generation of Parameter Assignments

In loop 4, our task is the following. Given a containment mapping $f : P_{\text{left}} \rightarrow P$, let $\rho = f|_{\Sigma_{\text{left}}}$, and generate all parameter assignments α such that $(Q_{\text{left}} \Rightarrow_{\rho} (f(H_{\text{left}}), P), \alpha)$ is frequent and confident in G . We show how this can be done in a parallel database-oriented fashion.

Recall from Section 3.6 that the frequency tables for P_{left} and P are available in a relational database. Our crucial observation is that we can compute precisely the required set of parameter assignments α , together with the frequency and confidence of the corresponding association rules, by a single relational algebra expression. This expression has the following form:

$$\pi_{plist} \sigma_{\frac{FreqTab_P.freq}{FreqTab_{P_{\text{left}}}.freq} \geq minconf} (FreqTab_{P_{\text{left}}} \bowtie_{\theta} FreqTab_P)$$

Here, π denotes projection, σ denotes selection, and \bowtie denotes join. The join condition θ and the projection list $plist$ are defined as follows. For θ , we take the conjunction:

$$\bigwedge_{\sigma \in \Sigma_{\text{left}}} FreqTab_{P_{\text{left}}}.\sigma = FreqTab_P.\rho(\sigma)$$

Furthermore, $plist$ consists of all attributes $P_{\text{left}}.\sigma_{\text{left}}$, with $\sigma_{\text{left}} \in \Sigma_{\text{left}}$; all attributes $P.\sigma$, with $\sigma \in \Sigma$; together with the attributes $FreqTab_P.freq$ and $FreqTab_{P_{\text{left}}}.freq$.

Referring back to our overall algorithm (Section 4.2), we thus generate, for each pattern P from loop 2 and each containment mapping f in loop 3, all association rules with the given Q_{left} as lhs in parallel, by one relational database query (which can be implemented by a simple SQL select-statement).

Example. Consider Q_{left} and $P = (\Pi, \Sigma)$ as shown in Figures 4.4(a) and 4.4(b). We have $\Sigma_{\text{left}} = \{x_1, x_4\}$ and $\Pi_{\text{left}} = \{x_3, x_6\}$, and $\Sigma = \{x_1, x_4, x_5\}$ and $\Pi = \{x_3\}$. Take the following containment mapping f from P_{left} to P :

f	
σ_1	σ_1
x_2	x_2
\exists_3	\exists
σ_4	σ_4
x_5	x_2
\exists_6	\exists
x_7	σ_4

Then the rhs query Q_{right} equals $((x_2, x_2, \sigma_4), P)$, and the relational algebra expression for computing all parameter assignments and their corresponding frequencies and confidences looks as follows:

$$\pi_{plist} \sigma_{\frac{FreqTab_P \cdot freq}{FreqTab_{P_{\text{left}}} \cdot freq} \geq minconf} (FreqTab_{P_{\text{left}}} \bowtie_{\theta} FreqTab_P)$$

with $plist$ equal to

$$FreqTab_{P_{\text{left}}} \cdot \sigma_1, FreqTab_{P_{\text{left}}} \cdot \sigma_4, FreqTab_P \cdot \sigma_1, FreqTab_P \cdot \sigma_4, FreqTab_P \cdot \sigma_5, \\ FreqTab_P \cdot freq, FreqTab_P \cdot freq / FreqTab_{P_{\text{left}}} \cdot freq$$

and θ equal to

$$FreqTab_P \cdot \sigma_1 = FreqTab_{P_{\text{left}}} \cdot \sigma_1 \wedge FreqTab_P \cdot \sigma_4 = FreqTab_{P_{\text{left}}} \cdot \sigma_4$$

In SQL, we get:

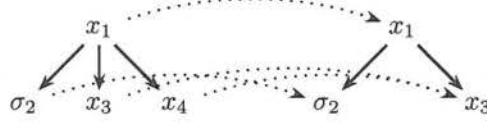
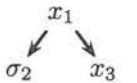

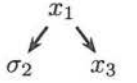
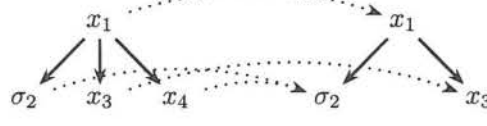
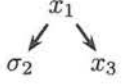
```
SELECT freqQleft.x1, freqQleft.x4, freqP.x1, freqP.x4,
       freqP.x5, freqP.freq, freqP.freq/freqQleft.freq
FROM freqP, freqQleft
WHERE freqQleft.x1= freqP.x1 AND freqQleft.x4=freqP.x4
      AND freqP.freq/freqQleft.freq >= minconf
```

4.5 Example Run

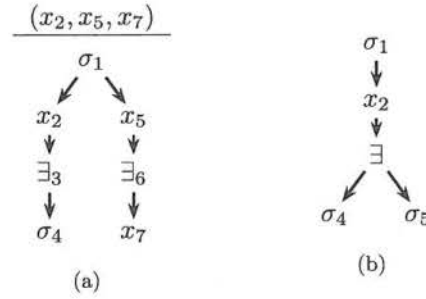
In this Section we give an example run of the algorithm discussed in Section 4. We use the same data graph G , tree T , and minimum support threshold, 3, as in the example run in Section 3.4.4 and Section 3.5.5. The fixed lhs tree query is given in Figure 4.5(a), its corresponding frequency table in Figure 4.5(b), and the minimum confidence threshold is 30%. All frequent tree patterns based on T were already generated in the example run of Section 3.5.5.

The example run is then given in Table 4.1.

Table 4.1: Example run of the association-rule-mining algorithm

P	Containment Mapping	Q_{right}	$ConfTab$																				
Level 0																							
(\emptyset, \emptyset)	No Containment Mappings																						
Level 1																							
$(\emptyset, \{x_1\})$	No Containment Mappings																						
$(\emptyset, \{x_2\})$		(x_1, x_3, x_3) 	<table><tr><th>$P_{\text{left}}.\sigma_2$</th><th>$P.\sigma_2$</th><th>$Freq$</th><th>$Conf$</th></tr><tr><td>1</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>2</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>3</td><td>3</td><td>3</td><td>33%</td></tr><tr><td>4</td><td>4</td><td>3</td><td>60%</td></tr></table>	$P_{\text{left}}.\sigma_2$	$P.\sigma_2$	$Freq$	$Conf$	1	1	3	33%	2	2	3	33%	3	3	3	33%	4	4	3	60%
$P_{\text{left}}.\sigma_2$	$P.\sigma_2$	$Freq$	$Conf$																				
1	1	3	33%																				
2	2	3	33%																				
3	3	3	33%																				
4	4	3	60%																				
$(\emptyset, \{x_2\})$		(x_1, σ_2, x_3) 	<table><tr><th>$P_{\text{left}}.\sigma_2$</th><th>$P.\sigma_2$</th><th>$Freq$</th><th>$Conf$</th></tr><tr><td>1</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>2</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>3</td><td>3</td><td>3</td><td>33%</td></tr><tr><td>4</td><td>4</td><td>3</td><td>60%</td></tr></table>	$P_{\text{left}}.\sigma_2$	$P.\sigma_2$	$Freq$	$Conf$	1	1	3	33%	2	2	3	33%	3	3	3	33%	4	4	3	60%
$P_{\text{left}}.\sigma_2$	$P.\sigma_2$	$Freq$	$Conf$																				
1	1	3	33%																				
2	2	3	33%																				
3	3	3	33%																				
4	4	3	60%																				
$(\emptyset, \{x_2\})$		(x_1, x_3, σ_2) 	<table><tr><th>$P_{\text{left}}.\sigma_2$</th><th>$P.\sigma_2$</th><th>$Freq$</th><th>$Conf$</th></tr><tr><td>1</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>2</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>3</td><td>3</td><td>3</td><td>33%</td></tr><tr><td>4</td><td>4</td><td>3</td><td>60%</td></tr></table>	$P_{\text{left}}.\sigma_2$	$P.\sigma_2$	$Freq$	$Conf$	1	1	3	33%	2	2	3	33%	3	3	3	33%	4	4	3	60%
$P_{\text{left}}.\sigma_2$	$P.\sigma_2$	$Freq$	$Conf$																				
1	1	3	33%																				
2	2	3	33%																				
3	3	3	33%																				
4	4	3	60%																				
$(\{x_1\}, \emptyset)$	No containment mappings																						
Level 2																							

$(\emptyset, \{x_1, x_2\})$		(σ_1, x_3, x_3) 	<table><tr><th>$P_{\text{left}.\sigma_2}$</th><th>$P.\sigma_1$</th><th>$P.\sigma_2$</th><th>Freq</th><th>Conf</th></tr><tr><td>1</td><td>0</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>2</td><td>0</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>3</td><td>0</td><td>3</td><td>3</td><td>33%</td></tr></table>	$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	Freq	Conf	1	0	1	3	33%	2	0	2	3	33%	3	0	3	3	33%
$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	Freq	Conf																			
1	0	1	3	33%																			
2	0	2	3	33%																			
3	0	3	3	33%																			
$(\emptyset, \{x_1, x_2\})$		$(\sigma_1, \sigma_2, x_3)$ 	<table><tr><th>$P_{\text{left}.\sigma_2}$</th><th>$P.\sigma_1$</th><th>$P.\sigma_2$</th><th>Freq</th><th>Conf</th></tr><tr><td>1</td><td>0</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>2</td><td>0</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>3</td><td>0</td><td>3</td><td>3</td><td>33%</td></tr></table>	$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	Freq	Conf	1	0	1	3	33%	2	0	2	3	33%	3	0	3	3	33%
$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	Freq	Conf																			
1	0	1	3	33%																			
2	0	2	3	33%																			
3	0	3	3	33%																			
$(\emptyset, \{x_1, x_2\})$		$(\sigma_1, x_3, \sigma_2)$ 	<table><tr><th>$P_{\text{left}.\sigma_2}$</th><th>$P.\sigma_1$</th><th>$P.\sigma_2$</th><th>Freq</th><th>Conf</th></tr><tr><td>1</td><td>0</td><td>1</td><td>3</td><td>33%</td></tr><tr><td>2</td><td>0</td><td>2</td><td>3</td><td>33%</td></tr><tr><td>3</td><td>0</td><td>3</td><td>3</td><td>33%</td></tr></table>	$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	Freq	Conf	1	0	1	3	33%	2	0	2	3	33%	3	0	3	3	33%
$P_{\text{left}.\sigma_2}$	$P.\sigma_1$	$P.\sigma_2$	Freq	Conf																			
1	0	1	3	33%																			
2	0	2	3	33%																			
3	0	3	3	33%																			
$(\{x_1\}, \{x_2\})$	No containment mappings																						
Level 3																							
$(\{x_1\}, \{x_2, x_3\})$	No containment mappings																						

Figure 4.4: Example Q_{left} and P .

4.6 Equivalent Association Rules

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

Let us first look at an example of the duplicate work that the algorithm presented until now performs. Consider Q_{left} , $Q_1 = (f_1(H_{\text{left}}), P)$, $Q_2 = (f_2(H_{\text{left}}), P)$; and $Q_3 = (f_3(H_{\text{left}}), P)$ in Figure 4.6 with f_1 , f_2 and f_3 as follows:

f_1	
x_1	u_1
x_2	u_2
x_3	u_2
x_4	u_3

f_2	
x_1	u_1
x_2	u_3
x_3	u_2
x_4	u_2

f_3	
x_1	u_1
x_2	u_2
x_3	u_3
x_4	u_3

Furthermore, consider pAR1: $Q_{\text{left}} \Rightarrow Q_1$; pAR2: $Q_{\text{left}} \Rightarrow Q_2$ and pAR3: $Q_{\text{left}} \Rightarrow Q_3$.

The confidence of the first rule (pAR1) equals the proportion of tuples from the answer set of Q_{left} where the values for variables x_2 and x_3 are equal (in the rhs those equal variables are represented by variable u_2 , and the lhs variable x_4 is represented by the rhs variable u_3). Similarly, the confidence of the second rule (pAR2) equals the proportion of tuples from the answer set of Q_{left} where the values for the variables x_3 and x_4 are equal (again the equal lhs variables x_3 and x_4 are represented by the rhs variable u_2 , and the lhs variable x_2 is represented by the rhs variable u_3). Since, due to the symmetry in the lhs pattern, the columns for x_2 , x_3 and x_4 are fully interchangeable in the answer set of Q_{left} , both rules convey precisely the same information: their confidences are equal. The third rule (pAR3) is yet another representation of the same association, but now the equal lhs variables x_3 and x_4 are represented by the rhs variable u_3 . Again, it has the same confidence as pAR1 and pAR2.

It is important to note that the above pARs only differ in the containment mappings f_1 , f_2 and f_3 that generate the rhs head. The algorithm discussed until now generates all these pARs, since we do not perform any check on the containment mappings generated in loop 3 of the overall approach (Section 4.2).

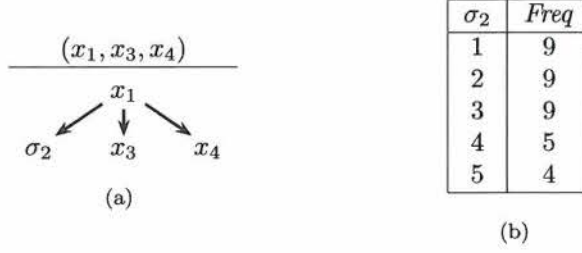


Figure 4.5: The fixed lhs and its frequency table for the example run in Section 4.5.

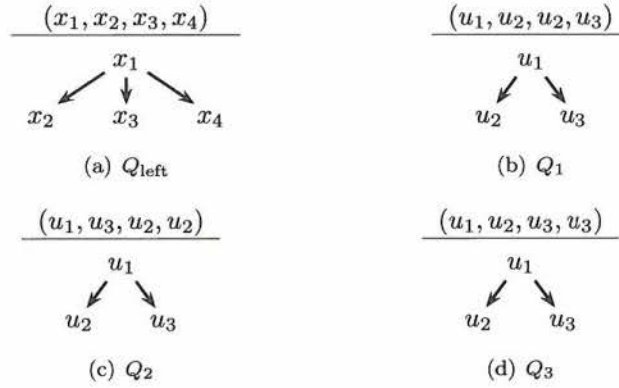
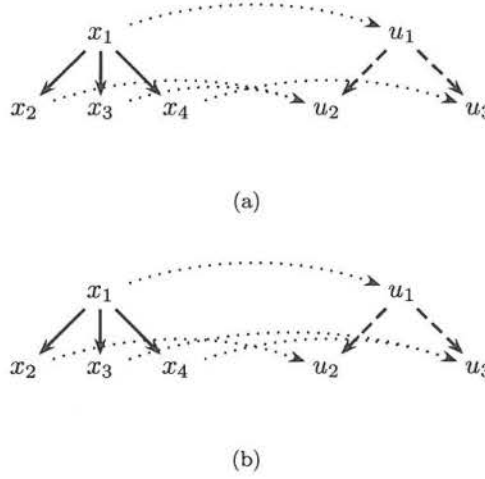


Figure 4.6: Queries to illustrate the duplicate work in the association-mining algorithm

In this Section, motivated by the above example, we consider the general problem of when two pARs $Q_{\text{left}} \Rightarrow_{\rho_1} Q_1$ and $Q_{\text{left}} \Rightarrow_{\rho_2} Q_2$ are equivalent, where Q_1 and Q_2 are of the form $(f_1(H_{\text{left}}), P)$ and $(f_2(H_{\text{left}}), P)$ for some common rhs pattern P , and containment mappings f_1 and f_2 from P_{left} to P . (Thus ρ_1 is $f_1|_{\Sigma_{\text{left}}}$ and ρ_2 is $f_2|_{\Sigma_{\text{left}}}$.) Since such two pARs differ only in f_1 and f_2 we can actually focus on f_1 and f_2 .

It is important to remember for the rest of this Section that P_{left} and P are arbitrary but fixed. Furthermore, without loss of generality we assume that the nodes of P_{left} and P are disjoint. This assumption greatly simplifies the representation of containment mappings by graphs, as we will see shortly.

Equivalent Containment Mappings Recall from Section 3.5.3 that an *isomorphism* from a parameterized tree pattern P_1 to a parameterized tree pattern P_2 is a homomorphism from P_1 to P_2 that is a bijection and that maps distinguished nodes to distinguished nodes, parameters to parameters and existential nodes to existential nodes. We now formalize equivalent containment mappings as follows: Two containment mappings f_1 and f_2 are *equivalent* if the structures $(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ are isomorphic. Specifically, there must exist isomorphisms (actually automorphisms)

Figure 4.7: The graph representations of f_1 and f_3 .

$g : P_{\text{left}} \rightarrow P_{\text{left}}$ and $h : P \rightarrow P$ such that $f_2 \circ g = h \circ f_1$.

Consider for instance f_1 and f_3 from the example above, then h swaps u_2 and u_3 , and g is the cyclic permutation $u_2 \mapsto u_3 \mapsto u_4 \mapsto u_2$.

4.6.1 Testing for Equivalence

To test for equivalent containment mappings efficiently, we represent them using graphs.

Graph representation of a containment mapping The graph representation of a containment mapping $f : P_{\text{left}} \rightarrow P$ is a directed, edge- and vertex-colored graph, with set of vertices $V_f = \text{Vertices}(P_{\text{left}}) \cup \text{Vertices}(P)$ and set of edges $E_f = \text{Edges}(P_{\text{left}}) \cup \text{Edges}(P) \cup \{(v, w) \mid f(v) = w\}$ (with the understanding that the edges of P_{left} and P go from parent to child). We use different colors for the edges of P_{left} , the edges of P and the pairs in f , and we also use different colors for the distinguished nodes, the existential nodes and the parameters.

As an illustration, Figure 4.7 shows the graph representation of f_1 and f_3 from our example in the introduction above.

Graph Isomorphism Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *colored isomorphic* if there exists a bijection $\varphi : V_1 \rightarrow V_2$, extended to edges $(v, w) \in E_1$ in a natural way by $\varphi(v, w) = (\varphi(v), \varphi(w))$, such that the colors of vertices and edges are preserved by φ , and such that $(v, w) \in E_1 \Leftrightarrow (\varphi(v), \varphi(w)) \in E_2$.

The following Lemma shows then the utility of the colored graph representation of containment mappings.

Lemma 6. *Two containment mappings are equivalent if and only if their colored graph representations are isomorphic.*

Proof. Let us start with the only-if direction. Consider two equivalent containment mappings f_1, f_2 from P_{left} to P . By definition of equivalent containment mappings, there exist isomorphisms $g : P_{\text{left}} \rightarrow P_{\text{left}}$ and $h : P \rightarrow P$ such that $f_2 \circ g = h \circ f_1$. Now take $\varphi = g \cup h$. Then, φ is clearly a bijection from V_{f_1} to V_{f_2} , and clearly preserves the colors of vertices and edges of G_{f_1} . Let $(v, w) \in E_{f_1}$. We show that φ is indeed an isomorphism from G_{f_1} to G_{f_2} . There are three possibilities:

1. $(v, w) \in \text{Edges}(P_{\text{left}})$. Note that then also $g(v, w) \in \text{Edges}(P_{\text{left}})$. We have:

$$\begin{aligned}
 (v, w) \in E_{f_1} &\Leftrightarrow (v, w) \in \text{Edges}(P_{\text{left}}) \\
 &\quad (g \text{ is an automorphism in } P_{\text{left}}) \\
 &\Leftrightarrow g(v, w) \in \text{Edges}(P_{\text{left}}) \\
 &\quad (\varphi(v, w) = g(v, w)) \\
 &\Leftrightarrow g(v, w) \in E_{f_2} \\
 &\quad (\varphi(v, w) \in E_{f_2}) \\
 &\Leftrightarrow \varphi(v, w) \in E_{f_2}
 \end{aligned}$$

2. $(v, w) \in \text{Edges}(P)$. Note that then also $h(v, w) \in \text{Edges}(P)$. We have:

$$\begin{aligned}
 (v, w) \in E_{f_1} &\Leftrightarrow (v, w) \in \text{Edges}(P) \\
 &\quad (h \text{ is an automorphism in } P) \\
 &\Leftrightarrow g(v, w) \in \text{Edges}(P) \\
 &\quad (\varphi(v, w) = h(v, w)) \\
 &\Leftrightarrow h(v, w) \in E_{f_2} \\
 &\quad (\varphi(v, w) \in E_{f_2}) \\
 &\Leftrightarrow \varphi(v, w) \in E_{f_2}
 \end{aligned}$$

3. $w = f_1(v)$. We have:

$$\begin{aligned}
 (v, w) \in E_{f_1} &\Leftrightarrow v = f_1(w) \\
 &\Leftrightarrow h(v) = h(f_1(w)) \\
 &\Leftrightarrow h(v) = f_2(g(w)) \\
 &\Leftrightarrow \varphi(v) = f_2(\varphi(w)) \\
 &\Leftrightarrow (\varphi(v), \varphi(w)) \in E_{f_2}
 \end{aligned}$$

So we can conclude that G_{f_1} and G_{f_2} are indeed colored isomorphic.

Let us now look at the if-direction. Let φ be the given isomorphism from G_{f_1} to G_{f_2} . Now take $g = \varphi|_{\text{Vertices}(P_{\text{left}})}$ and $h = \varphi|_{\text{Vertices}(P)}$. To prove that f_1 and f_2 are equivalent it suffices to show that:

1. g is an isomorphism from P_{left} to P_{left} ;

2. h is an isomorphism from P to P ; and
3. $f_2 \circ g = h \circ f_1$.

Items 1 and 2 hold because φ preserves the colors. For 3, let $v \in P_{\text{left}}$. Since φ is a graph isomorphism $f_2(\varphi(v)) = \varphi(f_1(v))$. We then have:

$$\begin{aligned} f_2(g(v)) &= f_2(\varphi(v)) \\ &= \varphi(f_1(v)) \\ &= h(f_1(v)) \end{aligned}$$

□

So, using graph isomorphism (to be precise, edge and vertex colored directed graph isomorphism), we can test for equivalence. Since our patterns are not very large, fast heuristics for graph isomorphism can be used. We use the program Nauty [36, 35], which is considered as the fastest heuristic for graph isomorphism. Nauty is very efficient for small, dense random graphs [39]. Since our graph representations are typically small (no more than 20 vertices) and dense, this works well in our case.

Theoretically this situation is not entirely satisfying, as graph isomorphism is not known to be efficiently (polynomial-time) solvable in general. We can show however that equivalence of our containment mappings is really as hard as the general graph isomorphism problem. This hardness argument is presented in the following Section. A special case of the equivalence problem that is solvable in polynomial time is presented in Section 4.6.3

4.6.2 Hardness Argument

First recall from graph theory that a graph $B = (V, E)$ is *bipartite* if V can be split in two disjoint parts, $V = V^a \cup V^b$ with $V^a \cap V^b = \emptyset$, such that for each $(v, w) \in E$ then $v \in V^a$ and $w \in V^b$. The vertices in V^a are called lhs vertices and those in V^b rhs vertices (lefthandside, righthandside).

We first reduce the problem of bipartite graph isomorphism to equivalence of our containment mappings. Let $B_1 = (V_1, E_1)$ and $B_2 = (V_2, E_2)$ be bipartite graphs. We describe an efficient construction that produces from B_1 and B_2 two association rules $(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ such that B_1 and B_2 are isomorphic if and only if the association rules are equivalent. This construction reduces the bipartite graph isomorphism problem to equivalence of containment mappings.

Without loss of generality, we assume that B_1 and B_2 have precisely the same multiset of outdegrees (for vertices of V_1^a and V_2^a), and precisely the same number of vertices in V_1^b and V_2^b . Indeed, if these conditions are not satisfied, then B_1 and B_2 are never isomorphic and our reduction can output some arbitrary P_{left}, P, f_1 and f_2 as long as $(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ are not equivalent.

The construction is now as follows. By the premisses on B_1 and B_2 , we may assume, without loss of generality, that $V_1^a = V_2^a$ and $V_1^b = V_2^b$. This can be accomplished by sorting the lhs vertices in each graph on their outdegrees and then numbering them arbitrarily (the rhs vertices can simply be numbered arbitrarily).

1. Construction of P_{left} : This is a tree with root called r_{left} and as children of the root, all lhs vertices. Moreover, each lhs vertex v has its own children as follows: if v has outdegree o , then v has o children denoted by $[v, 1], [v, 2], \dots, [v, o]$.
2. Construction of P : This is a tree with root called r_{right} , and exactly one child of the root, called c . Moreover, c has as children precisely all rhs vertices.
3. Construction of f_1 : We define $f(r_{\text{left}}) := r_{\text{right}}$, and define $f_1(v) := c$ for each lhs vertex v . Now for each such v , and all outgoing edges $(v, w_1), (v, w_2), \dots, (v, w_o)$ in B_1 , listed in some arbitrary order, we define $f_1([v, i]) := w_i$, for $i = 1, 2, \dots, o$.
4. The construction of f_2 is analogous to that of f_1 , but now we look at the outgoing edges in B_2 .

The construction is illustrated in Figure 4.8 for two bipartite graphs B_1 and B_2 . We now show the correctness of our reduction.

Lemma 7. B_1 and B_2 are isomorphic if and only if $(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ are isomorphic.

Proof. For the only-if direction, let ψ be an isomorphism from B_1 to B_2 . We define an isomorphism φ from $(P_{\text{left}}, P, f_1)$ to $(P_{\text{left}}, P, f_2)$ as follows:

- $\varphi(r_{\text{left}}) = r_{\text{left}}$, $\varphi(r_{\text{right}}) = r_{\text{right}}$ and $\varphi(c) = c$;
- $\varphi(v) = \psi(v)$, for any vertex of B_1 ;
- for any lhs vertex v of outdegree o , and any $i = 1, 2, \dots, o$, let w be the rhs vertex such that $f_1([v, i]) = w$. Then we define $\varphi([v, i]) := [\psi(v), j]$, where j is such that $f_2([\psi(v), j]) = \psi(w)$.

To verify that φ is indeed an isomorphism, we only check that $u = f_1([v, i]) \Leftrightarrow \psi(u) = f_2(\psi([v, i]))$. If $u = f_1([v, i])$ then (v, u) is an edge in B_1 and thus $(\varphi(v), \varphi(u)) = (\psi(v), \psi(u))$ is an edge in B_2 . Hence there exists a j such that, $\varphi(u) = f_2([\varphi(v), j])$, or equivalent, $\psi(u) = f_2([\psi(v), j])$. By definition of φ we have $\varphi([v, i]) = [\psi(v), j]$ and thus $\varphi(u) = f_2(\varphi([v, i]))$ as desired. Conversely, suppose $\varphi(u) = f_2(\varphi([v, i]))$. By definition of φ , we have $\varphi([v, i]) = [\psi(v), j]$ for some unique j , and $f_2([\psi(v), j]) = \psi(f_1([v, i]))$. Hence, $\psi(u) = \varphi(u) = \psi(f_1([v, i]))$, and thus $u = f_1([v, i])$ as desired.

For the if-direction, let φ be an isomorphism from $(P_{\text{left}}, P, f_1)$ to $(P_{\text{left}}, P, f_2)$. We define an isomorphism ψ from B_1 to B_2 . Actually, ψ is simple φ restricted to the vertices of B_1 . Indeed,

$$\begin{aligned}
 (v, w) \in E_1 &\Leftrightarrow \exists i : f_1([v, i]) = w \\
 &\Leftrightarrow \exists i : f_2(\varphi([v, i])) = \varphi(w) \\
 &\Leftrightarrow \exists j : f_2([\varphi(v), j]) = \varphi(w) \\
 &\Leftrightarrow (\varphi(v), \varphi(w)) \in E_2
 \end{aligned}$$

□

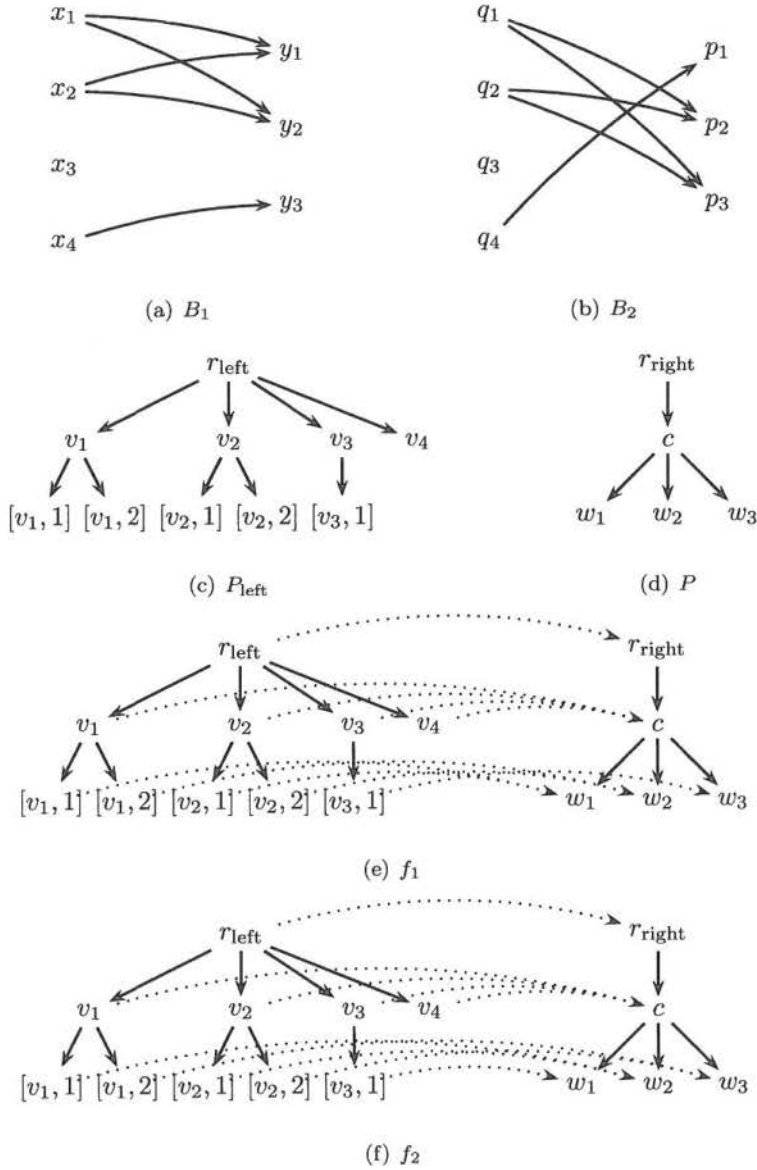


Figure 4.8: Illustration of the construction of pARs from bipartite graphs.

We can already conclude from this reduction that equivalence of pARs is really as hard as isomorphism of bipartite directed graphs. The latter problem, however, is well known to be as hard as isomorphism of general directed graphs. Indeed, any directed graph $G = (V, E)$ can be transformed into the bipartite directed graph $B(G) := (V \cup E, \{(v, (v, w)) \mid (v, w) \in E\} \cup \{((v, w), w) \mid (v, w) \in E\})$, and it is easily verified that G_1 and G_2 are isomorphic if and only if $B(G_1)$ and $B(G_2)$ are isomorphic.

So, we can now conclude that equivalence of our pARs is really as hard as the general graph isomorphism problem. But as we show next, we can still capture an important special case in polynomial time, so that the general graph isomorphism heuristics only have to be applied on instances not captured by the special case.

4.6.3 Polynomial Case

The special efficient case is to check whether $(P_{\text{left}}, P, f_1)$ and $(P_{\text{left}}, P, f_2)$ are already isomorphic with g the identity, i.e., whether the structures (P, f_1) and (P, f_2) are already isomorphic. So, we look for an automorphism h of P such that $f_2 = h \circ f_1$. This can be solved efficiently by a reduction to node-labeled tree isomorphism. As explained in Section 3.4, if we know the tree T underlying P , then P is characterized by the pair (Π, Σ) , and thus (P, f) is characterized by (Π, Σ, f) . We can view this triple as a labelling of T , as follows: We label every node y of P with a triple $(b_\Pi, b_\Sigma, f^{-1}(y))$, where b_Π is a bit that is 1 iff $y \in \Pi$; b_Σ is a bit that is defined likewise; and $f^{-1}(y)$ is the set of nodes of P_{left} that are mapped by f to y . Then (P, f_1) and (P, f_2) are isomorphic if and only if the corresponding node-labeled trees are isomorphic, and the latter can be checked in linear time using canonical ordering [3, 9].

4.6.4 The Algorithm

We are now in a position to describe how our general algorithm must be modified to avoid equivalent association rules. There is only extra checking to be done in loop 3 (recall Sections 4.2 and 4.3). For each new containment mapping f from P_{left} to P , we canonize the corresponding node-labeled tree and we check if the canonical form is identical to an earlier generated canonical form; if so, f is dismissed. We can keep track of the canonical forms seen so far efficiently using a trie data structure. If the canonical form was not yet seen, we can either let f through to loop 4, if the presence of duplicates in the output is tolerable for the application at hand, or we can perform the colored graph isomorphism check of Section 4.6.1 with the containment mappings previously seen, to be absolutely sure that we will not generate a duplicate.

5

Experimental Results

The algorithms presented in the previous chapters suggest a database-oriented implementation in SQL. Hence, we implemented both algorithms in C++ with embedded SQL, and we used DB2 UDB v8.2 as the relational database system. In this Chapter, we give results of the experiments performed with this prototype implementation. In Section 5.2 we give results of some smaller experiments we performed on random and real-life datasets, and in Section 5.3 we give results of a larger experiment using real-life data from Ecology. In this last Section, we also give some tips on how our prototype implementation can be used for real-life applications.

Before we give the results of the experiments, we first introduce an interactive browsing tool called *Certhia* in Section 5.1. As already mentioned in Section 3.6, the pattern database, that is the result of the tree-query-mining algorithm in Chapter 3, is an ideal platform for a tool for browsing the mined patterns, and generating association rules.

5.1 Certhia: Pattern and Association Browsing

In this Section we introduce an interactive tool, called *Certhia*, for browsing the frequent tree patterns, and generating association rules.

As already noted in Section 3.6, the result of our tree query mining algorithm in Chapter 3 is a structured database, called a pattern database, containing all frequency tables for each tree T that was investigated. This pattern database is an ideal platform for an interactive tool for browsing the frequent queries. However, this pattern database is also an ideal platform for generating association rules as explained in Section 4.2, since the first two loops of the association-rule-mining algorithm are exactly our tree-query-mining algorithm.

In a typical scenario for *Certhia*, the user draws a tree shape, marks some nodes

as existential, marks some others as parameters, instantiates some parameters by constants, but possibly also leaves some parameters open. The browser then returns, by consulting the appropriate frequency table in the database, all instantiations of the free parameters that make the pattern frequent, together with the frequency. The user can then select one of these instantiations, set a minconf value, and ask the browser to return all rhs's that form a confident association with the selected pure tree query as lhs.

In another scenario the user lets the browser suggest some frequent tree patterns to choose from as an lhs. When the user connects the browser with a particular pattern database, the browser builds an index of all parameterized tree patterns present in the pattern database. Afterwards, the browser suggests frequent tree patterns to the user by letting him "scroll", for each tree, through all its parameterized tree patterns that have frequent instantiations. The user can then choose one of the suggested parameterized tree patterns, and ask the browser to return all frequent instantiations or to generate association rules with this tree pattern as lhs.

Some screenshots of Certhia are given in Figure 5.1, Figure 5.2 and Figure 5.3.

- In Figure 5.1, the user draws a tree, marks some nodes as existential, some others as parameters, instantiates some parameters with constants, and asks the browser to return all possible instantiations of the remaining parameters and the corresponding frequencies.
- In Figure 5.2, the user asks the browser to return all association rules for a fixed lhs. The user selects a rhs in the dialogue box and asks the browser to return the instantiations and the corresponding confidences.
- In Figure 5.3, the browser suggests some frequent tree patterns where the user can choose from.

Efficiency The preprocessing step, i.e., the building up of the pattern database with frequent tree patterns, is of course a hugely intensive task. First because the large data graph must be accessed intensively, and secondly because the number of frequent patterns is huge. In Section 5.2.2 we show that this preprocessing step can be implemented with satisfactory performance. Also, in scientific discovery applications it is no problem, indeed typical, if a preprocessing step takes a few hours, as long as after that the interactive exploration of the found results can happen very fast. And indeed we found that the actual generation of association rules is very fast. This is also shown in Section 5.2.2.

5.2 Smaller Experiments

In this Section, we report on some experiments performed using our prototype implementation applied to both real-life and synthetic datasets to show that our approach is indeed workable.

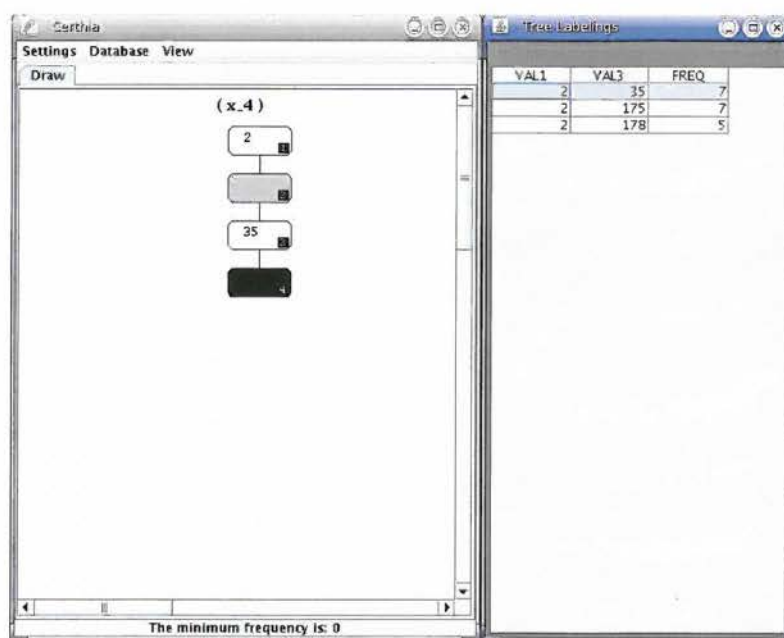


Figure 5.1: Screenshot of Certhia: the user draws a pattern and asks for the instantiations.

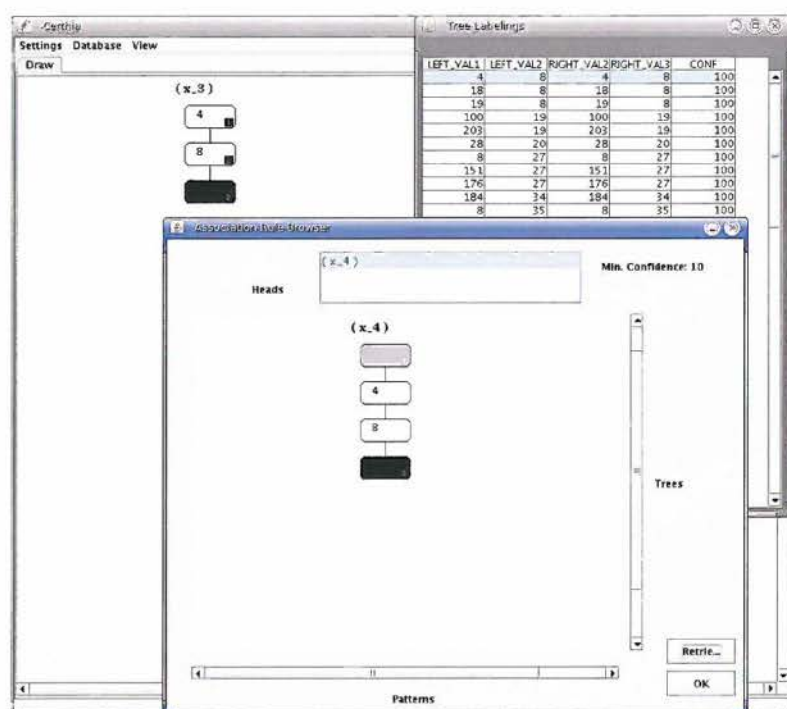


Figure 5.2: Screenshot of Certhia: the user asks for all association rules for this lhs.

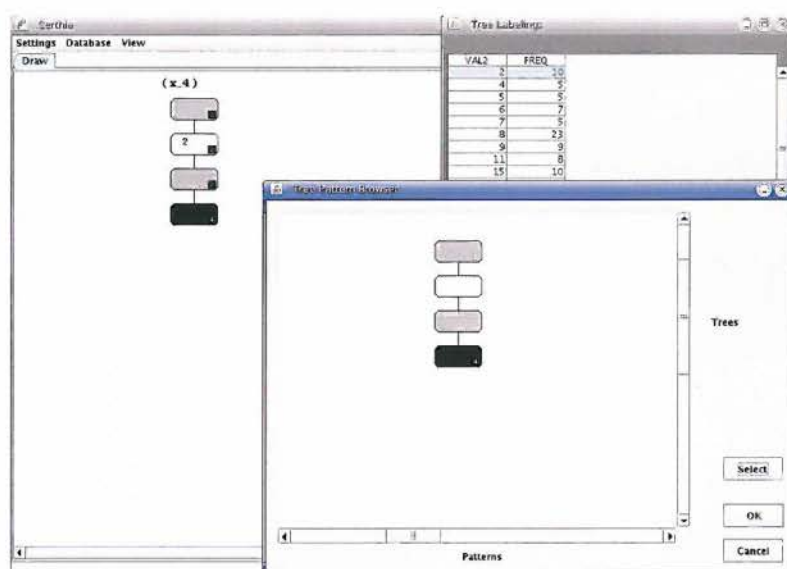


Figure 5.3: Screenshot of Certhia: the browser suggests frequent tree patterns.

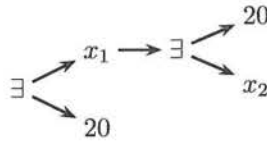
5.2.1 Real-Life Datasets

We have worked with a food web, a protein interactions graph, and a citation graph. For each dataset we built up a pattern database using the following parameters:

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

As we set rather generous limits on the maximum size of trees, or on the minimum frequency threshold, each run took several hours.

The **food web** [37] 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:



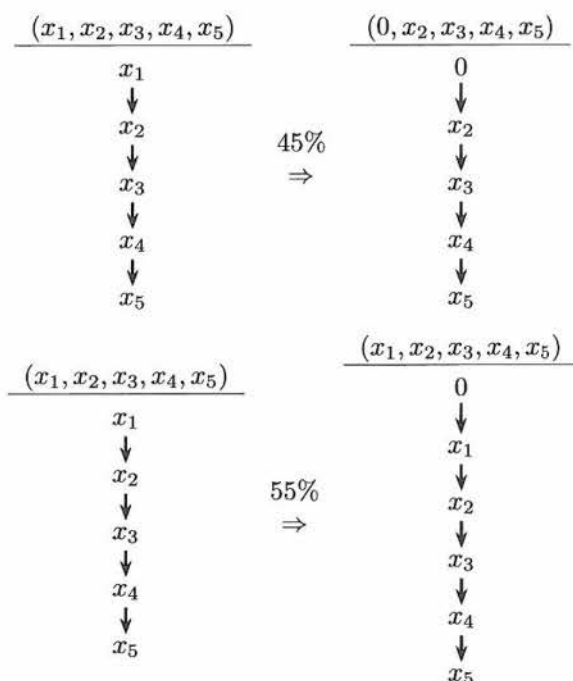
This is really a rather arbitrary example, just to give an idea of the kind of complex patterns that can be mined. Note also that, thanks to the constant 20 appearing twice, this is really a non-tree shaped pattern: we could equally well draw both arrows to a single node labeled 20.

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 adenocarpis*, 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:

$$\begin{array}{ccc}
 \begin{array}{c} \overline{(x_1, x_2)} \\ x_1 \\ \downarrow \\ \exists \\ \downarrow \\ \exists \\ \downarrow \\ \exists \\ \downarrow \\ x_2 \end{array} & \Rightarrow & \begin{array}{c} \overline{(x_1, x_2)} \\ x_1 \\ \downarrow \\ \exists \\ \downarrow \\ 20 \\ \downarrow \\ \exists \\ \downarrow \\ x_2 \end{array}
 \end{array}$$

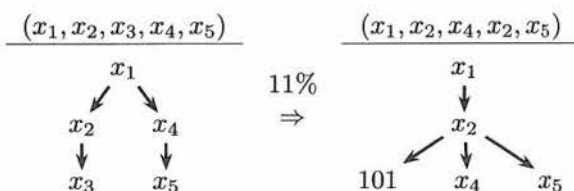
Indeed, the rule shown above turned up with 89% confidence! For 89% of all pairs of species that are linked by a path of length four, *Orthotylus adenocarpis* is involved in between.

Two other rules we discovered are:

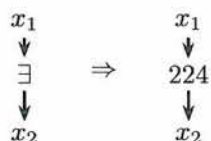


Since $45\% + 55\% = 100\%$, these rules together say that each path of length 5 either starts in 0, or one beneath 0. This tells us that the depth of the food web equals 6. Constant 0 turns out to denote the Scotch Broom itself, which is the root of the food web.

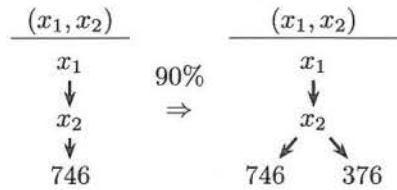
Another rule we mined, just to give a rather arbitrary example of the kind of rules we find with our algorithm, is the following:



The **protein interaction graph** [27] 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 occurs. Indeed, we discovered the following association rule with 10% confidence, indicating that protein #224 is highly connected:

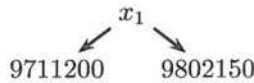


We also found the following rule:

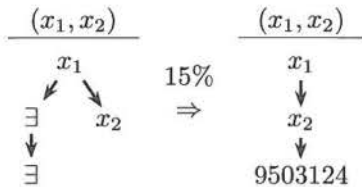


This rule expresses that almost all interactions that link to protein 746 also link to protein 376, which unveils a close relationship between these two proteins.

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).



One of the discovered rules is the following:



This rule shows that paper 9503124 is an important paper. In 15% of all “non-trivial” citations (meaning that the citing paper cites at least one paper that also cites a paper), the cited paper cites 9503124.

5.2.2 Performance

While our prototype implementations are not tuned for performance, we still conducted some preliminary performance measurements, with encouraging results. The experiments were performed on a Pentium IV (2.8GHz) architecture with 1GB of internal memory, running under Linux 2.6.

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 [38]. Various random graph models have been proposed in this respect, of which we have used the “copy model” for Web graphs [29, 5]. 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 of the tree query mining algorithm 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 5.4, show that the performance of these runs is quite adequate.

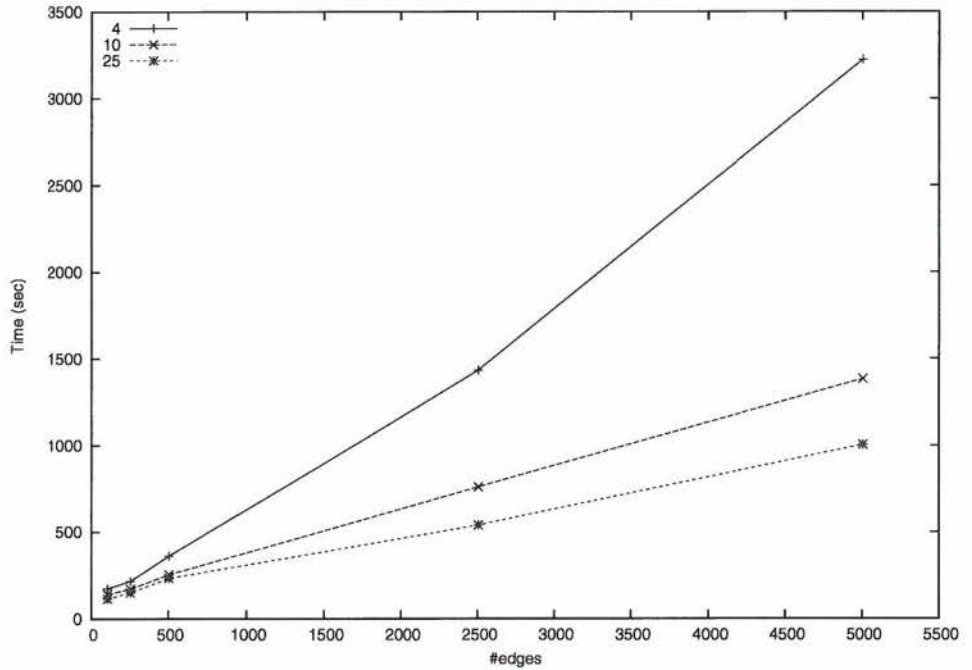


Figure 5.4: Performance on Web graphs.

Uniform random graphs We have also experimented with the well-known Erdős-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 of the tree-query-mining algorithm 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 5.5, 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).

On these uniform random graphs we also conducted some experiments to check the performance of the association-rule mining algorithm. We found the actual generation of association rules (i.e., loops 3 and 4, assuming that a pattern database is already build up) to be very fast. For instance, Figure 5.6 shows the performance of generating association rules for two different (absolute) values of minconf, against a pattern database built up for a random graph with 33 nodes and 113 edges, an absolute minsup of 25, and all trees up to size 7. We see that associations are generated with constant overhead, i.e., in linear-output time. The coefficient is larger for the larger minconf, because in this experiment we have counted instantiated rhs's, and per rhs query less instantiations satisfy the confidence threshold for larger such thresholds.

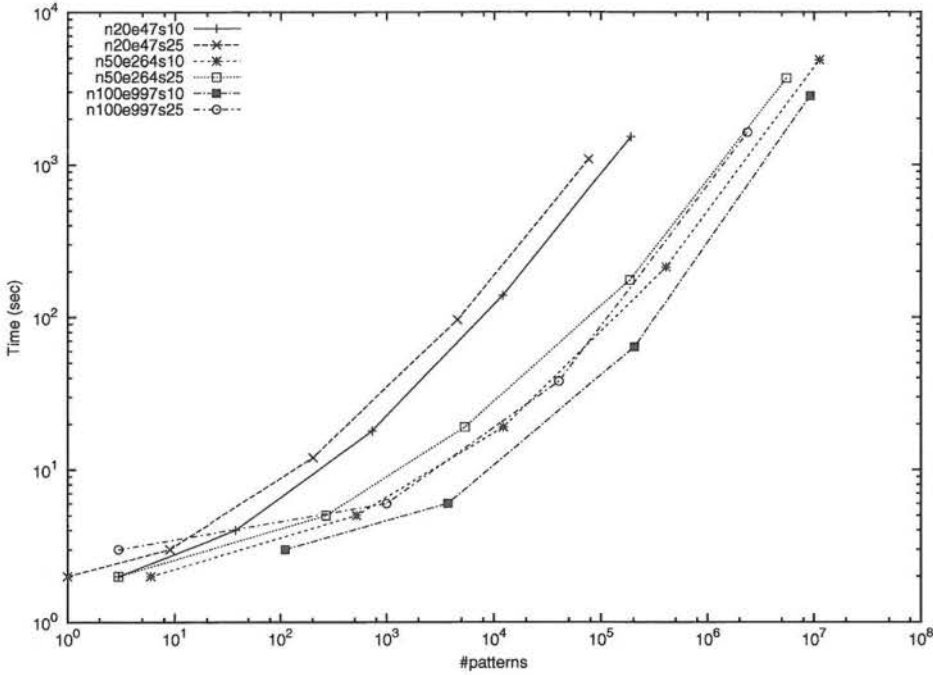


Figure 5.5: Performance in terms of number of discovered patterns.

Had we simply counted rhs's regardless of the number of confident instantiations, the two lines would have had the same slope.

Performance issues One major performance issue that we have not addressed in the present study is that some of the SQL queries that are performed due to pattern generation take a very long time (in order of hours) to answer by the database system. This happens in those cases where the data graph is large (5000 edges or more) with many cycles, and the candidate patterns are large (6 nodes or more). Certainly, some SQL queries can be hand-optimized (or replaced by a combination of simpler queries), to alleviate these performance problems, but we leave this issue to future research.

5.3 Ecology Experiment

In this Section we explain how our algorithms can be used to find interesting patterns and rules in a dataset from *Ecology*: the branch of Biology that is concerned with the relationships between organisms and their environment. The dataset we use here comes from the Animal Ecology research group of the University of Antwerp. It contains *natal-dispersal* data of a Great Tit population around Antwerp in Belgium. Natal dispersal is the movement an individual makes from its birth site to the place where it will reproduce. The dataset is discussed in more detail in Section 5.3.1.

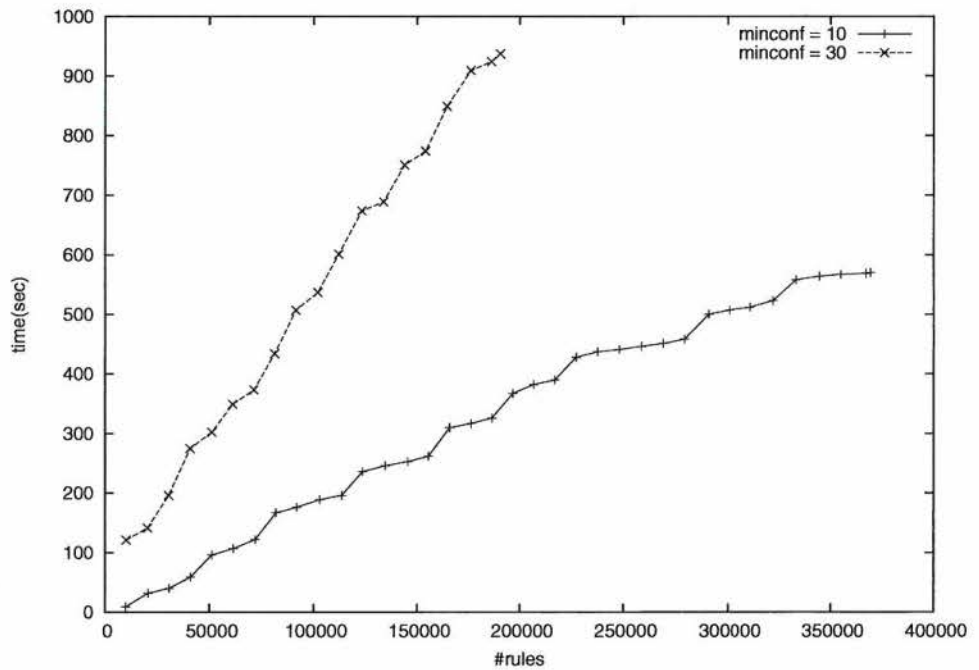


Figure 5.6: Performance in terms of number of discovered rules.

Since our algorithm requires a data graph as input, we need to construct an appropriate data graph for the natal-dispersal data. In Section 5.3.2 we explain how we constructed a natal-dispersal data graph. When the data graph is constructed, we can run our tree-query-mining algorithm. After the algorithm has produced a sufficiently large pattern database, we can use Certhia to browse for interesting patterns and to generate rules. In Section 5.3.3 we explain how to search for interesting patterns and we show some interesting patterns and rules we mined from the natal-dispersal data graph.

5.3.1 Natal-Dispersal Dataset

We use a dataset that contains natal dispersal data for a Great Tit population in a (study) area, called Boshoeck, near Antwerp in Belgium [34]. Natal dispersal is the movement an individual makes from its birth site to the place where it will reproduce itself. For a Great Tit this is the movement it makes from the nest-box where it was born, to the nest-box where it will breed. Note that individuals only make one such dispersion in a life cycle. Natal-dispersal data is used to find correlations between environmental properties and dispersal behaviour. It is useful when studying species persistence and evolution.

The considered dataset was collected between 1994 and 1999, by visiting nests, counting birds and ringing young birds. It contains 424 natal dispersions, and for

each dispersion we have the following information:

- the *year* in which the dispersion was made;
- the *birth-nest-box number*: number of the nest-box where the Great Tit is born;
- the *birth plot*: plot where the birth nest-box is situated;
- the *breed-nest-box number*: number of the nest-box where the Great Tit bred;
- the *breed plot*: plot where the breed-nest-box is situated;
- the *identifier* of the Great Tit who made the dispersion; and
- the *sex* of the Great Tit who made the dispersion.

A *plot* is a fragment of forest in the study area Boshhoek. Boshhoek is divided in 13 different plots. The combination of a nest-box number and its plot form an unique identifier for a nest-box.

We also have some extra information for each nest-box:

- the *plot* where the nest-box is situated;
- the kind of *wood* in the surroundings of the nest-box;
- the *success rate*: number of young ones that left the nest-box divided by the years the nest-box was used;
- the *occupancy*: number of years the nest-box was used divided by the number of years the nest-box was studied; and
- the *parasite rate*: number of birds in the nest-box with parasites divided by the number of checked birds for this nest-box.

In Figure 5.7 an illustration of the input dataset is given.

In the next Section we describe how we construct a data graph from this dataset.

5.3.2 Graph Construction

For the natal-dispersal dataset discussed in the previous Section, we now need to construct an appropriate data graph. This construction seems like a straightforward task, however there are some things we need to consider:

1. The tree-query mining algorithm presented in Chapter 3 expects as an input a directed graph. Recall from the definition on page 12, that a directed graph is a finite set of nodes, and a finite set of ordered pairs of nodes, called edges. Clearly, this is the simplest notion of a directed graph, since edges and nodes are not labeled. However, this can have as a consequence that we need to encode edge and node labels in some cases. But as already mentioned in Section 1.1, node labels and edge labels can easily be simulated using constants.

year	birth box	birth plot	breed box	breed plot	bird id	sex
1995	5	HM	4	HN	25V56330	male
1995	87	KB	43	KB	25V56342	female
1995	3	HN	6	HM	25V56351	female
1994	76	KB	6	HN	34V72683	male
1994	60	KB	8	VS	36V14001	female
...						

nest-box	plot	wood	success	occupancy	parasite
5	HM	EK	7.58	0.92	0.43
6	HM	EK	7	0.85	0.33
3	HN	EK	6.7	0.77	0.88
4	HN	EK	8.5	0.46	0.75
6	HN	EK	6.88	0.62	0.44
43	KB	EG	7.67	0.69	0.3
60	KB	EG	5.73	0.85	0.4
76	KB	EG	9.1	0.77	0.25
87	KB	EG	8	0.62	0.17
8	VS	EK	7.25	0.62	0
...					

Figure 5.7: Illustration of the natal dispersal dataset

2. The constructed data graph must have the same semantics as the input dataset. Note that this does not necessarily mean that all information of the dataset must be contained in the data graph. As we will see later on, it is not always feasible to put all information of the dataset in the data graph due to efficiency reasons.
3. While creating a data graph it is useful to think about the kind of patterns the owners of the dataset are interested in. Sometimes extra node and/or edge labels can facilitate the browsing afterwards, or some information can be dismissed since we are not interested in it afterwards.
4. A last point we need to consider is efficiency. As already mentioned in Section 5.2.2, some SQL queries in our implementation may take a very long time for large data graphs with many cycles. When creating the data graph it is important to avoid putting too much information in the graph. This because if we put more information in the graph, the graph will have more nodes and edges, hence it will be larger. In Chapter 6, we show how occurrences of cycles can affect efficiency, and hence need to be minimized.

We are now ready to describe how we constructed a data graph for the natal-dispersal dataset. We start by describing the nodes:

- **nesting place:** we create a unique node, called nesting place, for each nest-box, by combining its nest-box number with the plot where it is situated in, for instance HM_6;

- **nest-box**: a special node that is used to ‘label’ all nesting places;
- **bird**: we create a node for each Great Tit that makes a dispersion by using its unique identifier provided by the input dataset;
- **plot**: we create a node for each of the 13 different plots;
- **wood**: we create a node for each of the 3 kinds of wood that occur in Boshhoek;
- **sex**: we create a node for **male** and one for **female**;
- **success**: we divide all values for the success rate into 3 classes: low, medium, and high, and we create a node for each class;
- **occupancy**: we also divide all values for the occupancy rate into 3 classes: low, medium, and high, and we create a node for each class; and
- **parasite**: as with success and occupancy, we divide all values for the parasite rate into 3 classes: low, medium, and high, and we create a node for each class.

The following edges are drawn in the data graph for the natal-dispersal dataset:

- **nesting place** → **nest-box**, for labeling the nesting places;
- **nesting place** → **wood**, if the surroundings of the nesting place have that kind of wood;
- **nesting place** → **plot**, if the nesting place is situated in that plot;
- **nesting place** → **success**, if the nesting place belongs to that success class;
- **nesting place** → **occupancy**, if the nesting place belongs to that occupancy class;
- **nesting place** → **parasite**, if the nesting place belongs to that parasite class;
- **nesting place** → **bird**, if the bird is born in that nesting place;
- **bird** → **nesting place**, if the bird breeds in that nesting place; and
- **bird** → **sex**, if the bird has that sex.

The construction of the data graph for the natal-dispersal dataset is rather straightforward. Almost all information from the input dataset is contained in the data graph except for the year when the dispersion happened. This was feasible since there are only 424 dispersions to consider. The owners of the data suggested to dismiss the year since they are mainly interested in finding correlations between properties of the nesting place where a bird is born and the nesting place where a bird has bred. The special node **nest-box** is used to label all nesting places, which facilitates browsing as you will see in Section 5.3.3

In Figure 5.8 the data graph for the dataset in Figure 5.7 is given.

The constructed natal-dispersal data graph is now ready to be input in the tree-query-mining algorithm. It contains a total of 714 nodes and 2823 edges. We let

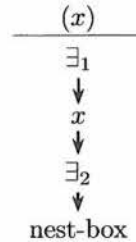
our algorithm from Chapter 3 generate a pattern database for tree patterns with a maximum of 7 nodes, and with a minimum support of 5. The minimum support is rather low, but it is sufficient since the information represented in the data graph is rather sparse.

In the next Section we show how to browse for interesting patterns and rules in this data graph and we give some examples.

5.3.3 Tree-Query Browsing

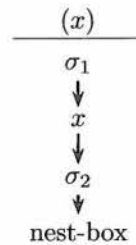
In this Section we show how to search for interesting patterns and rules in real-life datasets, by giving example patterns and rules we mined from the natal-dispersal data graph.

Let us start with some simple examples. Consider the following tree query we mined:



This tree query describes all Great Tits that make a natal dispersion. In fact, since each Great Tit makes only one such movement, it describes all natal dispersions present in our dataset. The frequency of this tree query, 424, is the number of Great Tits or natal dispersions present in our dataset. In this tree query we use one special “nest-box” node to ensure that \exists_2 is matched with nesting places only. If we had not used this node, \exists_2 could also be matched with a “success”-node, an “occupancy”-node, a “wood”-node, a “parasite”-node or a “plot”-node.

Consider the following tree query, that describes for each pair of nesting places, the Great Tits that made a dispersion between them:



Our algorithm did not find any pair of nesting places where between 5 or more Great Tits made a dispersion. The following tree query shows that almost all of the 424 dispersions are made between unique pairs of nesting places:

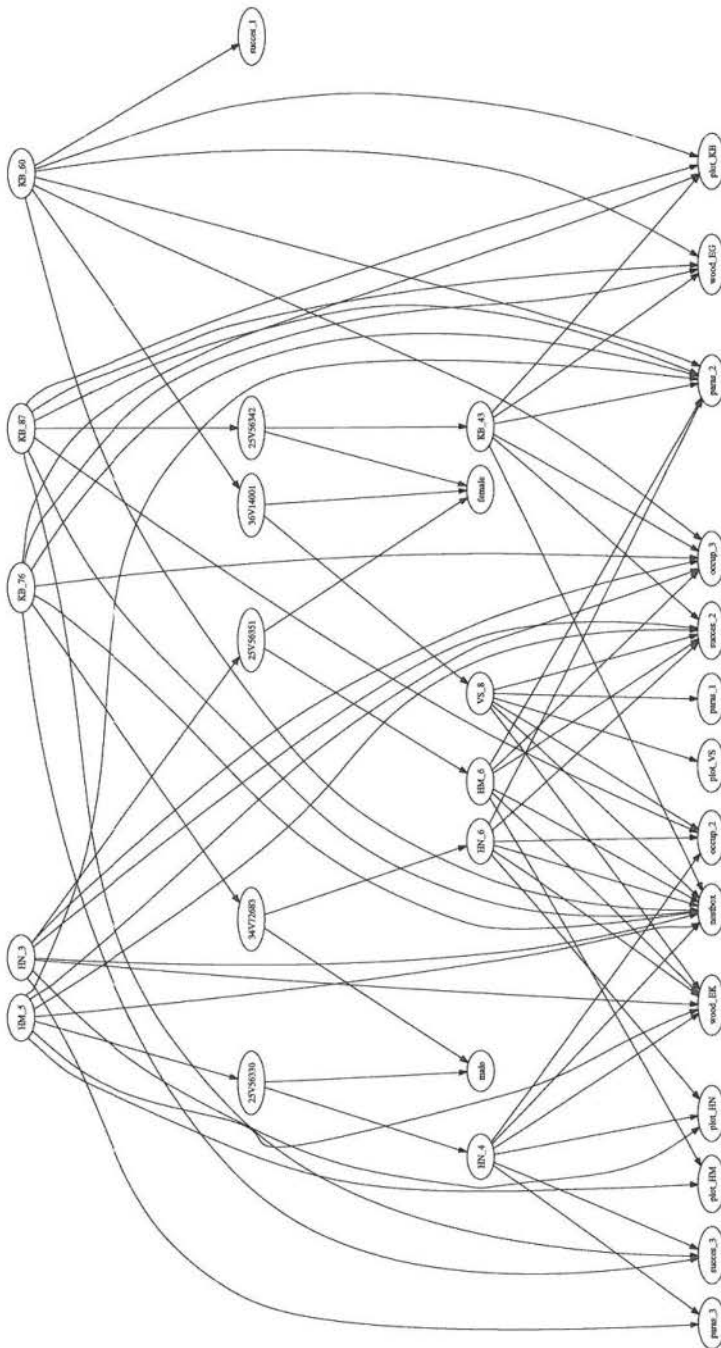
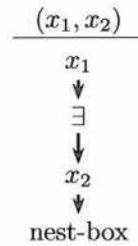
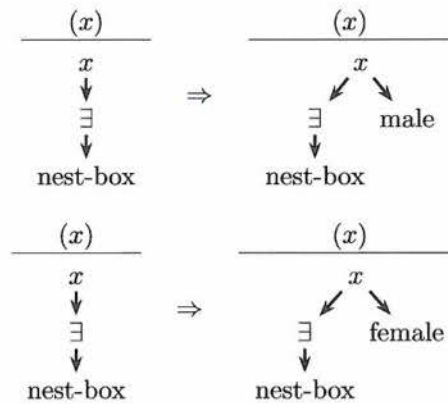


Figure 5.8: Data graph for the example dataset in Figure 5.7.



The frequency of this tree query is 416, hence 98% of the dispersions are made between an unique pair of nesting places.

The following rules we mined, show us that 56% of the dispersions are made by male Great Tits and 44% by female Great Tits:



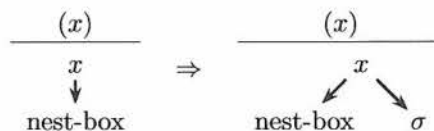
The patterns and rules mined above are just simple didactical examples to show what kind of patterns and rules can be found in the natal-dispersal data graph. In fact they only confirm that we constructed the data graph correctly.

We consulted the owners of the dataset to find out in what kind of patterns and correlations (rules) they are interested. They are particularly interested in patterns and rules that express possible correlations between properties of the nesting places, such as success, occupancy and wood, and dispersal behaviour. They are also interested in dispersal behaviour differences between male and female Great Tits.

In the next Paragraphs we show how we can combine pattern browsing and rule generation with simple SQL queries on frequency and confidence tables to find possible correlations between the properties of the nesting places and the dispersal behaviour.

Properties of nesting places versus dispersal behaviour

Consider the following rule:



Clearly, x represents a nesting place and σ a property of a nesting place, such as medium success, low occupancy and high parasite; a bird; or the special **nest-box**-node. Hence the confidence table for this rule contains for each possible property, bird or the **nest-box**-node, the proportion of nesting places that have that particular property, the proportion of nesting places from where that bird leaves the nest or the proportion of nesting places that are labeled as a nest-box (clearly 100% of the nesting places). If σ is mapped to a bird, this proportion typically will be very small, since each bird only leaves one nest.

In the construction of the data graph we assumed that we know something about all properties for each nesting place (**nil** if a property was not checked for a particular nesting place). Now, we will use SQL to limit σ to a particular property, such as success and occupancy. We can then see how the nesting places are distributed over the different values for this property. For instance, if we limit σ to success, we get the following restricted version of the confidence table:

Success	Percentage
nil	2%
low	22%
medium	52%
high	24%

So, we can conclude that 22% of the nesting places have a low success rate, 52% a medium success rate, 24% a high success rate and for 2% of the nesting places the success rate was not calculated.

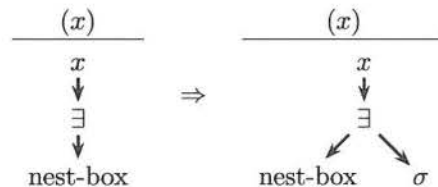
We did the same for occupancy, wood and parasite:

Occupancy	Percentage
nil	2%
low	17%
medium	45%
high	36%

Wood	Percentage
EK	29%
EG	61%
BE	10%

Parasite	Percentage
nil	2%
low	16%
medium	50%
high	32%

Now also consider the following rule, which says something about the properties of the nesting places where birds breed:



Clearly, x is mapped to birds (equivalent with dispersions), \exists to nesting places and σ to properties of nesting places, birds or the special **nest-box**-node. So, the confidence table of this rule says something about the proportion of dispersions that go to a nesting place with a particular property, the proportion of dispersions that go to a nesting place from where a particular bird leaves the nest (typically very small), or the proportion of dispersions that go to a nesting place (clearly, 100%). Again, using

SQL on the confidence table, we will limit σ to a particular property, such as success. Hence, then we can say something about the distribution of the dispersions over the different properties of the nesting places where dispersions arrive. For instance, for the success rate this will be as follows:

Success	Percentage
nil	2%
low	18%
medium	58%
high	22%

So, we can conclude that 18% of the dispersions go to a nesting place with a low success rate, 58% to a nesting place with a medium success rate, 22% to nesting places with a high success rate and for 2% nothing is known about the success rate of the breeding-nesting place.

If there is no correlation between the properties of nesting places and the dispersal behaviour, dispersions and nesting places should be distributed equally over the different success classes. However, we see that there are some minor differences: there are less dispersions to nesting places with a low success rate than expected, more dispersions to nesting places with a medium success rate than expected, and slightly less dispersions to nesting places with a high success rate than expected. In general, the differences are not that high to conclude that there is a clear correlation between the success rate of a nesting place and dispersions arriving at that nesting place.

We did the same for occupancy, wood and parasite. The restricted confidence tables then look as follows:

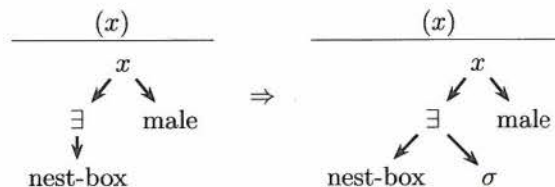
Occupancy	Percentage
nil	2%
low	11%
medium	39%
high	48%

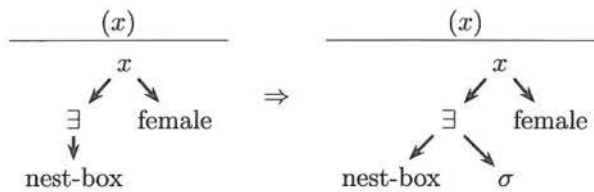
Wood	Percentage
EK	28%
EG	62%
BE	10%

Parasite	Percentage
nil	1%
low	13%
medium	56%
high	30%

From the above restricted confidence tables we can conclude that there is no clear correlation between the parasite rate of a nesting place, the kind of wood surrounding the nesting place, and the dispersal behaviour. In contrast, there is a clear correlation between the occupancy of a nesting place, and the dispersal behaviour. If a nesting place has a low occupancy rate, less dispersions than expected will go to this nesting place. However, when the nesting place has a high occupancy rate, more dispersions than expected will go to this nesting place.

Now we will check if there is a difference in dispersal behaviour between male and female Great Tits. We use the following rules for this:





The rules are exactly the same as above, only do we restrict ourselves to dispersions made by birds of a particular sex, male or female. As above, we use SQL to limit σ to a particular property. The restricted confidence tables then look as follows for male dispersions:

Success	Percentage
nil	2%
low	17%
medium	59%
high	22%

Occupancy	Percentage
nil	2%
low	10%
medium	39%
high	49%

Wood	Percentage
EK	27%
EG	62%
BE	11%

Parasite	Percentage
nil	2%
low	11%
medium	57%
high	30%

and for female dispersions:

Success	Percentage
nil	1%
low	20%
medium	56%
high	23%

Occupancy	Percentage
nil	2%
low	12%
medium	40%
high	46%

Wood	Percentage
EK	29%
EG	61%
BE	10%

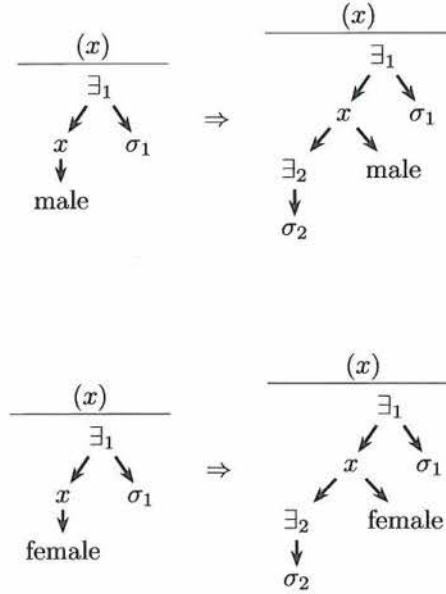
Parasite	Percentage
nil	0%
low	16%
medium	54%
high	30%

We can conclude from these restricted confidence tables that especially for male Great Tits there is a correlation between some properties of the nesting place where they breed, such as occupancy and parasites, and the dispersal behaviour. For female Great Tits this correlation is not so clear.

The above rules are good examples of how we can combine pattern and rule mining with SQL to find interesting correlations. In the next Paragraph we will use SQL again to study if there is a correlation between properties of the nesting place where the bird is born and properties of the nesting place where the bird breeds, and we want to know if it differs for both sexes.

Properties of the nesting place where the bird is born versus properties of the nesting place where the bird breeds.

Consider the following rules:



If we limit σ_1 to properties of nesting places using SQL, then the lhs of the above rules represents all dispersions made by a male or female Great Tit that start from a nesting place with property σ_1 . If we also limit σ_2 to properties of nesting places using SQL, then the rhs of the above rules represents all dispersions made by a male or female Great Tit that go from a nesting place with property σ_1 to a nesting place with property σ_2 . The complete rules then tell us something about the proportion of dispersions, made by a male or a female Great Tit, that go from a nesting place with property σ_1 , to a nesting place with property σ_2 .

Note that we do not need to use the special **nest-box-node** in these rules since we make sure, by adding a constant, namely the sex of the Great Tit, that x can only be mapped to a Great Tit. Hence, \exists_1 can only be a nesting place. In the rhs of the rules, \exists_2 can only be a nesting place and not the sex of the Great Tit, since it has an outgoing edge.

We limit in the confidence tables, of the above rules, σ_1 and σ_2 to the properties occupancy, success, wood and parasite. As a result we get a restricted confidence table for the male rule and one for the female rule. A fragment of the restricted confidence table for the male-rule looks as follows:

σ_1	σ_2	confidence
low occupancy	low success	30%
low occupancy	medium success	48%
low occupancy	high success	22%
wood EG	low occupancy	7%
wood EG	medium occupancy	44%
wood EG	high occupancy	47%
low parasite	medium occupancy	44%
low parasite	high occupancy	49%
...		

For instance the tuple (low occupancy, low success, 30%), expresses that 30% of the male Great Tits that leave a nesting place with a low occupancy rate, make a dispersion to a nesting place with a low success-rate.

Since the owners of the dataset are interested in significant differences in dispersal behaviour between male and female Great Tits, we combined the restricted confidence tables of the male rule and the female rule to create a new table that only gives those combinations of properties wherefore the difference in confidence between male and female Great Tits is larger than 10%. This table looks as follows:

σ_1	σ_2	conf. male	conf. female
low occupancy	medium occupancy	35%	45%
low occupancy	high occupancy	61%	27%
low occupancy	wood EG	52%	68%
low occupancy	low parasite	70%	45%
low occupancy	medium success	48%	59%
medium occupancy	medium parasite	60%	47%
medium occupancy	high success	17%	29%
high occupancy	low success	10%	22%
low parasite	medium success	63%	50%
low parasite	high success	19%	41%
high parasite	medium occupancy	37%	49%
high parasite	high occupancy	48%	36%
low success	wood EG	53%	66%
low success	medium parasite	58%	37%
low success	high parasite	30%	43%
low success	medium success	67%	54%
high success	medium parasite	64%	53%

The most significant difference between male and female Great Tits is for dispersions leaving a nesting place with a low occupancy rate. We see that 61% of the male Great Tits leaving a nesting place with a low occupancy rate make a dispersion to a nesting place with a high occupancy rate, while this is only 27% for the female Great Tits. Another significant difference exists for dispersions leaving a nesting place with a low parasite rate. We then see that 19% of the male Tits that leave a nesting place with a low parasite rate go to a nesting place with a high success rate, while this is 41% for the female Tits. This could mean that male Tits are affected more by parasites than female Tits.

This table can be useful for the Animal Ecology research group to find new hypotheses about differences in dispersal behaviour between male and female Great Tits.

5.3.4 Conclusion

In this Section we showed how we can use our algorithms to find interesting rules and patterns in real-life data. An important first step is to analyze the dataset thoroughly with the owners and to create a data graph that represents the semantics of the dataset as good as possible. After the tree-query mining algorithm has generated enough results, our browser Certhia is used to search for interesting patterns and to generate interesting rules. We showed that SQL is a very useful tool to query the frequency tables and confidence tables for filtering the results. The found patterns and rules in Section 5.3.3 can be useful for the Animal Ecology research group to analyze their dataset, and to find new hypotheses about dispersal behaviour.

6

Conclusions and Future Work

New applications of data mining, such as in biology, bioinformatics or sociology, are faced with large datasets structured as graphs. In this dissertation we introduced a novel class of tree-shaped patterns called tree queries, and we presented algorithms for mining tree queries and tree-query associations in a large data graph.

Tree Queries In Chapter 2 we introduced tree queries, powerful tree-shaped patterns, inspired by conjunctive database queries [17]. In comparison to the kind of patterns used by most other graph-mining approaches, our patterns can contain constants, and can contain existential nodes which are not counted when determining the frequency. Another important difference with other graph-mining approaches is, that in our setting an occurrence of a pattern in a data graph G , is any homomorphism from the pattern in G . In most other approaches, an occurrence of a pattern in G , is a subgraph isomorphism from the pattern in G . Since the subgraph isomorphism problem is known to be NP-complete [14], other graph-mining approaches try to minimize the number of patterns wherefore the frequency must be computed, by using heuristics to estimate the frequency. Since our patterns are inspired by conjunctive database queries, we benefit fully from results from database theory to compute the frequency of all our patterns exactly and efficiently.

Tree-query-Mining Algorithm In Chapter 3 we presented an algorithm for mining tree queries in a large data graph. Basically the algorithm consists of two loops: an outer loop where we generate trees of increasing sizes, avoiding the generation of isomorphic ones; and an inner loop where we generate for each considered tree, all tree queries based on that tree. The presented algorithm is incremental in the number of nodes of the tree patterns, and we can stop the algorithm when it has run long enough, or when it has produced a sufficiently large pattern database. Note

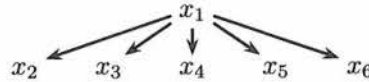
however, that the algorithm is not levelwise in general. As we already mentioned in Section 3.2, due to the way we count the occurrences of a tree pattern, the frequency of a subgraph can be smaller than the frequency of the pattern itself. However we do work in a levelwise fashion for generating all tree queries based on a particular tree.

In Chapter 3 we also defined the important notion of equivalent tree patterns, and we introduced (and proved the correctness of) useful techniques to carefully avoid the generation of equivalent tree patterns. Detecting equivalent tree patterns is very important since it prevents us from performing duplicate work, by avoiding that we compute frequencies that we already know. By using what is known from the theory of conjunctive queries and with our restriction to trees, we can efficiently check if tree patterns are equivalent. This is an improvement in comparison with the Warmr system [11], where equivalence can not be checked efficiently.

A very important feature of the presented tree-query-mining algorithm is, that it suggests a database-oriented implementation. This turned out to be very useful for several reasons: (1) we do not need to move our data graph out of the database before we can start mining; (2) we use the database system to store the huge amount of discovered patterns in a structured manner, in a pattern databases; (3) we use SQL to compute the frequency of a large number of patterns in parallel.

The pattern database is a very useful platform for browsing the found patterns and for generating association rules. Hence, in Chapter 5 we introduced an interactive tool, called Certhia, for browsing the patterns and generating association rules.

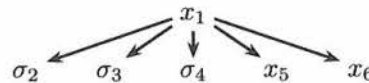
While, using a database system has a lot of advantages, experiments showed that some of the SQL queries performed due to pattern generation take a very long time (in order of hours) to answer by the database system. This happens in those cases where the data graph is large (5000 edges or more) with many cycles, and the candidate patterns are large (6 nodes or more). Consider for instance the following tree pattern:



The frequency of this tree pattern equals $\sum_{x \in G} \deg x^6$, where the sum is over all nodes x in the data graph G . Hence, even for rather small data graphs the frequency of this pattern will be huge. As an illustration, for the data graph in Figure 2.2(a) the frequency is 5020.

The SQL query to compute the frequency of this tree pattern is a heavy join of 5 times the graph table with itself. Clearly, if the graph has a lot of edges, this query will take a long time to compute. If the average degree of the nodes in the graph is high (a dense graph), the situation is even worse.

Consider the same tree pattern where we replaced some distinguished variables by parameters:



To compute the candidacy table of this tree pattern, we have to join its parents frequency tables: $FreqTab_{\emptyset, \{x_2, x_3\}}$, $FreqTab_{\emptyset, \{x_2, x_4\}}$, and $FreqTab_{\emptyset, \{x_3, x_4\}}$, as explained

Bibliography

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In Fayyad et al. [13], pages 307–328.
- [3] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [4] A. Amir, R. Feldman, and R. Kashi. A new and versatile method for association generation. *Information Systems*, 2:333–347, 1997.
- [5] K. Bharat, B.-W. Chang, M. Henzinger, and M. Ruhl. Who links to whom: Mining linkage between Web sites. In N. Cercone, T.Y. Lin, and X. Wu, editors, *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM 2001)*, pages 51–58. IEEE Computer Society Press, 2001.
- [6] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, volume 26:2 of *SIGMOD Record*, pages 255–264. ACM Press, 1997.
- [7] 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.
- [8] 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.
- [9] Y. Chi, Y. Yang, and R.R. Muntz. Canonical forms for labeled trees and their applications in frequent subtree mining. *Knowledge Information Systems*, 8(2):203–234, 2005.
- [10] 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.

- [11] L. Dehaspe and H. Toivonen. Discovery of frequent Datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
- [12] L. Dehaspe and H. Toivonen. Discovery of relational association rules. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, pages 189–212. Springer-Verlag, 2001.
- [13] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. MIT Press, 1996.
- [14] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.
- [15] 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.
- [16] B. Goethals, E. Hoekx, and J. Van den Bussche. Mining tree queries in a graph. In R.L. Grossman, R. Bayardo, K. Bennett, and J. Vaidya, editors, *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 61–69. ACM, 2005.
- [17] B. Goethals and J. Van den Bussche. Relational association rules: getting warmer. In D. Hand, R. Bolton, and N. Adams, editors, *Proceedings of the ESF Exploratory Workshop on Pattern Detection and Discovery in Data Mining*, volume 2447 of *Lecture Notes in Computer Science*, pages 125–139. Springer-Verlag, 2002.
- [18] E. Gudes, S.E. Shimony, and N. Vanetik. Discovering frequent graph patterns using disjoint paths. *IEEE Transactions on Knowledge and Data Engineering*, 18(11):1441–1456, 2006.
- [19] L.M. Haas and A. Tiwary, editors. *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, volume 27:2 of *SIGMOD Record*. ACM Press, 1998.
- [20] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, second edition, 2006.
- [21] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.
- [22] E. Hoekx and J. Van den Bussche. Mining for tree-query associations in a graph. In *Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006)*, pages 254–264. IEEE Computer Society, 2006.
- [23] T. Horváth, J. Ramon, and S. Wrobel. Frequent subgraph mining in outerplanar graphs. In T. Eliassi-Rad, L.H. Ungar, M. Craven, and D. Gunopulos, editors, *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 197–206. ACM, 2006.

- [24] 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.
- [25] A. Inokuchi, T. Washio, and H. Motoda. A general framework for mining frequent subgraphs from labeled graphs. *Fundamenta Informaticae*, 66(1-2):53–82, 2005.
- [26] 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.
- [27] H. Jeong, S.P. Mason, et al. Lethality and centrality in protein networks. *Nature*, 411(3 May 2001).
- [28] R.J. Bayardo Jr. Efficiently mining long patterns from databases. In Haas and Tiwary [19], pages 85–93.
- [29] 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.
- [30] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1038–1051, 2004.
- [31] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11(3):243–271, 2005.
- [32] 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.
- [33] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.
- [34] E. Matthysen, F. Adriaensen, and AA Dhondt. Local recruitment of great and blue tits (*parus major*, *p-caeruleus*) in relation to study plot size and degree of isolation. *Ecography*, 24(1):33–42, 2001.
- [35] B.D. McKay. Nauty User's Guide, version 2.2. <http://cs.anu.edu.au/~bdm/nauty/nug.pdf>.
- [36] B.D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.
- [37] 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.

- [38] M. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.
- [39] M. De Santo, P. Foggia, C. Sansone, and M. Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8):1067–1079, 2003.
- [40] 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.
- [41] H.I. Scions. Placing trees in lexicographic order. In D. Michie, editor, *Machine Intelligence 3*, pages 43–62. Edinburgh University Press, 1968.
- [42] W.-M. Shen, K. Ong, B.G. Mitbender, and C. Zaniolo. Metaqueries for data mining. In Fayyad et al. [13], pages 375–398.
- [43] S. Tsur, J.D. Ullman, et al. Query flocks: A generalization of association-rule mining. In Haas and Tiwary [19], pages 1–12.
- [44] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume II. Computer Science Press, 1989.
- [45] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 721–724. IEEE Computer Society Press, 2002.
- [46] M.J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering*, 17(8):1021–1035, 2005.

Notations

Notation	Interpretation
U	set of data constants
T	ordered rooted tree
G	data graph
P	parameterized tree pattern
Π	set of existential nodes
Δ	set of distinguished nodes
Σ	set of parameters
\exists	existential node
σ	parameter
x	distinguished node
α	parameter assignment
$P^\alpha, (P, \alpha)$	instantiated tree pattern
$P^\alpha(G)$	$\{\mu _\Delta : \mu \text{ is a matching of } P^\alpha \text{ in } G\}$
minsup	the frequency threshold
$Q = (H, P)$	parameterized tree query with H the head and P the body
$Q^\alpha, (Q, \alpha)$	instantiated tree query
$Q^\alpha(G)$	answer set of the instantiated tree query Q^α in G
ρ	parameter correspondence
$Q_2 \subseteq_\rho Q_1$	Q_2 is ρ -contained in Q_1
$\text{freeze}_\beta(P)$	the freezing of a tree pattern P
pAR	parameterized association rule
iAR	instantiated association rule
$Q_1 \Rightarrow_\rho Q_2$	pAR from Q_1 to Q_2
$(Q_1 \Rightarrow_\rho Q_2, \alpha)$	iAR from Q_1 to Q_2
minconf	the confidence threshold
$\text{Freq}(P^\alpha)$	the frequency of P^α in G if G is understood
(Π, Σ)	a parameterized tree pattern P based on a fixed tree T
(Π, Σ, α)	an instantiated tree pattern P based on a fixed tree T
$\text{CanTab}_{\Pi, \Sigma}$	$\{\alpha \mid P^\alpha \text{ is a candidate instantiated tree pattern}\}$
$\text{FreqTab}_{\Pi, \Sigma}$	$\{\alpha \mid P^\alpha \text{ is a frequent instantiated tree pattern}\}$
δ	answer set correspondence
$P_1 \equiv_\rho^\delta P_2$	P_1 is (δ, ρ) -equivalent with P_2

$P_2^{\alpha_2}(G) \circ \delta$	$\{f \circ \delta : f \in P_2^{\alpha_2}(G)\}$
$P_1 \cong P_2$	P_1 and P_2 are isomorphic

Samenvatting

Data mining is een nieuw onderzoeksgebied dat de voorbije jaren veel aandacht heeft gekregen. Een gekend handboek [21] over data mining motiveert het ontstaan van dit nieuwe onderzoeksgebied als volgt:

Grote technologische vooruitgang in het geautomatiseerd verzamelen en opslaan van gegevens heeft als gevolg dat er gigantische verzamelingen gegevens ontstaan en blijven groeien. Deze verzamelingen kunnen veel verschillende soorten gegevens bevatten: transactiegegevens van supermarkten; overheidsstatistieken; gedetailleerde gegevens van telefoongesprekken; gegevens over het gebruik van kredietkaarten; beelden van hemellichamen; moleculaire databases en medische databases. Geen wonder dat de interesse groeide om bruikbare en handelbare informatie uit deze gegevens te halen zodanig dat deze beter geanalyseerd kunnen worden.

Data mining houdt zich bezig met het geautomatiseerd verwerken van grote hoeveelheden gegevens tot bruikbare en handelbare informatie, die gebruikt kan worden door de eigenaars van de gegevens om deze beter te analyseren.

In het begin werd data mining vooral toegepast op eerder eenvoudige verzamelingen gegevens zoals bijvoorbeeld transactiegegevens van supermarkten. Recent is de interesse gegroeid om data mining toe te passen op meer complexe gegevens zoals gegevensstromen, grafen, bomen en XML-bestanden.

In deze thesis concentreren we ons op gegevens die voorgesteld worden als een graaf. Grafen worden steeds belangrijker voor het modelleren van ingewikkelde structuren zoals elektrische netwerken, beelden, chemische componenten, proteïne structuren, biologische netwerken, sociale netwerken, het World Wide Web, workflows en XML-bestanden. Vermits er een groeiende vraag is naar het analyseren van grote hoeveelheden gegevens voorgesteld als een graaf, is *Graph Mining* een actief en belangrijk thema binnen data mining geworden.

Van alle verschillende soorten graafpatronen, zijn *frequente deelstructuren* de meest eenvoudige patronen die ontdekt kunnen worden in een graaf of een collectie van grafen. Frequentie deelstructuren zijn nuttig om verzamelingen grafen te onderscheiden van elkaar; grafen in klassen op te delen; grafen te clusteren en om de zoektocht naar gelijkaardige grafen in een collectie van grafen te vergemakkelijken.

Er zijn veel verschillende thema's binnen graph mining zoals het clusteren van grafen; het opdelen van grafen in klassen en het zoeken naar vaak voorkomende deel-

grafen, maar in deze thesis concentreren we ons op het zoeken naar boompatronen en associatieregels over deze patronen in een graaf.

Het probleem van het ontginnen van patronen in gegevens voorgesteld als een graaf heeft veel aandacht gekregen de voorbije jaren, omdat het veel interessante toepassingen heeft in verschillende gebieden zoals biologie, de levenswetenschappen, het World Wide Web of de sociale wetenschappen. In deze thesis stellen we een nieuwe soort van patronen voor, die we boomqueries noemen en associatieregels over deze boomqueries. We geven ook algoritmes voor het ontginnen van boomqueries en associatieregels over boomqueries in een grote ongelabelde gerichte graaf.

De meeste resultaten uit deze thesis werden gepresenteerd op twee conferenties [16, 22].

Boomqueries zijn krachtige boompatronen die geïnspireerd zijn door conjunctive database queries [17]. In vergelijking met de patronen die gebruikt worden in andere graph mining benaderingen, hebben onze patronen enkele speciale kenmerken:

- De patronen kunnen “existentiële” knopen bevatten: elk voorkomen van het patroon moet een kopie hebben van dergelijke knoop, maar existentiële knopen worden niet geteld als we het aantal voorkomens van een patroon bepalen.
- De patronen kunnen ook “geparameteriseerde” knopen, gelabeld met een constante (knoop identifier), bevatten: deze knopen moeten afgebeeld worden op specifieke knopen van de gegevensgraaf.
- Een “voorkomen” van een patroon in een gegevensgraaf G is gedefinieerd als een homomorfisme van het patroon in G . Als we het aantal voorkomens van een patroon bepalen, zorgen we ervoor dat voorkomens die enkel verschillen in de afbeelding van de existentiële knopen slechts één keer geteld worden.

In vroeger werk over graph mining werden gelabelde knopen reeds beschouwd, maar dan alleen met niet-unieke labels. We tonen aan dat niet-unieke labels gemakkelijk gesimuleerd kunnen worden door unieke labels (constanten), maar het is niet duidelijk hoe constanten gesimuleerd kunnen worden door niet-unieke labels.

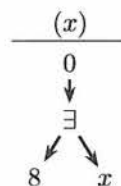


Figure 6.1: Een voorbeeld van een boomquery.

In Figuur 6.1 wordt een eenvoudig voorbeeld van een boomquery gegeven. Als we deze boomquery toepassen op een voedselnetwerk, een gegevensgraaf van organismen, waar er een pijl $x \rightarrow y$ is, als y zich voedt met x , dan beschrijft deze boomquery alle organismen x , die strijden met organisme #8 om zich te voeden met een organisme, dat zich wederom voedt met organisme #0. Dit patroon bevat één existentiële knoop,

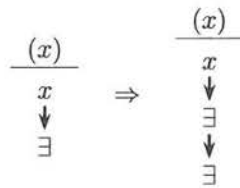


Figure 6.2: Voorbeeld van een associatieregels over boomqueries.

twee parameters, en één knoop waarvan we het aantal onderscheidbare voorkomens gaan tellen.

In feite zijn boomqueries wat we kennen uit database onderzoek als *conjunctive queries* [8, 44, 1]: dit zijn queries die we kunnen stellen aan de gegevensgraaf (opge-slaan als een tabel met twee kolommen) door alleen maar gebruik te maken van het kernfragment van SQL, waar we geen samengestelde of deelqueries gebruiken, maar enkel conjuncties van gelijkheden in de where-voorwaarden. Bijvoorbeeld, voor het patroon uit Figuur 6.1 zal de SQL-query op een tabel $G(\text{from}, \text{to})$ er als volgt uitzien:

```

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
    
```

In deze thesis introduceren we ook associatieregels over boomqueries. Door het zoeken naar boomquery-associaties kunnen we subtiële kenmerken van de gegevens-graaf ontdekken. Beschouw bijvoorbeeld de eenvoudige associatieregels in Figuur 6.2 die we in een voedselnetwerk kunnen ontdekken. Als deze associatieregels een betrouw-baarheid c heeft, wil dit zeggen dat van alle organismen die niet aan de top van de voedselketen staan, er een fractie c minstens op diepte twee in de voedselketen zitten.

De voorbeelden van een boomquery en een boomquery-associatie waren enkel di-dactische voorbeelden. In deze thesis worden er nog andere, meer ingewikkelde voor-beelden gegeven, onder andere boomqueries en boomquery-associaties die we ontdekt hebben in gegevens uit Ecologie.

In deze thesis geven we algoritmes voor het ontginnen van boomqueries en boom-query-associaties in een grote gegevensgraaf. De algoritmes die we voorstellen, hebben de volgende belangrijke eigenschappen:

1. Onze algoritmes behoren tot de groep van graph-mining algoritmes waar de input één enkele grote graaf is, en de opdracht er in bestaat om patronen te ontdekken die vaak genoeg voorkomen in deze grote graaf. Deze groep van graph-mining algoritmes noemt men de single-graaf categorie. Er is ook een tweede categorie van graph-mining algoritmes, waar de input bestaat uit een collectie van grafen, en de opdracht er in bestaat om patronen te ontdekken die minstens één keer voorkomen in een voldoende aantal grafen. Deze categorie noemt men de transactie categorie.
2. We beperken ons tot boompatronen. Boompatronen zijn reeds uitvoerig be-studeerd in de transactie categorie, maar ze hebben nog geen speciale aandacht

gekregen in de single-graaf categorie. Merk wel op dat we op geen enkele manier restricties leggen op onze gegevensgraaf.

3. Het algoritme voor het ontginnen van boomqueries is incrementeel in het aantal knopen van de boom. Met andere woorden, ons algoritme beschouwt systematisch grotere bomen, en kan gestopt worden op elk tijdstip waarop we vinden dat het lang genoeg gelopen heeft, of als het voldoende resultaten opgeleverd heeft. Buiten de opslagruimte nodig voor de gevonden patronen, heeft ons algoritme geen opslagruimte nodig. Door te beperking tot bomen kunnen we efficiënt bomen genereren zonder duplicaten.
4. Voor elke boom, genereren we alle conjunctive queries gebaseerd op die boom levelwise [33].
5. Net zoals in het klassieke data mining probleem waar er gezocht wordt naar associatieregels over verzamelingen items, worden onze associatieregels gegenereerd na het genereren van de vaak voorkomende patronen. Er is er geen toegang tot de originele gegevens meer nodig en we maken enkel gebruik van de reeds gevonden patronen.
6. We gebruiken de theorie over conjunctive queries [8, 44, 1] om associatieregels over boomqueries formeel te definiëren op een correcte manier, en om ze correct te genereren. De conjunctive-query aanpak voor het zoeken van voorkomens van een patroon in de graaf, laat ons toe om efficiënt het aantal voorkomens van een patroon in een gegevensgraaf te bepalen. In benaderingen gebaseerd op deelgrafen is het bepalen van het aantal voorkomens van een patroon in een gegevensgraaf een NP-compleet probleem.
7. We hebben een notie van equivalentie voor boomqueries en boomquery-associatieregels. Door gebruik te maken van resultaten uit de theorie over conjunctive queries, kunnen we efficiënt en nauwkeurig het genereren van equivalente boomqueries en boomquery-associatieregels vermijden. Door onze beperking tot bomen, kunnen we het bestaan van equivalenties en redundanties efficiënt controleren.
8. Een laatste, maar zeker niet minder belangrijke eigenschap van onze algoritmes is dat ze heel natuurlijk een database-georiënteerde implementatie in SQL suggereren. Dit is nuttig om verschillende redenen: (1) Het aantal ontdekte patronen kan best wel groot zijn. Het is belangrijk om al deze patronen beschikbaar te houden op een consistente en gestructureerde manier, zodanig dat ze gemakkelijk doorzoekbaar zijn, en gebruikt kunnen worden voor het genereren van associatieregels. (2) We kunnen SQL gebruiken om het aantal voorkomens te tellen van een groot aantal patronen in parallel. We doen dit door gebruik te maken van de query-optimalisaties waarover moderne relationele databasesystemen beschikken. (3) Het is niet meer nodig om onze gegevensgraaf uit de database te halen alvorens we kunnen beginnen met het ontginnen van patronen. In het klassieke data mining probleem, waar er gezocht wordt naar vaak voorkomende verzamelingen items, werd er reeds veel aandacht besteed aan

database-georiënteerde implementaties [43, 40], maar in graph mining werd er nog niet veel aandacht aan besteed, buiten een recente uitzondering waar er een implementatie van het SUBDUE algoritme in SQL werd voorgesteld [7].

Ten slotte geven we in deze thesis ook resultaten van experimenten die we hebben uitgevoerd met de implementaties van onze algoritmes. We tonen dat beide algoritmes voldoende performant zijn, en we tonen hoe ze toegepast kunnen worden op wetenschappelijke gegevens om interessante patronen en associaties te ontdekken.

