

**A GENERIC LIBRARY FOR
FREQUENT PATTERN MINING**

By

Nagender Parimi

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Approved:

Prof. Mohammed J. Zaki
Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

April 2005
(For Graduation May 2005)

CONTENTS

LIST OF TABLES	iv
LIST OF FIGURES	v
ACKNOWLEDGMENT	vi
ABSTRACT	vii
1. INTRODUCTION	1
1.1 Data Mining Techniques	1
1.2 Generic Programming	3
2. RELATED WORK & MOTIVATION	4
2.1 Definitions	4
2.2 Itemset Mining	5
2.3 Sequence Mining	6
2.4 Tree Mining	7
2.5 Graph Mining	8
2.6 Vertical Mining	9
2.7 Motivation	10
3. PROPERTY FRAMEWORK	13
3.1 Pattern Properties from Pattern Relations	13
3.2 Property Hierarchy	15
3.3 Mining Properties	16
4. DESIGN PRINCIPLES OF THE IMPLEMENTATION	19
4.1 Mining Methodology	19
4.2 Template based Approach	22
4.2.1 Generic Containers	22
4.2.2 Specializations	22
4.2.3 Static Type List	23
4.3 Other Features	25

5. DMTL ARCHITECTURE	27
5.1 Generic Components	27
5.1.1 Reading in the Database	27
5.1.2 The Pattern Class	27
5.1.3 Determining Support of a Pattern	28
5.1.4 Support of a Pattern	29
5.1.5 Storage Manager	30
5.2 Specialized Components	31
5.2.1 Parsing a Transaction	31
5.2.2 Generating Candidates	31
5.2.3 Vertical Attribute Lists (VATs)	32
5.3 Graph VAT	34
6. EXPERIMENTAL STUDY	39
6.1 Experimental Setup	39
6.2 Itemset & Sequence Mining	40
6.3 Tree Mining	41
6.4 Level-1	45
7. CONCLUSIONS AND FUTURE WORK	46
7.1 Conclusions	46
7.2 Future Work	46
BIBLIOGRAPHY	47

LIST OF TABLES

6.1	Effect of varying database sizes on number of frequent itemsets	41
6.2	Effect of varying minsup on number of frequent itemsets	42
6.3	Fraction of mining time spent in level-1 with varying DB size for induced, unordered trees	45

LIST OF FIGURES

2.1	FPM Instances	12
3.1	Pattern Property Hierarchy	17
4.1	Pattern Mining Algorithm	20
4.2	A Static Type List	23
4.3	Application of Property Hierarchy	24
5.1	Pattern Class Interface	29
5.2	Storage Manager Interface	30
5.3	Sample Graph Database & Level-1 VATs	35
5.4	Graph VAT Intersection	37
6.1	Effect of varying database size on performance of itemset mining	40
6.2	Effect of varying minsup on performance of itemset mining	41
6.3	Effect of varying database size on performance of induced sequence mining	42
6.4	Comparison of DMTL with SLEUTH for mining induced, unordered trees over varying database sizes	43
6.5	Comparison of DMTL with SLEUTH for mining induced, unordered trees for varying minsup	43
6.6	Comparison of DMTL with SLEUTH for mining embedded, ordered trees over varying database sizes	44
6.7	Comparison of DMTL with SLEUTH for mining induced, unordered trees for varying minsup	44

ACKNOWLEDGMENT

I would like to thank my advisor, Prof. Mohammed J. Zaki, for the guidance he provided during this work. Working on DMTL was a great learning experience and I am grateful to him for giving me this opportunity. The project is a cumulative work of several students, and my work often relied on that of others. I thank all students in Prof. Zaki's data mining group for their endeavors. A special thanks to Prof. David Musser for several insightful discussions on the use of generic programming in our context.

I am also grateful to Dr. Frank Xiao, Dr. Raj Minhas and Jean Ellefson (Wilson Center for Research & Technology) and members of the data mining project at the Imaging Services & Technology Center, both a part of Xerox Corp., Rochester, NY, where I spent a very enjoyable and enriching summer. I thank David Bauer and the CS labstaff for providing valuable resources and equipment utilized during the course of this work.

Finally, I'd like to thank my parents and my brother for their constant encouragement and support.

ABSTRACT

Frequent Pattern Mining (FPM) is one of the main classes of problems in data mining. The goal of FPM is to discover oft-occurring structures (patterns) from a database of such structures. Frequently occurring patterns are deemed interesting and hence their enumeration often leads to insights into the domain under consideration. FPM problems can be categorized based on the type of pattern being mined for; the common pattern types are itemsets, sequences, trees and graphs. FPM has been successfully applied to diverse domains such as market basket analysis, web mining and bio-informatics. One of the key challenges of data mining algorithms in general is to find efficient solutions which can scale to massive datasets.

This thesis was part of the Data Mining Template Library (DMTL) project under Prof. Zaki. Our goal under this project was to develop a library of data structures and algorithms, providing a systematic solution to the class of FPM problems. The toolkit was designed so it may be extended to new pattern types and database formats. We introduce the concept of generic pattern mining, which is termed as the genericity of data structures and algorithms for FPM. We believe that generic pattern mining can help understand the space of patterns and their mining algorithms, and hence this project seeks to motivate research in this field. Other key design features include memory management and persistency of the results. The project is currently under progress. The contribution of this thesis in particular was the design of generic data structures and implementation of the mining algorithms for itemset, sequence, tree and graph patterns. These algorithms interact with the back-end database systems to produce the desired results. We also present comparisons of the toolkit with stand-alone FPM implementations.

CHAPTER 1

INTRODUCTION

Data mining is the science of discovering hidden and useful knowledge from large databases. It is synonymously known as Knowledge Discovery and Data mining (KDD). It draws its roots from machine learning but exists today at the confluence of machine learning, statistics and databases. Its focus is the extraction of actionable information, hence it finds direct applications in very many real world problems. This makes it an interesting field, at the same time posing several challenges. Such real world problems demand action-able and relevant results, and require efficient and scalable algorithms.

Data mining has emerged as a rapidly growing research field over the last decade. In part, this has been motivated by several concurrent developments - the prolific growth of the Internet, ease of availability of computational resources as well as inexpensive media for storage of massive datasets (often up to terabytes of data). Data mining techniques are now popularly used for applications ranging from marketing strategies and bio- informatics to scientific research.

1.1 Data Mining Techniques

Data mining is usually broken up into three major classes of problems:

- Classification (Supervised Learning). Given a dataset of attributes (one of them identified as the *class label*) and several records containing values for each attribute, the goal of classification is to learn the correlation between the class label and all remaining attributes [15]. It is termed supervised learning since the algorithm's learning is guided by the class label. Once the system has learnt the input data, it is deployed to predict the class label on new, unseen data points which do not have prior class labels associated with them. Prediction of this form is implicitly based on the assumption that all data points are taken from the same distribution. Prediction for numerical class labels is called *regression*, while classification deals with nominal classes. However,

classification is often used to refer to the set of supervised learning schemes. Some of the common classification techniques are decision trees, nearest neighbor learning, Bayesian classifiers and support vector machines [13]. Recent advances in this field include bagging and boosting [7]. Classification systems have to contend with problems such as noisy and imbalanced data and missing values. Nevertheless, classifiers have been successfully applied for outlier detection and customer segmentation.

- Clustering (Unsupervised Learning). Clustering seeks to identify groups (clusters) of similar data points in a multidimensional dataset [19]. Since clustering does not have access to a class label to be guided by, this is known as unsupervised learning. Clustering systems use a distance based metric to measure dissimilarity between two data points. The goal (objective function) is to maximize distance between any two points belonging to distinct clusters, simultaneously minimizing distance between points in the same cluster. Several clustering techniques have been proposed, for numerical as well as nominal valued datasets. Problems faced by classifiers also plague clustering schemes. Dealing with huge datasets (high dimensional as well as large sized) is often a drawback with clustering systems; sub-space clustering has attracted attention recently in order to address this issue.
- Frequent Pattern Mining (FPM). Frequent pattern mining entails discovering oft-occurring structures from a database of such structures. Work in this field originated with mining of association rules [3] (also known as itemset mining). The first widely accepted efficient solution was the *Apriori* algorithm [4]. Apriori discovers all collections of items (called itemsets) that occur frequently in the given dataset, and can then enumerate correlations/rules between the frequent itemsets. Apriori was immensely successful for determining sets of frequently bought products at stores, and thus is also known as a method for *market basket analysis*. In itemset mining, it was evident that discovery of frequent itemsets was the bottleneck since generating rules from them was a fast process. There have been several enhancements to frequent itemset dis-

covery since, and we shall discuss some of the prominent ones later. Efficient schemes for discovery of more complex patterns have been proposed recently, these are detailed in Chapter 2.

1.2 Generic Programming

We use the concepts of generic programming extensively in the C++ based implementation of our library. Generic programming was introduced into C++ with support for templates and the Standard Template Library (STL) [2]. Templates facilitate generic programming, which is the capability to write code that works not just for a single type, rather for several types. Such a function or class is said to be templated in C++ jargon. The STL provides implementation for many commonly used data structures and algorithms. We do not use templates to define truly generic implementations as STL does, nonetheless understanding the template mechanism is essential in order to use the toolkit optimally.

Templates provide a means of compile time or static polymorphism. This causes lesser overhead than the typical runtime polymorphism in C++, but may lead to the problem of *code bloat* [24]. Templates play a vital role in providing means of extending our toolkit for new pattern types.

CHAPTER 2

RELATED WORK & MOTIVATION

Frequent Pattern Mining has witnessed significant activity in the last decade. The space of FPM problems varies with the definition of the pattern under consideration. Itemsets are the most primitive form of patterns, and also the easiest to mine. Sequences, trees and graphs are progressively more complex, and hence their mining algorithms are also progressively more challenging. There exist mining algorithms for each of these four pattern types, and we discuss them below. We focus on these four pattern types primarily because they span the entire spectrum of possible patterns, and also because their real world applications motivate strong interest in finding efficient solutions.

2.1 Definitions

We refer to the different kinds of patterns as pattern types, i.e. itemset, sequence, tree and graph are a few pattern types. Pattern refers to a single instance of any pattern type. A database D comprises several patterns of the same kind, \mathcal{P} . Each pattern in the database is identified with a unique transaction identifier, or tid. The tid for a pattern p is denoted as $tid(p)$. The *support* or *frequency* of a pattern, denoted as $\sigma(p)$ is the number of distinct transactions it occurs in. We say p occurs within \hat{p} iff p is a *subpattern* of \hat{p} , denoted as $p \preceq \hat{p}$. The subpattern relation is defined by \mathcal{P} . Let $\delta_{\hat{p}}$ denote the number of occurrences (defined by \mathcal{P}) of p in a pattern \hat{p} . Let d_p be an indicator variable, with $d_{\hat{p}}(p) = 1$ if $\delta_{\hat{p}}(p) > 0$ and $d_{\hat{p}}(p) = 0$ if $\delta_{\hat{p}}(p) = 0$. Then, $\sigma(p) = \sum_{\hat{p} \in D} d_{\hat{p}}(p)$. A pattern p is said to be *frequent* if $\sigma(p) \geq minsup$, where *minsup* is a user defined threshold criterion. *Relative support* is defined as $\frac{\sigma(p)}{|D|}$.

We can define $p_x \succeq p_y$ where the superpattern relation (\succeq) is defined by \mathcal{P} . The superpattern relation can be inferred from the subpattern relation, and is simply its inverse. A frequent pattern is said to be *closed* if there is no frequent superpattern with the same support. A *maximal* pattern is one which possesses

no frequent superpattern at all. For a given dataset, let \mathcal{F} be the set of frequent patterns, \mathcal{C} the set of closed patterns and \mathcal{M} the set of maximal patterns, then in general $|\mathcal{F}| \gg |\mathcal{C}| \gg |\mathcal{M}|$. Furthermore, the set of closed patterns is a lossless representation of all frequent patterns, i.e. the support of all frequent patterns can be unambiguously determined from \mathcal{C} [38].

Association rules introduced the concept of rules of correlation between frequent patterns. If p_x and p_y be frequent patterns, we can find correlation between them as follows: $p_x \Rightarrow p_y$. The *confidence* of such a rule is defined as $\frac{\sigma(p_x \cap p_y)}{\sigma(p_x)}$. Intuitively, the confidence represents the fraction of transactions containing p_x that contains p_y as well. Such rules have been explored primarily for itemsets [4]. Our work is centered around the discovery of frequent patterns. In the following sections, we define the size or length of a pattern. Mining algorithms in use today are incremental, i.e., they perform several iterations to discover all frequent patterns. Each iteration is also termed a *level*, and k -sized frequent patterns are enumerated during the k th iteration or level- k .

2.2 Itemset Mining

Itemsets denote co-occurrence of items. The itemset FPM was proposed in [3]. Several advanced algorithms followed, including Apriori [4], FP-Tree [16] and ECLAT [32]. Formally, if \mathcal{L} be the set of labels (items), we define an *itemset* I as a subset of \mathcal{L} . The size or length of I is simply the number of items it contains, a k -sized itemset is any itemset whose size is k . An itemset database comprises several itemsets (of varying sizes), with a unique *tid* attached to each. Given such a dataset D and *minsup*, the itemset FPM task is to enumerate all frequent itemsets. For itemsets, the subpattern relation is simply the subset relation over \mathcal{L} e.g. omitting set notation for brevity, $ABC \prec ABCD$. Without loss of generality, items in a itemset are assumed to be sorted lexicographically.

The cardinality of set of possible itemsets is exponential in the size of \mathcal{L} . Hence, an intelligent search through the space of possible itemsets is needed. Apriori does a level wise growth of the set of frequent patterns: from frequent k -sized itemsets it discovers frequent $(k + 1)$ -sized itemsets. Based on the k -sized frequent itemsets,

Apriori generates a set of *candidate* itemsets, which may potentially be frequent. The support of each candidate is then ascertained, to determine which of them is frequent. A k -sized itemset (k -*itemset*) belongs to the partition \mathcal{F}_k . The FP-Tree is an alternative approach which maintains a data structure called frequent pattern tree, enabling it to discover frequent itemsets without enumerating any infrequent candidates. However, for large datasets the FP-Tree may longer fit in memory.

ECLAT seeks to reduce I/O overhead incurred during support counting of candidates in Apriori. The basic idea is to associate with each itemset, the list of *tids* it occurs in. We term this data structure a *Vertical Attribute Table (VAT)*. The VAT structure enables us to determine the support of a candidate from the VATs of smaller sized frequent patterns. We term this approach *vertical mining*. We refer the reader to Section 2.6 for a discussion on the usage and efficacy of VATs. In [37] the authors propose ways of further speeding up the support counting step.

2.3 Sequence Mining

A sequence denotes a total ordering among its elements, possibly in addition to the co-occurrence relation of itemsets. An *event* is a non-empty unordered set of items from \mathcal{L} (we assume items of an event are sorted lexicographically without loss of generality). A *sequence* is an ordered list of events. An event is denoted as $(i_1 i_2 \dots i_k)$ where i_j is an item. A sequence S is denoted as $(S_1 \rightarrow S_2 \rightarrow \dots \rightarrow S_m)$ where S_j is an event. A k -*sequence* is a sequence with k items ($k = \sum_j |S_j|$). For a sequence S , if the event S_i occurs before S_j we denote it as $S_i <_k S_j$ where k is the number of events that separate them. When unspecified, it should be assumed that k can take any appropriate value.

The subpattern relation for sequences depends on the kind of sequences being mined: induced or embedded. For sequences S and \hat{S} , S is an embedded subpattern of \hat{S} , denoted as $S \preceq_e \hat{S}$ iff there exists a one-to-one order preserving function f that maps events in S to events in \hat{S} , i.e. i) $S_i \subseteq f(S_i)$, and ii) if $S_i < S_j$ then $f(S_i) < f(S_j)$. S is an induced subpattern of \hat{S} , $S \preceq_i \hat{S}$ iff the second condition above is modified to: ii) if $S_i <_1 S_j$ then $f(S_i) <_1 f(S_j)$. As an example, $A \rightarrow D$ is an embedded subpattern of $A \rightarrow BC \rightarrow D$, but not an induced one. For S to be

an induced subpattern of \hat{S} , consecutive events in S must also be consecutive in \hat{S} .

The sequence FPM was introduced in [5]; the same authors proposed an improved algorithm in [28]. Several enhancements have been presented since, including SPADE [34]. SPADE extends the notion of vertical mining and VATs for sequences, and was observed to outperform most other techniques for large databases [34]. The above work focused on mining embedded sequences, an approach for discovering induced sequences was given in [33].

2.4 Tree Mining

A rooted, labeled tree $T = (V, E)$ is a directed, acyclic, connected graph with $V = 0, 1, \dots, n$ as the set of vertices and $E = (x, y) | x, y \in V$ as the set of edges. One vertex is distinguished as the root, such that there exists a path from the root to all other vertices. Furthermore, $l : V \rightarrow \mathcal{L}$ is the labeling function. In an *ordered* tree, the child nodes of each vertex possess an ordering else the tree is said to be *unordered*. If $x, y \in V$ and there is a path from x to y then x is called an ancestor of y (and y a descendant of x), denoted as $x \leq_p y$ where p is length of path from x to y . If $x \leq_1 y$ then x is the parent of y , and y its child.

Induced and embedded subpattern relations for trees are defined as follows [36]. Given two trees $T = (V_T, E_T)$ and $\hat{T} = (V_{\hat{T}}, E_{\hat{T}})$ we say that T is an *isomorphic subtree* of \hat{T} iff there exists a one-to-one mapping $\varphi : V_T \rightarrow V_{\hat{T}}$ such that $(x, y) \in E_T$ iff $(\varphi(x), \varphi(y)) \in E_{\hat{T}}$. T is an induced subpattern of \hat{T} , $T \preceq_i \hat{T}$ iff T is an isomorphic subtree of \hat{T} and φ preserves labels, i.e. $l_T(x) = l_{\hat{T}}(\varphi(x)), \forall x \in V_T$. That is, for induced subtrees φ preserves parent-child relationships as well as vertex labels. $T = (V_T, E_T)$ is an embedded subpattern of $\hat{T} = (V_{\hat{T}}, E_{\hat{T}})$, $T \preceq_e \hat{T}$ iff there exists a one-to-one mapping $\varphi : V_T \rightarrow V_{\hat{T}}$ that satisfies i) $(x, y) \in E_T$ iff $(\varphi(x) <_p \varphi(y))$, and ii) $l(x) = l(\varphi(x)), \forall x \in V_T$.

Tree mining algorithms were proposed in [35], [6]. These methods were applicable for embedded, ordered trees. FreeTreeMiner [10] mines induced, unordered free trees, which are undirected, acyclic graph with no distinct root. The same authors devised HybridTreeMiner [11] for mining closed and maximal free trees. uFreqT [25] discovers all unordered induced trees, while SLEUTH [36] provides a

unified scheme for mining ordered/unordered induced/embedded trees. TreeMinerV and SLEUTH present a vertical mining approach for trees and provide definitions for a tree VAT. Henceforth, in this document we shall use the term trees to imply directed, acyclic, rooted graphs; we shall refer to free trees explicitly as needed. DMTL can mine induced/embedded and ordered/unordered trees.

2.5 Graph Mining

Graph mining is the most challenging due to the inherent complexity of graphical structures. A graph $G = (V, E)$ possesses a set of vertices and edges, and a labeling function $l : V \cup E \rightarrow \mathcal{L}$. The size of a graph is the number of edges it contains. Graphs may be directed or undirected; DMTL can currently mine undirected graphs only. However, it is not difficult to extend an undirected graph mining scheme for digraphs. Here on, when referring to graph mining we assume undirected graphs.

A graph $G = (V_G, E_G)$ is isomorphic to $\hat{G} = (V_{\hat{G}}, E_{\hat{G}})$ iff there exists a bijective function $\varphi : V_G \rightarrow V_{\hat{G}}$ such that i) $l_G(x) = l_{\hat{G}}(\varphi(x)) \forall x \in V_G$, and ii) $l_G(x, y) = l_{\hat{G}}(\varphi(x), \varphi(y))$ and $(\varphi(x), \varphi(y)) \in E_{\hat{G}} \forall (x, y) \in E_G$. G is a subpattern of \hat{G} , $G \preceq \hat{G}$ iff G is isomorphic to a subgraph of \hat{G} . A variant of the above is induced graph mining: $G \preceq_i \hat{G}$ iff G is isomorphic to a subgraph of \hat{G} and in addition G contains all edges of \hat{G} connecting nodes in V_G . As indicated in [17], graph mining is particularly challenging because of i) subgraph isomorphism which is an NP-complete problem [26], and ii) problem of efficiently enumerating all frequent subgraphs.

AGM [18] was the first algorithm for graph mining, and was largely based on Apriori. FSG [23] significantly improved on the running time of AGM by assigning a canonical labeling to graphs. Recently, gSpan [30] and FFSM [17] have reduced the running time further by offering better means of candidate generation and isomorphism checking. gSpan proposed a Depth First Search based canonical ordering, while FFSM introduced Canonical Adjacency Matrix (CAM) to deal with graph isomorphism. However, the question of subgraph isomorphism, i.e., determining support of a given graph, is still a bottleneck. We propose a novel vertical

mining scheme for graphs, which seeks to circumvent this NP-complete problem by storing additional information in the form of VATs.

2.6 Vertical Mining

Vertical mining refers to the scheme of support counting used by DMTL. It draws roots from the schemes proposed in [32], [34] and [35]. The VAT design is similar to that of Binary Attribute Tables in the Monet database system [8]. To provide native database support for objects in the vertical format, DMTL adopts a fine grained data model, where records are stored as *Vertical Attribute Tables* (VATs). Given a database of objects, where each object is characterized by a set of attributes, a VAT is essentially the collection of objects that share the same values for the attributes. For example, for a relational table, `cars`, with the two attributes, `color` and `brand`, a VAT for the attribute-value `color=red` stores all the transaction identifiers of cars whose color is red. The main advantage of VAT is that they allow for optimizations of query intensive applications like data mining where only a subset of the attributes need to be processed during each query.

The basic theme behind vertical mining is to provide an intelligent means of determining the support of a pattern. Typically, the support is determined by querying the input database. This process is I/O bound as database size grows. The vertical mining approach associates a data structure, called Vertical Attribute Table (VAT), with each pattern. The VAT of a pattern p is referred to as $vat(p)$. The purpose of the VAT is to uniquely identify each occurrence of its pattern. Hence, $\sigma(p) = |vat(p)|$. The structure of the VAT for each pattern type is different, and needs to be designed distinctly. The following lemma motivates the use of VATs for fast support computation.

Lemma 1. *Let p , p_x and p_y be patterns such that $p_x \prec p$, $p_y \prec p$, with $p_x \neq p_y$. In addition, we require that the join of p_x and p_y be p , $p = p_x \odot p_y$. Then, we can define an intersection operation on VATs such that $vat(p) = vat(p_x) \otimes vat(p_y)$. The join and intersection operations are dependent on the pattern type, \mathcal{P} .*

Proof. The join operation (\odot) gives us a means of constructing p from unification

of p_x and p_y . Hence, each occurrence of p must be captured by either $vat(p_x)$ or $vat(p_y)$. Then, by construction, we can determine $vat(p)$ from $vat(p_x)$ and $vat(p_y)$. It is in order to achieve this that we define the intersection operation (\otimes). It gathers the common occurrences of p from $vat(p_x)$ and $vat(p_y)$. The join and intersection operations must be suitably defined for the pattern type under consideration. \square

We exploit this lemma while employing vertical mining in the toolkit. We chose vertical mining over traditional approaches of counting support since it had been shown to be more efficient in the existing literature. Conventional means of counting support of a pattern p entail querying if p is contained in p_T , for every transaction p_T , $p \preceq p_T$ in the database. We term this approach *horizontal mining* since it involves subpattern checks for every horizontal record (transaction). Optimizations on this approach are possible and have been suggested, but the underlying approach stays the same. It has been previously shown [32, 34] that vertical mining outperforms horizontal mining when the number of patterns being counted is much smaller than the number of transactions in the database. However, in the initial stages of the mining algorithm (especially level-2) the number of patterns tends to be very large. We can seek to combine the strengths of both approaches by employing the horizontal approach to begin with, and switching to vertical mining thereafter. It should be noted that the toolkit presently employs vertical mining for the entire algorithm, and incorporating the horizontal approach is part of future work. Vertical mining seeks to cut down I/O by utilizing the information stored with each pattern. Thus, this represents a trade-off of memory v/s I/O. The additional VAT data structures occupy considerable space, and we present schemes to maximize memory utilization under the vertical mining approach. The VAT structure and their intersections are detailed in Section 5.2.3.

2.7 Motivation

Our toolkit addresses instances of FPM. However, since there currently do exist algorithms that solve various FPM tasks, in this section we give the motivation behind our work.

The spectrum of FPM problems varies from itemsets (simplest) to graphs

(most challenging). In addition to the four primary pattern types discussed above, there exist other pattern types which can be mined, e.g. free trees [10] and directed acyclic graphs (dags) [9]. However, the current practice in frequent pattern mining basically falls into the paradigm of incremental algorithm improvements and solutions to specific problems. While there exist libraries such as MLC++ [22] and Weka [29], these are toolkits for entire space of data mining or machine learning algorithms. There is little generality among the various algorithms. However, we believe that the space of FPM problems are inter-related and hence that much can be gained from a holistic viewpoint of the problems. Figure 2.7 substantiates this claim. An itemset can be viewed as a graph with empty edge set; a sequence is a tree constrained to have outdegree of each vertex as one; and trees are directed, acyclic graphs with an indegree of at most one for each node. Thus, it is evident that FPM tasks are inter related. We believe that a better understanding of such relationships can facilitate more efficient algorithms for existing patterns, and also help design algorithms for newer pattern types. In [39], we had explored the unifying features among the internal working of various mining algorithms. That work served as a precursor to the current framework; the insights gained from it were very useful in designing this toolkit.

We propose a framework for addressing the class of FPM problems, rather than looking for solutions for specific FPM instances. The driving motivation was to search for a unifying theme among the class of FPM problems. We introduce the notion of properties (Chapter 3) towards this end. The main contributions of DMTL are:

- Developing a framework for solving problems that belong to this class as a whole. Our work poses fundamental questions such as: what characteristics distinguish tree mining from graph mining? We believe that finding answers to such questions shall help identify common tasks among FPM instances and motivate research in generic pattern mining.
- We introduce a property-based approach to clearly analyze and delineate various patterns and their FPM instances. We distinguish patterns based on pattern-properties and employ mining-properties to represent the varying

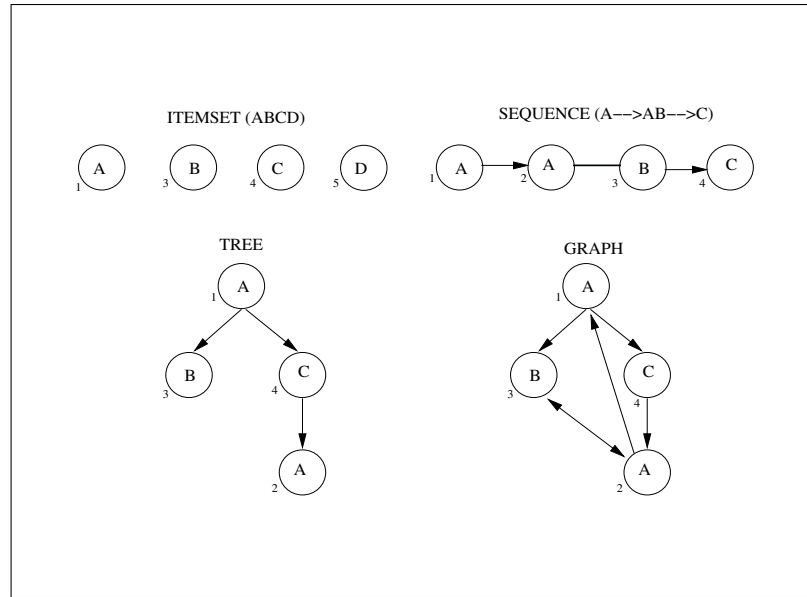


Figure 2.1: FPM Instances

FPM instances.

- Showing how the framework may be extended by incorporating new FPM tasks with minimal modifications. This toolkit attempts to provide a set of standardized solutions to common FPM tasks, at the same time keeping the framework extensible so it can be utilized for newer mining tasks e.g. induced mining.

With this background, the next chapter introduces our property based framework. Properties help us formulate and quantify the relationships between pattern types. We exploit such similarities between pattern types to design the framework.

CHAPTER 3

PROPERTY FRAMEWORK

There exist a slew of pattern mining algorithms, where each new algorithm solves a different FPM instance e.g. Apriori solves itemset mining, SPADE performs sequence mining etc. There exists a wide variety of substructure mining algorithms as well e.g. gSpan, FFSM, TreeMiner, uFreq etc. Each of these algorithms is an FPM instance, and the motivation of our work is to explore a framework addressing the entire class of FPM problems.

While belonging to the same class of problems, each FPM instance differs from another in some definite way. We introduced the concept of a property to formalize these characteristics. A property is an abstract notion in itself. We define a *property* to represent a characteristic that distinguishes an instance of the FPM problem from another. Such characteristics may arise in two forms:

- in the pattern definition; the definition of the pattern obviously greatly influences the FPM task, and
- differences in the mining process, for instance how the support of patterns is counted or how candidate patterns are generated.

We term a property of the first kind *pattern property*, and the latter as *mining property*. A complete definition of an FPM problem must indicate both these property sets to be identified unambiguously. Details on each of the property types follow.

3.1 Pattern Properties from Pattern Relations

A pattern property is one that helps qualify its definition. There are four primary categories of patterns in existence today: itemset, sequence, tree and graph. Graphs are the most complex pattern, and itemsets the least. We can define a hierarchy based on the geometric attributes of the above four patterns - and these very attributes guided us towards identifying pattern properties that would distinguish

one pattern from another. Every other pattern can be represented as a graph, in particular as a constrained graph. In other words, each pattern property defines a constraint imposed on the generic graph structure.

We present a mathematical motivation for our formulation of the pattern property space. We introduce properties corresponding to constraints imposed upon the FPM pattern space. Assume a graph g with vertex set V_g and edge set E_g , then for any $x \in V_g$ the set $in_g(x) = \{y | (y, x) \in E_g\}$ represents the incoming edges of vertex x . Similarly we define $out_g(x) = \{y | (x, y) \in E_g\}$. We now present the following categories of properties (constraints) of the graph:

1. Edge relation: The edge set E_g is defined as $E_g \subseteq V_g \times V_g$. The edge relation may possess several properties -
 - reflexive - $x E_g x, \forall x \in V_g$. Determines if self-loops are permitted in the graph. We typically deal with irreflexive graphs.
 - symmetric - $x E_g y$ implies $y E_g x, \forall x, y \in V_g$. A symmetric edge relation leads to an undirected graph, which we characterize by the **undirected** property. A directed graph (asymmetric edge relation) is denoted by the **directed** property.
 - transitive - $x E_g y$ and $y E_g z$ implies $x E_g z, \forall x, y, z \in V_g$. Transitivity leads to embedded patterns, which we denote with a property named **embedded**. An intransitive edge relation implies a pattern with the **induced** pattern. As we shall see, transitivity falls under the purview of mining properties. Given $x \in V_g$, transitive-closure of x , denoted as $tc(x)$ is defined as follows: $tc(x) \subseteq E_g$ where $tc(x) = \{(x, y) | \exists z \in V_g, x E_g z \ \& \ z E_g y\}$.
 - cyclicness - A pattern possess the **cyclic** property if $\exists x \in V_g$ such that $(x, x) \in E_g$ under the assumption that all edges in $tc(x)$ exist. In other words, a graph is cyclic if at least one vertex is reflexive on edge relation assuming all of the vertex's transitive closures hold. Else the pattern possesses the **acyclic** property.

- **order** - The ordered property states the existence of an ordering on the neighbors of a vertex, else the pattern is said to be unordered. Ordering is usually relevant for tree patterns only.
2. **Degree**: For a vertex $x \in V_g$, its indegree, denoted as $indegree_g(x)$, is $|in_g(x)|$. We define $outdegree_g(x)$ similarly. Some patterns are specializations of others constrained on the degree. For instance, an ordered tree is an acyclic, directed graph each of whose vertex's indegree is not greater than one. Similarly, a sequence is a subpattern of an ordered tree - in a sequence each of the vertex's outdegree is not greater than one.
- **indegree_lte_one** This property constrains all vertices of a graph to have $indegree \leq 1$.
 - **outdegree_lte_one** This property constrains all vertices of a graph to have $outdegree \leq 1$.
3. **Label**: A function $l : V \rightarrow L$ maps each vertex to a label from set L . All patterns except an itemset possess such a labeling function.
- **unique_label** This property requires the labeling function to be one-to-one (injective). Each vertex thus maps to a unique label, a common example of such a pattern is an itemset.

3.2 Property Hierarchy

The characteristics of a pattern play a central role in the solution to an FPM instance. Hence, in an attempt to provide some order to the vast number of FPM algorithms existing today, we have devised a property hierarchy (Figure 3.1). The hierarchy is based on the pattern properties defined earlier. The hierarchy provides a systematic model for analyzing attributes which distinguish one pattern from another. For instance based on the hierarchy we can conclude that an unordered tree is a dag each whose vertex's indegree is ≤ 1 . Importantly, it also establishes relationships between pattern types. We say a pattern type higher in the hierarchy

encapsulates the *specializations* below it. For a pattern type \mathcal{P}_1 , \mathcal{P}_2 is its specialization if the mining algorithms for \mathcal{P}_1 perform correctly for \mathcal{P}_2 as well. In the absence of a mining component for \mathcal{P}_2 , this enables us to employ the corresponding component of \mathcal{P}_1 . We term this phenomenon *upward propagation* of properties. For instance, a tree mining algorithm is capable of mining sequences as well and this fact is captured well with this hierarchy. Obviously, a specialized component for \mathcal{P}_2 may potentially be more efficient, but this need not hold always. Nevertheless, our toolkit facilitates mining of new pattern types by maximizing re-use of generic components. We show how such re-use of components works in Section 4.2.3.

The hierarchy was designed following careful analysis of the existing pattern space. It is by no means exhaustive, nevertheless we do expect to capture the relationship between common patterns. New patterns may be appended to the hierarchy by defining new properties as required. Since we have employed the basic geometric attributes of patterns to formalize the properties, any pattern can be specialized by adding a new property specialization.

3.3 Mining Properties

A mining property discloses how the mining process should act. The mining properties are independent of pattern properties. Analysis of existing FPM tasks revealed the following mining property categories. Each FPM instance is required to pick exactly one property from each category. As with pattern properties, new mining properties can be added through the template mechanism illustrated in the next chapter.

1. join-type: This category influences the candidate generation phase, in which potentially frequent pattern are generated, to be counted later to determine their support. During candidate generation, the algorithm typically constructs a new pattern p' by *joining* two frequent patterns p_i, p_j . p' is termed a *candidate* pattern and p_i, p_j are its *parents*. The nature of this join is a property itself. A suitably correct and complete algorithm has to be provided for the chosen property.

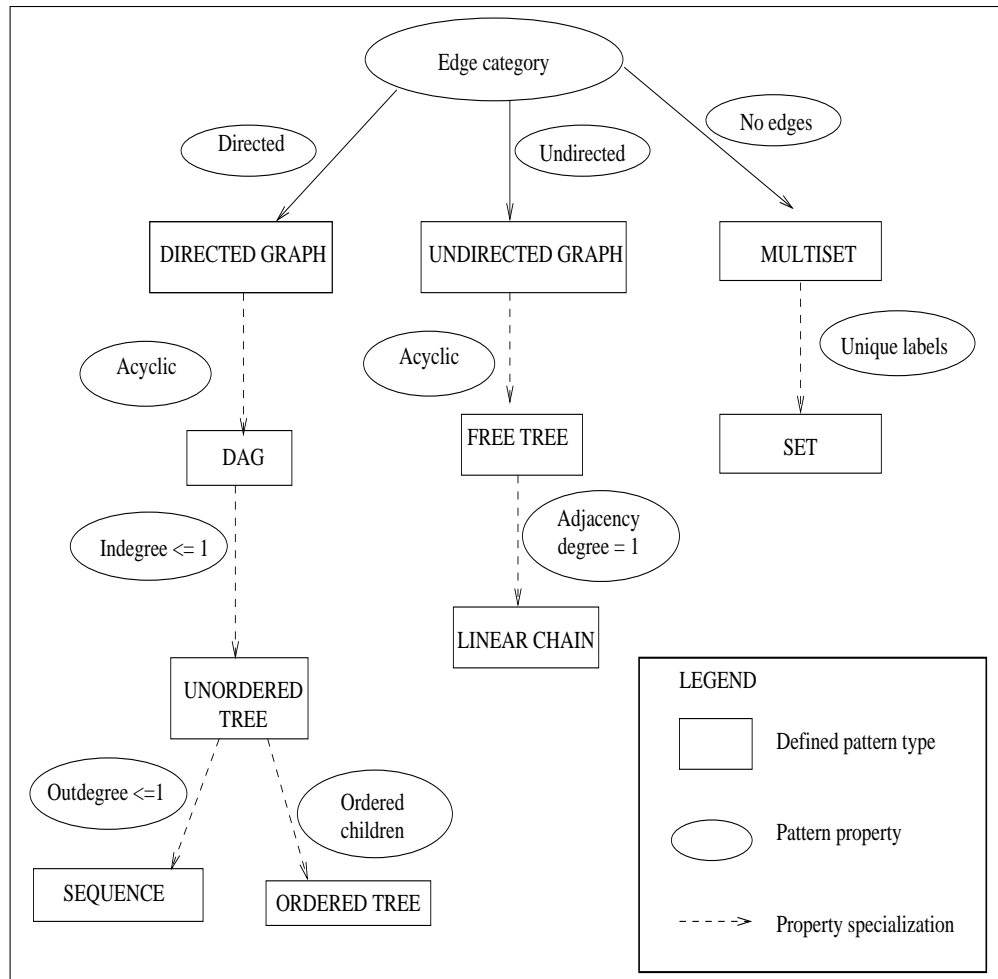


Figure 3.1: Pattern Property Hierarchy

- $\mathbf{F}_k \times \mathbf{F}_1$: A $(k + 1)$ -length pattern is constructed from a k -length parent and a unit length one.
 - $\mathbf{F}_k \times \mathbf{F}_k$: A $(k + 1)$ -length pattern is constructed from two k -length parent patterns. It is implicitly based on the notion of equivalence classes, which we discuss in Section 4.1. This join is usually more efficient since it generates fewer infrequent candidates.
2. support-counting: This category specifies how the support of a candidate pattern is determined. Two ways exist in use today:
- **horizontal** Indicates that the support for a candidate pattern shall be

determined by counting its occurrences in the transactional database. Since large databases may not entirely fit in memory, this method usually involves significant I/O overhead.

- **vertical** Support for a process is determined based on the intersection of Vertical Attribute Tables (VATs) of its parent patterns. Each pattern has an associated VAT, where the VAT identifies every unique occurrence of the pattern in the database. Vertical support counting is typically faster as it reduces the I/O cost.
3. **transitivity**: This category indicates if embedded occurrences of a pattern should be included in its support. Though transitivity of the edge relation was discussed in the previous section (Section 3.1), we place it among mining properties for the following reason: transitivity does not affect the pattern definition, rather it impacts how the support for a pattern is determined.
- **induced** Only induced occurrences are counted.
 - **embedded** Transitive closures on the edge relation are included in the support as well. The transitivity leads to discovery of embedded occurrences of the pattern.

The property-based model enables us to move from the *pattern-space* to a *property-space*. The *property-space* lends structure to the class of FPM problems, and its formulation is one of the main contributions of our work. This model clearly demarcates features of the mining process from variations in the pattern definition. We believe that such a unified framework shall facilitate comparative analysis among the various FPM problems. In the next chapter we give an overview of our implementational approach and elaborate upon some of the design choices.

CHAPTER 4

DESIGN PRINCIPLES OF THE IMPLEMENTATION

Our implementations of itemset, sequence, tree and graph algorithms are based on ECLAT [32], SPADE [34], TreeMinerV [35] and gSpan [30]. In order to better understand our implementations of these mining algorithms, we first present an overview of the mining approach that is common to all of these algorithms. The next section looks at the generic FPM methodology, thereafter we discuss two key features of the toolkit: i) template-based approach and ii) support for persistency, buffer management and DBMSs.

4.1 Mining Methodology

The general approach of a mining algorithm is given in Figure 4.1. **Enumerate-Frequent-Patterns** outlines the main tasks to be undertaken by any mining algorithm. We do not require a pattern type to conform to this precise formulation of the mining process. However, most FPM algorithms (including the ones in our toolkit) do conform closely to this outline. The mining process searches incrementally in the pattern space: first the set of 1-sized patterns, F_1 is enumerated, then the set of 2-sized patterns, F_2 , is grown from it and so on till no more frequent patterns exist in the database. Hence, the algorithm undergoes several iterations, at each iteration discovering frequent patterns sized one greater than the previous. Each iteration can itself be broken down into three main steps:

Candidate Generation Given the set of k -sized frequent patterns, \mathcal{F}_k , this step generates the set of $(k + 1)$ -sized candidate patterns, denoted as \wp_{k+1} . It may be recalled from Section 2.2 that a candidate pattern is one that *may* potentially be frequent. Hence, for correctness of the algorithm, $\mathcal{F}_{k+1} \subseteq \wp_{k+1}$, i.e., every frequent pattern must be generated as a candidate. Obviously, varying pattern types differ in how they generate candidates. However, there does exist significant commonality among the varying pattern types. This was explored by us in a previous work [39].

<p>Enumerate-Frequent-Patterns ($[P]$):</p> <ol style="list-style-type: none"> 1. $\wp_{k+1} = \text{cand_gen}([P])$ 2. \forall candidates $c \in \wp_{k+1}$ 3. if ($\text{check_isomorphism}(c)$) then 4. $\text{count_support::count}(c, DB)$ 5. if ($c.\text{sup} \geq \text{minsup}$) then 6. $\mathcal{F}_{k+1} = \mathcal{F}_{k+1} \cup c$ 7. for every equivalence class $[P_i] \in \mathcal{F}_{k+1}$ 8. Enumerate-Frequent-Patterns($[P_i]$)
--

Figure 4.1: Pattern Mining Algorithm

Isomorphism Checking All candidate patterns are then passed through an isomorphism checker to prune isomorphisms. For a pattern $p = (V, E)$ belonging to generic pattern type \mathcal{P} , an isomorphism may be loosely defined as a bijective function $\varphi : V \rightarrow V$ such that it preserves edge and vertex labels, as well as structure of the pattern (Sections 2.4, 2.5). Detecting such isomorphic patterns is critical for two reasons: i) the number of frequent patterns shall be incorrect since we might end up repeatedly enumerating isomorphisms of the same pattern, and ii) the search space would redundantly grow exponentially with number of isomorphic patterns. To overcome these issues, isomorphism checking and pruning of isomorphic patterns is essential. Fortunately, for itemsets and sequences we can circumvent generating isomorphic patterns by intelligent candidate generation [4, 34]. Essentially, we can use lexicographic ordering on the labels to avoid generating redundant patterns. Isomorphism checking can also be avoided for ordered trees by an appropriate candidate generation scheme [35]. However, unordered trees [25, 36], free trees [10] and graphs [30, 17] require isomorphism testing.

Support Counting The last step in an iteration is to determine the support of candidates, and discard ones that do not pass the *minsup* criterion. This may be achieved through the horizontal or the vertical approaches.

Candidate generation may be achieved through either of $\mathbf{F}_k \times \mathbf{F}_k$ or $\mathbf{F}_k \times \mathbf{F}_1$ mining properties. We circumvent isomorphism checking for itemsets and sequences

through intelligent candidate generation which avoids generating isomorphic patterns. Isomorphism checking cannot be avoided for some trees and graphs, and is accomplished using one of the existing approaches. Our toolkit is not tied with any particular isomorphism checker and a more efficient, if available, may be plugged in at a later point. The properties **horizontal** or **vertical** can be used to choose the type of support counting method desired. The default currently is vertical mining.

It should be noted that the approach sketched out in Figure 4.1 incorporates two key aspects of FPM algorithms: *right-most extension* and *equivalence class based growth*. Recent approaches at solving tree and graph mining [35, 30] have focused on the right most extension strategy: a new node is added to the pattern only on the right most path from a designated root. This method exhaustively enumerates all candidates for all patterns ranging from itemsets to graphs. Equivalence class based growth is also a common trait among several FPM algorithms and leads to efficient candidate generation [32, 34, 36]. Hence we have modeled our current algorithms on equivalence class growth. The equivalence classes here are prefix based, i.e., two patterns belong to the same class iff they share a common prefix. Define a function $\phi(p, i)$ to return the first i elements in p (the pattern type of p enables us to define an ordering on the elements in it and hence obtain the first i elements). Define an equivalence relation θ on k -sized patterns p_x and p_y such that $p_x \theta p_y \Leftrightarrow \phi(p_x, k-1) = \phi(p_y, k-1)$. The above relation states that two patterns belong to the same equivalence class iff they share a common prefix up to the last but one element. Equivalence class growth facilitates $\mathbf{F}_k \times \mathbf{F}_k$ candidate generation. We generate a $(k+1)$ length pattern from two k -sized parents belonging to the same equivalence class. With this approach, the possible number of candidates with an equivalence class $[P]_k$ of k -sized patterns is $|[P]_k|^2$. This may be contrasted with the $\mathcal{F}_k \times \mathcal{F}_1$ approach, in which each pattern in \mathcal{F}_k can potentially be joined with each one in \mathcal{F}_1 , yielding a total number of $|\mathcal{F}_k||\mathcal{F}_1|$ possible candidates. Typically, $|[P]_k| \ll |\mathcal{F}_1|$, which explains why $\mathcal{F}_k \times \mathcal{F}_k$ is much faster than $\mathcal{F}_k \times \mathcal{F}_1$.

Another noteworthy feature of the algorithms used is the *search strategy* in the pattern space. Several variants exist, depth-first search (DFS) and breadth-first search (BFS) being the primary ones. A BFS approach generates all k -sized

patterns, then all $(k+1)$ -sized and so on. BFS has the advantage of providing better pruning of candidates but suffers from the cost of storing all of a given levels frequent patterns in memory. For large datasets this can be a significant constraint. Recent algorithms for mining complex patterns like trees and graphs have focused on the DFS approach. A DFS approach explores the search space based on equivalence classes, generating candidates in a depth-first manner. Candidates of the same length may no longer be enumerated together, but such a DFS approach facilitates better memory management. At a time, only patterns of the same equivalence class need be kept in memory. Hence it is the preferred choice for our toolkit as well. However, variants may be explored as future work.

4.2 Template based Approach

Templates are a powerful element of C++, which motivate design in the generic programming paradigm. The C++ Standard Template Library (STL) [2] has demonstrated the power and efficacy of the use of templates as a design tool. Templates are a crucial part of our design. Templates enable us to formulate generic data structures and algorithms, and this is central to our theme of devising a generic framework.

4.2.1 Generic Containers

We have designed generic containers to encapsulate and abstract away generic tasks and components. The *pattern* class encapsulates generic methods for a pattern entity; the *count_support* class abstracts the functionality of counting a pattern's occurrences in the dataset; and *db_reader* class is a generic wrapper for reading and parsing the input database. These are elaborated in Section 5.1.

4.2.2 Specializations

We make extensive use of class and function template specialization [24]. This enables us to define generic classes and methods and at the same time to provide efficient implementations for specific ones. Template specialization facilitates compile-time resolution of function calls, and we exploit this mechanism heavily.

```

template<class prop, class next_property=null_prop>
class proplist {
public:
    typedef prop FIRST;
    typedef next_property SECOND;
};

```

Figure 4.2: A Static Type List

4.2.3 Static Type List

A critical design requirement of our property-based formulation was to develop a static list of types - to provide for accumulating properties. Each pattern is uniquely identified by several pattern properties; mining properties play the same role in determining the specific FPM task. Furthermore, each pattern is defined by varying number and types of pattern properties. Hence, a container to hold various properties was needed. We model properties as types themselves. In order to incur as little runtime overhead as possible, the requirement was of a *static type-list*.

A static type-list is a generic, extensible container of types, just as the STL list is a generic container of objects. It should be noted that such a type-list is a static accumulator i.e. it relies on the template compile-time mechanism and hence incurs no runtime overhead. We have modeled each of our properties as distinct classes, which are then collected into a static type-list. A type list gives us the flexibility to append properties to it, making the design generic and extensible.

We designed a data structure which gives us the aforementioned benefits, it is called a `proplist`. It was designed by borrowing ideas from the collection of C++ Boost libraries - the Boost Graph Library and the MetaProgramming Library [1]. Since it is simply a holder of types, the class itself is uncomplicated and is given in Figure 4.2.

The class `null_prop` is used as a delimiter in the type-list. In addition to its utility as a type-list, the `proplist` possesses the nice feature of facilitating upward propagation of properties, initially described in Section 3.2. This behavior is demonstrated in Figure 4.3.

For sake of demonstration in the figure, we create dummy property classes and

```

/// dummy property class definitions ///
class directed {};
class acyclic {};
class planar {};
class null_prop {};

/// generic function ///
void candidates(const proplist<directed>&);

/// specialized function for DAGs ///
void candidates(const proplist<directed, proplist<acyclic> >&);

///// an illustration of how it works /////
proplist<directed> digraph;
proplist<directed, proplist<planar> > planar_graph;
proplist<directed, proplist<acyclic> > dag;

/// following function call compiles to generic function ///
candidates(digraph);

/// following function call compiles to specialized function ///
candidates(dag);

/// following function call compiles to generic function ///
candidates(planar_graph);

```

Figure 4.3: Application of Property Hierarchy

give the prototype of a function that generates candidates from a given pattern (it is assumed that the function populates a global parameter with the generated candidates). As pointed out in Section 4.1, candidate generation is one of the three tasks a mining algorithm must undertake. In the figure, two prototypes of `candidates()` are provided - one for digraphs and one for dags. Since dags do not possess cycles, the specialized function does not generate cyclic graphs as candidate dags. On the other hand, the generic function generates all possible digraphs, including cyclic ones. It should be noted at this point that the pattern property, `planar` is not defined in our toolkit. Hence, it is a new pattern property representing planar graphs. Let \mathcal{P}_1 denote the pattern type, digraphs, and \mathcal{P}_2 denote directed, planar

graphs. Since $\mathcal{P}_2 \prec \mathcal{P}_1$, the toolkit enables the user to invoke the candidate generation of digraphs for planar, directed graphs. The user is free to plug in a more efficient implementation for planar digraphs, if one exists. In case it does not, our framework enables such re-use of components. It is to be noted that we implicitly utilize the property hierarchy to provide this feature. The hierarchy inherently defines a pattern lower down to be a specialization of one above it. We capture these very relationships with our definitions of `propList` and its use in the library. We believe demonstrating such clear relationships between patterns and their mining algorithms is a noteworthy contribution of our work.

4.3 Other Features

The toolkit also provides additional functionalities to aid users in the mining process. Frequent pattern mining is often performed on very large datasets, which may not fit in memory on most machines. Most mining algorithms do not provide explicit means of memory management nor is the issue addressed within the algorithm. We seek to provide a systematic solution by undertaking explicit buffer management and providing persistency of the mined results. Within the vertical mining paradigm, we can efficiently manage the memory used by VATs. This work is under progress, hence we outline the approach being taken.

We aim to maintain a fixed size buffer, where the size is provided by the user. The two primary structures we need to store are the patterns and the VATs. In order to not exceed the buffer size, we maintain a buffer replacement policy. All patterns and VATs that do not fit in memory are shipped to disk or a database. At any point in the mining algorithms, at most two patterns and their VATs are required to be memory resident and hence we are guaranteed to find patterns or VATs that can be evicted. We also aim to provide DBMS support in the toolkit. Current alternatives being looked into include Gigabase [21] and PSTL [14]. Gigabase is an embedded object relational database which provides its own buffer management. PSTL is a library of persistent containers, in design akin to STL. PSTL achieves persistency by maintaining employing memory mapping of data files. This ensures that mining results and intermediate data (like VATs) are stored on disk and available for pro-

cessing at a later point. The toolkit currently supports in memory computations for all algorithms.

The next chapter describes the basic building blocks of the toolkit - the main classes, their roles and discusses the impact of some of the design decisions.

CHAPTER 5

DMTL ARCHITECTURE

This chapter gives an overview of the core classes employed by DMTL. The toolkit comprises several generic classes as well as a few specialized ones. The generic classes form the central design of the library. A few classes have been specialized to provide specific and efficient implementations for distinct pattern types.

5.1 Generic Components

In this section we look at the main generic classes in our design. These classes accomplish generic tasks, often relying in turn on specialized function or classes.

5.1.1 Reading in the Database

One of the first tasks for a mining process is to read and parse the input database. The toolkit can currently parse flat file based datasets, however we aim to provide support for DBMS based datasets shortly. This step represents the first iteration of the mining process; it determines all frequent length-1 patterns in the dataset. The `db_reader` class encapsulates this process for a generic pattern type. The class depends on a few pattern type-specific functions to accomplish its task. It utilizes the `tokenizer` class, specialized on the pattern type to parse the specific pattern type. Additionally, in the case of vertical mining, `db_reader` computes VATs for length-1 patterns. These VATs are used later in the mining process during the support counting stage. In the vertical approach, the input database is usually not accessed directly after this first iteration.

5.1.2 The Pattern Class

The pattern class holds an instance of any pattern type, its high-level interface is shown in Figure 5.1. As indicated in Section 2.7, graphs are the most complex pattern type. Additionally, every other pattern type viz. itemset, sequence and tree, can be represented as a constrained graph. We exploit in our representation of pat-

terns - every pattern is stored as a graph. The template parameter `pattern_props` indicates the pattern type, by specifying constraints through properties discussed in Section 3.1. Parameter `mining_props` specifies the particular FPM task to be performed (Section 3.3). Both these parameters are actually static type lists. As discussed earlier, each pattern is represented as a graph, and is stored internally as an object of type `graph_model`. We have defined a custom adjacency list class for this purpose, though there are no specific requirements of it. The `canonical_code` template parameter is provided to facilitate isomorphism checking. For simpler patterns such as itemsets, sequences and trees, this member is not used. Graphs are more complex, and typically necessitate storage of some information to assist in isomorphism checking. No restrictions are placed on the `canonical_code` parameter; gSpans's *dfs code* is used, FFSM's *CAM* may be used alternatively. It should be noted that the isomorphism checking component would have to be consistent with the `canonical_code` representation chosen.

The pattern class provides functions for adding vertices and edges, as well as accessing the neighbor list of a vertex. Our implementation relies only on these generic functions to perform specific FPM tasks e.g. an itemset is represented as a graph, though with an empty edge-list. Hence all edge based functions would have no meaning in itemset mining; the entire mining task is achieved by calls to `add_vertex()`. A sequence, tree or graph FPM would issue calls to the edge functions. In particular, undirected patterns such as free trees and graphs employ calls to `add_out_edge()` only. Incoming edges and the calls to `add_in_edge()` hold meaning for digraphs, trees, sequences and other directed patterns. The routine `get_rmost_path()` is provided to facilitate right most extension for candidate generation as discussed in Section 4.1.

5.1.3 Determining Support of a Pattern

The `count_support` class encapsulates the functionality of counting the occurrences of a pattern in the input database. It is templated on the mining properties and the storage manager employed. There exist two specializations of `count_support`, for vertical and horizontal mining respectively. Horizontal min-

```

template<class pattern_props, class mining_props,
class graph_model, class canonical_code>
class pattern {
    add_vertex(v);
    add_out_edge(v1, v2, e);
    add_in_edge(v1, v2, e);
    get_neighbors(v);
    get_rmost_path();
};

```

Figure 5.1: Pattern Class Interface

ing entails determining the support of a pattern from the transactional database. Vertical mining associates a VAT with each pattern. The VAT is devised so as to unambiguously identify the occurrences of a pattern in the database. As detailed in Section 2.6, support computation is simplified with the use of VAT - for a pattern p we *intersect* the VATS of p 's parent patterns to get p 's VAT and hence its support. Thus vertical and horizontal mining represent a trade-off of space against time: vertical mining is typically faster by associating additional information with each pattern; while the horizontal approach may take lesser space but suffers from the cost of repeated I/O accesses. These differences are abstracted away into the `count_support` class, and thus the mining process is oblivious of details of the counting operations. We employ the VAT definitions outlined in Section 5.2.3 for itemsets, sequences, trees and graphs respectively. More details on VAT intersection and its interaction with `count_support` are provided therein. The search strategy over candidate patterns (breadth-first or depth-first discovery) can also be modeled as an additional mining property.

5.1.4 Support of a Pattern

Another generic component is a pattern's support. The support is always an integer value, independent of the pattern type. However, we need to store additional information when performing induced mining using $\mathcal{F}_k \times \mathcal{F}_k$ candidate generation. We abstract this entity into a `pattern_support` class. The generic definition of the class holds a plain integer based support, and we provide a specialization capable

```

template<class pat, class vat, class buffer>
class storage_manager {
    add_vat(pat p, vat v);
    vat get_vat(pat p);
    delete_vat(pat p);
    intersect(pat p1, pat p2);
};

```

Figure 5.2: Storage Manager Interface

of storing more information for induced, $\mathcal{F}_k \times \mathcal{F}_k$ mining.

5.1.5 Storage Manager

The `storage_manager` is a generic class that encapsulates buffer management options and interactions between a pattern and its VAT. It is employed internally by `count_support`'s specialization for vertical mining. The class acts as a generic wrapper, and is templated on the pattern, `pat`, the VAT structure, `vat` and the buffer policy, `buffer` (Figure 5.2). The pattern definition is identified with a static type-list of properties. `vat` is the VAT data structure for the pattern definition, `pat`. The parameter `buffer` allows for sophisticated buffer policies. Currently little buffer management is provided; extending the storage manager with buffering policies is one of the features currently in development. When mining large datasets, such a feature is expected to ease the mining process significantly.

The storage manager is essentially a storehouse of patterns, their VATs and the mapping between them. Figure 5.2 gives the interface for this class. The default for `buffer` parameter is the in-memory manager. All objects are kept in memory unless explicitly reclaimed by the user. `add_vat()` stores the pattern `p`, and its VAT, `v` in the manager, and adds the mapping between them. This mapping is utilized by `get_vat()`, which returns `vat(p)` given `p`. The mapping and VAT may be deleted from the store by `delete_vat()`. The `intersect()` routine performs VAT intersection - it intersects the VATs of `p1` and `p2`, and returns the resulting VAT. Since the VAT intersection depends on the internal structure of the VAT, each distinct VAT data structure is expected to provide an intersection routine, which is

internally invoked by `intersect()`.

5.2 Specialized Components

Several specialized components are needed by the generic framework. The components are specialized on the pattern type, they perform specific tasks for a given pattern type. Some of the main ones are listed below.

5.2.1 Parsing a Transaction

The `tokenizer` class is responsible for reading in and parsing one transaction from the input database. Since the transaction format varies with pattern type, there exist four specializations of this class - one each for itemset, sequence, tree and graph pattern types. A consistent interface across each specialization enables the `db_reader` class to invoke the appropriate tokenizer seamlessly. `db_reader` calls `tokenizer` for each input transaction; the specialized `tokenizer` class processes a transaction and returns the patterns found in it. It is to be noted that a change in database format would necessitate modifying the tokenizers accordingly.

5.2.2 Generating Candidates

Each distinct pattern type is provided with a `cand_gen()` routine for candidate generation. The design choices in play here - equivalence classes, right most extension and depth-first generation, were discussed in Section 4.1. There are four candidate generation routines provided for each of the primary pattern types. As pointed out in Section 3.2, upward propagation of properties may be employed to re-use existing components for new pattern types. We employ $\mathcal{F}_k \times \mathcal{F}_k$ equivalence classes for itemset, sequence and tree patterns. $\mathcal{F}_k \times \mathcal{F}_k$ equivalence classes have not yet been devised for graphs, hence we utilize $\mathcal{F}_k \times \mathcal{F}_1$ equivalence classes with depth first generation. These algorithms broadly conform to the outline delineated in Figure 4.1. A candidate pattern is generated by *joining* two frequent patterns (Section 3.3). We provide a brief review of the join routines below. Further details of the algorithms may be found in the corresponding references.

- Itemset: Itemset join is the least complex among all patterns. It is simply the

set union operation over elements of the two parent patterns. The join operation is defined for two itemsets Px and P_y , both belonging to the equivalence class, $[P]$ - this join yields $Pxy \in [P_x]$.

- **Sequence:** An equivalence class of sequences comprises sequence atom members, of the form $P \rightarrow X$. Event atoms (such as PY) are not considered in our framework, but have been explored in [34]. Hence, each event in a sequence consists of a single item only. A join of two sequences within the same equivalence class, $P \rightarrow A$ and $P \rightarrow B$, can lead to two sequence atoms, $P \rightarrow A \rightarrow B$ and $P \rightarrow B \rightarrow A$.
- **Tree:** An equivalence class of trees comprises members which share the common prefix, but differing in the last node of the tree and the position where it is attached to the prefix. Hence members of the same equivalence class $[P]$ may be denoted as pairs of $(last_node, position)$. A join of (x, i) with (y, j) leads to the following possibilities: i) if $i = j$ add (y, j) and (y, n_i) to $[P_x]$, where n_i is the depth-first number of node x ; ii) if $i > j$ the new candidate is (y, j) in class $[P_x]$; and iii) no candidates are possible when $i < j$. We refer the reader to [35, 36] for elaboration on the prefix based representation scheme used for trees.
- **Graph:** We currently mine undirected graphs, however extending it to digraphs is not challenging. To assist in systematic candidate generation and isomorphism testing, DMTL uses the ordering of vertex and edge labels to generate graphs from a core tree structure [30]. This tree structure is built by a DFS traversal of the graph [12]. An $\mathcal{F}_k \times \mathcal{F}_1$ join on graphs is a complex operation; at each such extension a new edge is added to the given graph. Two types of edge extensions are defined: a *back edge* which introduces a cycle, and a *forward edge* which adds a new node to the graph.

5.2.3 Vertical Attribute Lists (VATs)

In DMTL there is one VAT structure per pattern-type. Each VAT structure has its own intersection routine defined. We now describe briefly the VAT structures

and their intersection for the primary pattern types. We propose a VAT structure for graphs, which we believe is a noteworthy contribution of this work. The description below assumes usage of containers from the `std` namespace of the C++ STL. VAT structures for itemsets, sequences and trees have been borrowed from the respective algorithm used, and we refer the reader to the pertinent work for more elucidation.

- Itemset: For an itemset the VAT is simply a `vector<tid>`, where each `tid` may be stored as an `int`. In other words, the VAT stores the set of tids it is contained in. VAT intersection in this case is straight forward; for an itemset I_{xy} , obtained by join of I_x and I_y , $vat(I_{xy})$ is simply the set intersection operation over $vat(I_x)$ and $vat(I_y)$.
- Sequence: The VAT for a sequence is defined as a `vector<pair<tid, vector<time-stamp>>>`. In this case the intersection has to take into account the type of extension under consideration. The intersection operation is a simple intersection of tid-lists for a join of two event atoms, but requires comparison of the timestamps when doing sequence joins. For instance, when computing the VAT intersection for $P \rightarrow A \rightarrow B$ from its parents $P \rightarrow A$ and $P \rightarrow B$, one needs to match the tid *and* ensure that the timestamp of A in that tid is less than that of B . Our toolkit possesses the capacity to mine induced as well as embedded sequences. The same VAT structure is retained across the two variations, but for an additional parameter stored for induced VATs. The gap between any two consecutive events in an induced sequence is one. This parameter facilitates checks for unit time gap.
- Tree: Let a triple be `(tid, scope, match-label)`, then the VAT for a tree pattern is a `vector<triple>`. The `tid` identifies a tree in the input database; `scope` is an interval `[l,u]` which denotes the range of DFS ids which lie embedded under the last depth-first node of the tree, and `match-label` is a list of DFS positions at which the current tree is embedded in that tree of the database. Intersection of tree VATs is an involved operation, comprising in-scope (child extension) and out-scope (cousin extension) tests corresponding

to the two types of tree extensions described earlier. Embedded and induced tree mining require distinct VAT intersection routines [36]. The structure of tree VAT remains almost the same, however there are minor differences in the intersection. The nature of induced vs. embedded tree mining was observed to be similar to the same problem in sequences.

DMTL provides support for creating VATs during the mining process, i.e., during algorithms execution, as well as support for updating VATs (add and delete operations). In DMTL VATs can be either persistent or non-persistent. Finally DMTL uses indexes for a collection of VATs for efficient retrieval based on a given attribute-value, or a given pattern.

5.3 Graph VAT

We shall first discuss the VAT structure for graphs, and how it captures each occurrence of a graph in a dataset. VAT intersection operation for graphs is defined thereon. We propose a novel VAT representation for graphs. Intuitively, a graph is a collection of several edges. If we capture each occurrence of every edge belonging to a graph, we would have captured every occurrence of the graph itself. Hence, a graph VAT is defined to consist of edge VATs. At this point, it should be noted that in graphs we follow $\mathcal{F}_k \times \mathcal{F}_1$ extension, i.e., a new edge is added to a k -sized graph to get a $(k + 1)$ -sized graph. Furthermore, edges are added to vertices on the right most path only (by virtue of right most extension). Hence, to determine if the new edge can be attached on the right most path to get a valid occurrence, we need only keep edges on the right most path.

With this background, we give the following definitions. The VAT for a graph g is defined to be a collection of edge-VATs, **evats**, corresponding to each edge on the right most path in the graph. $vat(g)$ distinctly identifies each occurrence of g in a graph g_d , from the graph dataset GD . This is essentially the *subgraph isomorphism* problem: we wish to determine if $g \preceq g_d \forall g_d \in GD$. An **evat** captures occurrences of an edge within a graph. It is defined to be **vector**<pair<**vid**, **vid**>>: each occurrence of the edge is represented by storing the vertex identifiers (**vids**) of its two end points, and multiple such occurrences can be stored in the **vector**.

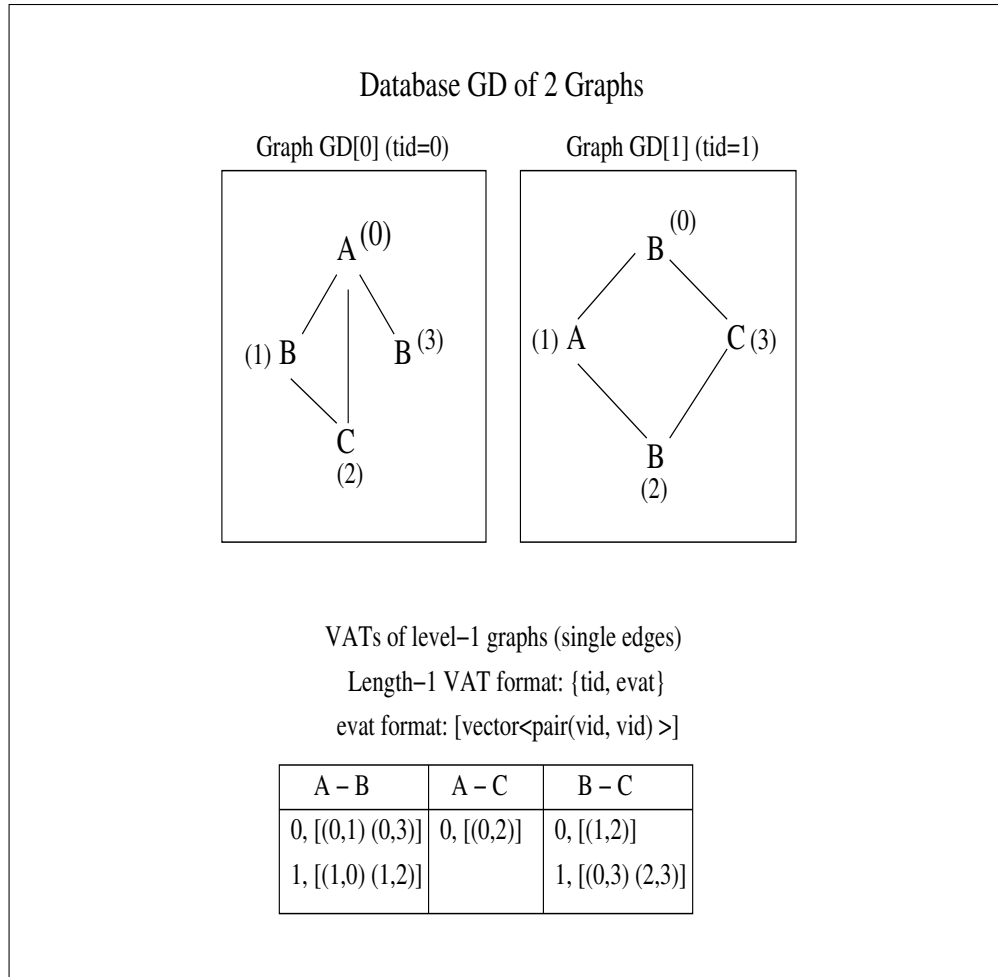


Figure 5.3: Sample Graph Database & Level-1 VATs

A unique vid is assigned to each vertex in a graph. The VAT definition for a graph is $\text{vector}\langle\text{pair}\langle\text{tid}, \text{vector}\langle\text{evat}\rangle \rangle \rangle$. Each occurrence of a graph is denoted by the pair, which itself consists of several *evats* to hold all edges on right most path. In this manner, the VAT enables us to capture every distinct occurrence of a graph. The scheme described above works for both undirected and directed graphs with minimal changes.

Figure 5.3 shows a sample dataset *GD* of two graphs. The nodes are represented by their labels (A, B or C) and each node's vid is indicated besides it in paranthesis. For ease of demonstration, we have ignored edge labels (all edges have the same dummy label) from this example. Dealing with labelled edges is however a

simple extension of the methodology described in the example, and is implemented in DMTL. The edge $(A - B)$ occurs twice each in $G1$ and $G2$. Hence its VAT has two entries, one for each graph. Furthermore, since $(A - B)$ is simply an edge in itself, its VAT is simply a single **evat**, which corresponds to $(A - B)$'s occurrence in $G1$: between vertex ids 0 and 1, and between 0 and 3. Moving to higher level graphs such as g_1 in (Figure 5.4, the **gvat** now comprises two **evats**, for the two edges $(A - B)$ and $(B - C)$. $(A - B)$'s **evat** lists its occurrences as part of g_1 , and hence $(A - B)$'s **evat** here differs from its VAT in Figure 5.3. In $GD[0]$ ($tid=0$), the edge between vertex ids 0 and 3 constitutes an $(A - B)$ edge, but it is not part of a linear $(A - B - C)$ chain. Thus the pair $(0, 3)$ is not contained in $(A - B)$'s **evat** in g_1 .

We now give details of the intersection operation of VATs for two graphs. There exist two types of intersections, corresponding to the two kinds of extension described earlier:

- **Forward Intersection:** This corresponds to the forward extension of an F_k graph with an F_1 i.e. addition of a new vertex and an edge. In Figure 5.4 g_1 is constructed by the extension of the F_k graph $(A - B)$ ($k = 1$) with the single edge $(B - C)$. $(A - B)$ is termed the parent of g_1 . It qualifies as a forward extension since C is a new vertex added to g_1 . Within the **evats** of these two edges, we now search for a common occurrence of B 's vid. We wish to find two pairs of vids (one each from $(A - B)$ and $(B - C)$) such that the second vid in pair from $(A - B)$ is equal to the first one in pair from $(B - C)$. If such a match is found, the pair denotes an occurrence of g_1 . In $GD[0]$, this is true for the vid pairs $(0, 1)$ and $(1, 2)$ with $vid(B) = 1$. This was a successful forward intersection, so we add the pairs $(0, 1)$ to $(A - B)$, and $(1, 2)$ to $(B - C)$ in $vat(g_1)$.
- **Backward Intersection:** Backward intersection leads to a cycle in the graph and corresponds to the backward extension e.g. g_3 in Figure 5.4. Backward extension adds an edge to the graph but no new vertices. g_3 was formed by the extension of g_1 with $(A - C)$, and the former is termed the parent of g_3 . Both A and C are part of g_3 , only the edge is a new addition. Furthermore,

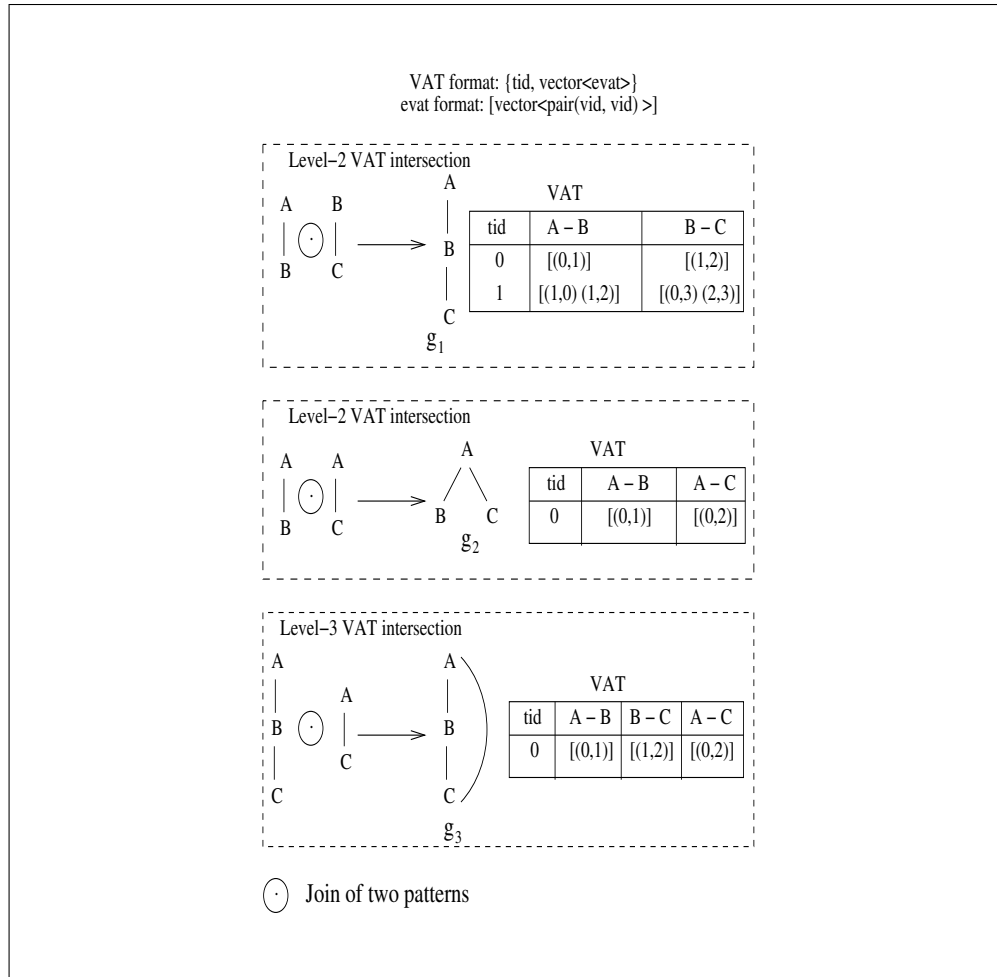


Figure 5.4: Graph VAT Intersection

the edge leads to a cycle in the graph. In backward intersection, we wish to determine if the new edge co-occurs with the given graph's parent. Towards this end, we search the *evats* of g_1 's $(B - C)$ and $(A - C)$ for a common *vid* of C . In $GD[0]$, a successful backward intersection is found with $vid(C) = 2$. Hence we add the corresponding *evats* to $(A - B)$ and $(B - C)$. It should be noted that in a backward extension, the rightmost path of the resulting graph is same as that of its parent. Hence, no new edge is added to the VAT.

Graph g_2 represents an example where the rightmost path of resulting graph is altogether different from the parent, $(A - B)$ due to branching. We are governed by rightmost extension here: since a new edge may be added only on the rightmost

path, we exploit it to retain `evats` for edges on rightmost path only.

Subgraph isomorphism is a known NP-complete problem [26], and the method we have proposed solves it by utilizing the additional information (the VAT) associated with each graph. Storing this information assists the subgraph isomorphism check, and circumvents the inherent complexity of the problem. Suppose g were an extension of g_k ; then i) g can be present in only those g_d for which $g_k \subseteq g_d$ ($g \subseteq g_d \Rightarrow g_k \subseteq g_d$), and thus ii) efficiently storing occurrences of g_k in GD shall circumvent the need to perform the costly subgraph isomorphism for g from scratch. Our VAT scheme detailed above is built upon these ideas.

CHAPTER 6

EXPERIMENTAL STUDY

Performance measurement and analysis is a critical to the success of any data mining algorithm. Real world data is usually voluminous and efficient techniques are needed to mine such large datasets. We analyze our toolkit's performance on the primary input parameters viz. minsup and database size. minsup is the minimum support threshold, and the size of a database is simply the number of transactions contained in it. We also compare DMTL's performance against stand-alone FPM implementations, where possible.

6.1 Experimental Setup

We ran our experiments on a Hyper-threaded Dual Pentium IV/2.8GHz machine, with 6GB of RAM and running Linux. Most data mining queries are processed on data warehouse servers, and hence this machine was deemed suitable for our purpose. The itemset and sequence datasets used were generated synthetically, and typically varied in size from 10,000 transactions (10K) to 100,000,000 transactions (100M) [27]. Tree datasets were generated using TreeGen [31] and graphs from the PAFI package [20]. All input datasets to our algorithms were flat file-based and no preprocessing was applied beforehand.

The results of our toolkit's algorithms were matched against a comparative algorithm for correctness, where applicable. All algorithms we deal with are complete algorithms, hence the main metric for comparison then is the mining time. Mining time is the wall clock time measured from start of the mining process to the end. Most of our experiments illustrate the variation of mining time against database size or minsup. Additionally, we can also analyze time taken by the main phases of the algorithm. In level-1 the algorithm reads and parses the input database, determines frequent length-1 patterns, and constructs their VATs. Hence this stage is I/O intensive. It is at this stage that the database is essentially converted from the native, horizontal format to a vertical one, based on VATs. Level-2 onwards,

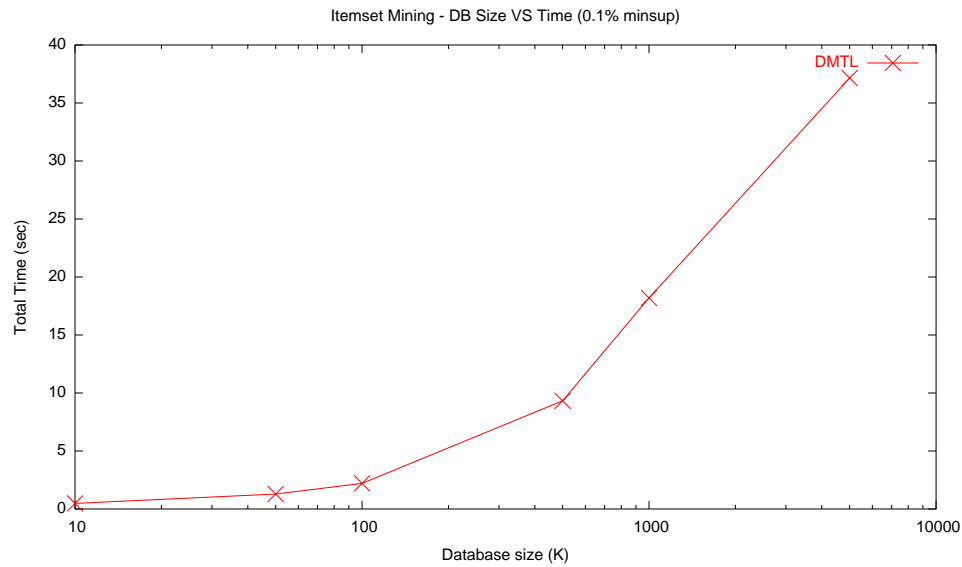


Figure 6.1: Effect of varying database size on performance of itemset mining

the algorithm determines support of patterns by VAT intersection. There is usually no I/O bound on the process after level-1. We now present results and analysis of DMTL’s itemset, sequence and tree mining components. At time of writing this thesis, the components of graph mining had been implemented but were not fully functional yet.

6.2 Itemset & Sequence Mining

Figure 6.1 plots the mining time for DMTL’s itemset mining on dataset sizes varying from 10K to 100M. The minsup was maintained was 0.1% of database size. A low minsup was chosen to test the robustness of the algorithm. The graph shows mining results on databasets from 10K to 5M in size. Table 6.1 shows the number of frequent patterns with increasing database size.

Figure 6.2 shows the mining time on the same dataset of size 1M, but for runs with varying minsup. The time taken increases non-linearly as minsup is decreased. This trend is justified by Table 6.2, in which we see that the number of frequent itemsets grows exponentially with decrease in minsup. Hence we can infer that the performance of the algorithm is closely tied in with the number of frequent patterns

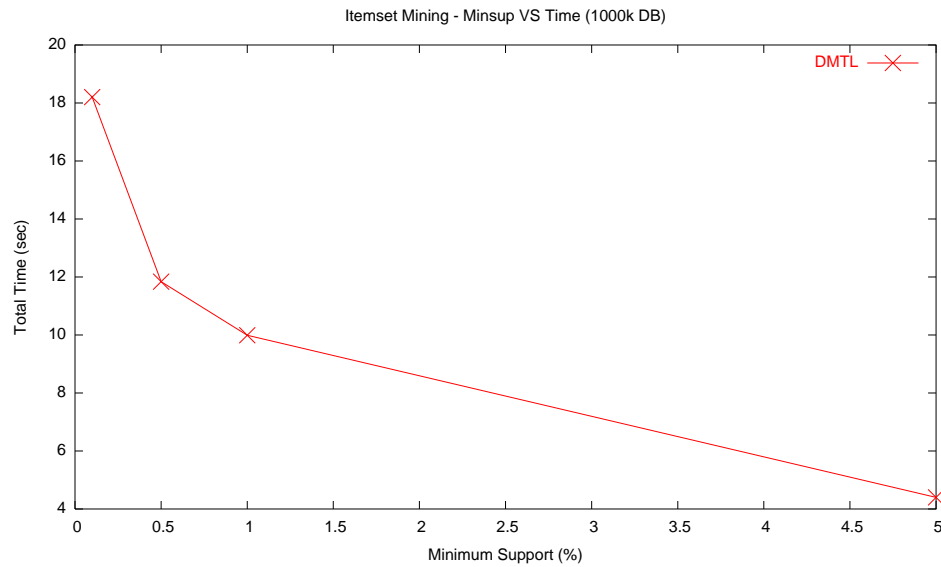


Figure 6.2: Effect of varying minsup on performance of itemset mining

DB Size	Frequent Patterns
10K	16638
50K	15417
100K	15993
500K	15807
1M	15837
5M	3779
10M	937
50M	106

Table 6.1: Effect of varying database sizes on number of frequent itemsets

discovered.

In Figure 6.3 we analyze the induced sequence mining component. The database size was varied from 10k to 500k, with the minsup maintained at 1% of DB size.

6.3 Tree Mining

We now analyze two tree mining variants available with DMTL - induced, unordered trees and embedded, ordered trees. DMTL is compared against SLEUTH [36], which performs a similar set of mining tasks, also in the vertical framework. In Figure 6.4, we present mining of induced, unordered Trees with database size

Minsup(%)	Frequent Patterns
0.1	15837
0.5	570
1	177
5	32

Table 6.2: Effect of varying minsup on number of frequent itemsets

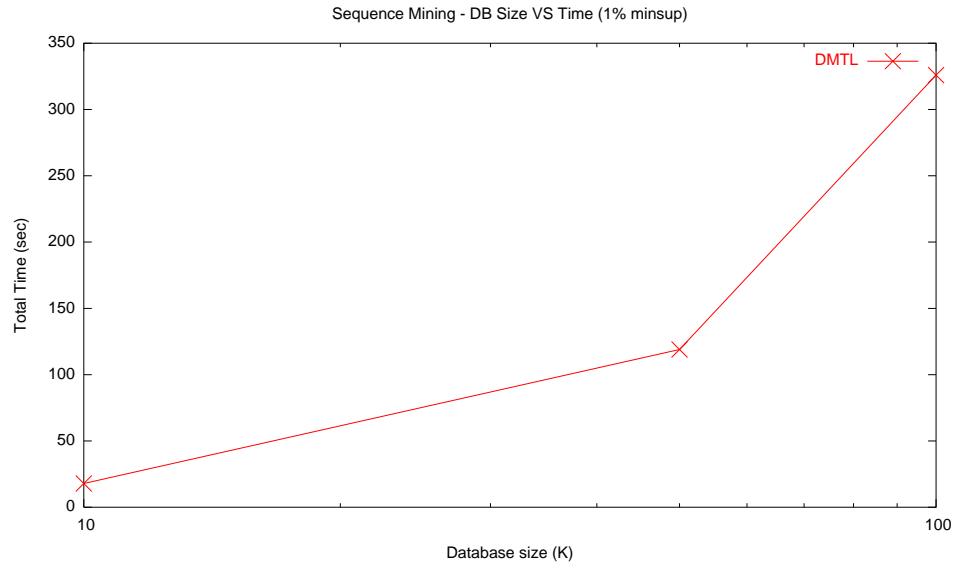


Figure 6.3: Effect of varying database size on performance of induced sequence mining

varying from 10K to 100M transactions. Minsup was maintained at 0.1% of DB size. As can be seen, DMTL fares almost as good as SLEUTH on these datasets.

Figure 6.5 compares induced unordered mining in DMTL and SLEUTH again: this time over different minsup for the same 1M dataset. In this experiment, DMTL outperforms SLEUTH.

Figures 6.6 and 6.7 present analogous plots for embedded, ordered mining. Here again we see that DMTL remains competitive to the stand-alone tree mining implementation. DMTL offers good scalability with large datasets, and remains robust for low minsup as well.

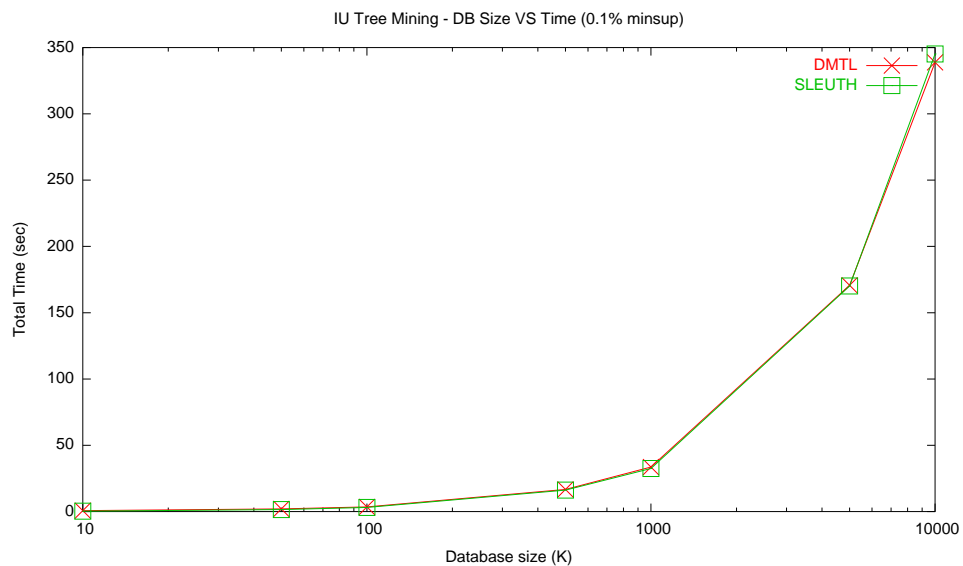


Figure 6.4: Comparison of DMTL with SLEUTH for mining induced, unordered trees over varying database sizes

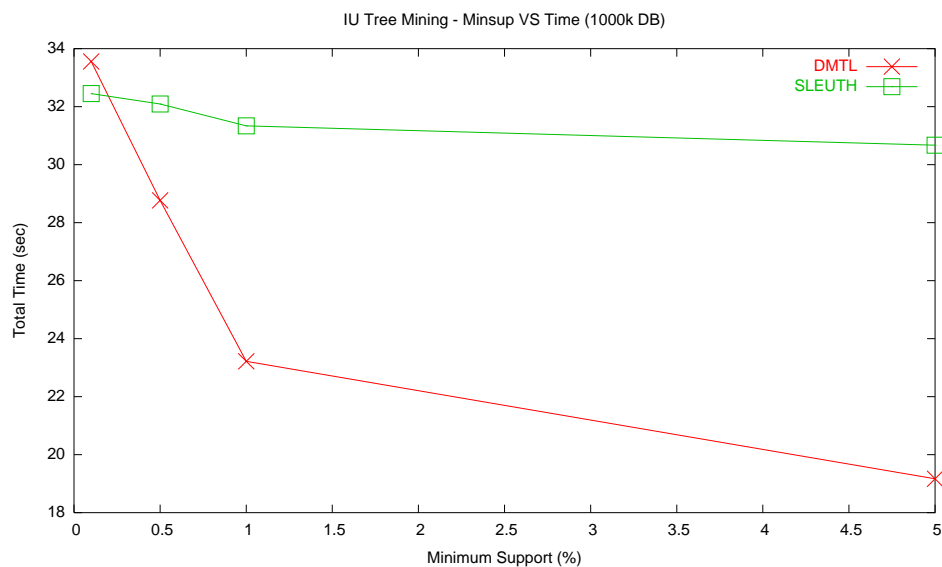


Figure 6.5: Comparison of DMTL with SLEUTH for mining induced, unordered trees for varying minsup

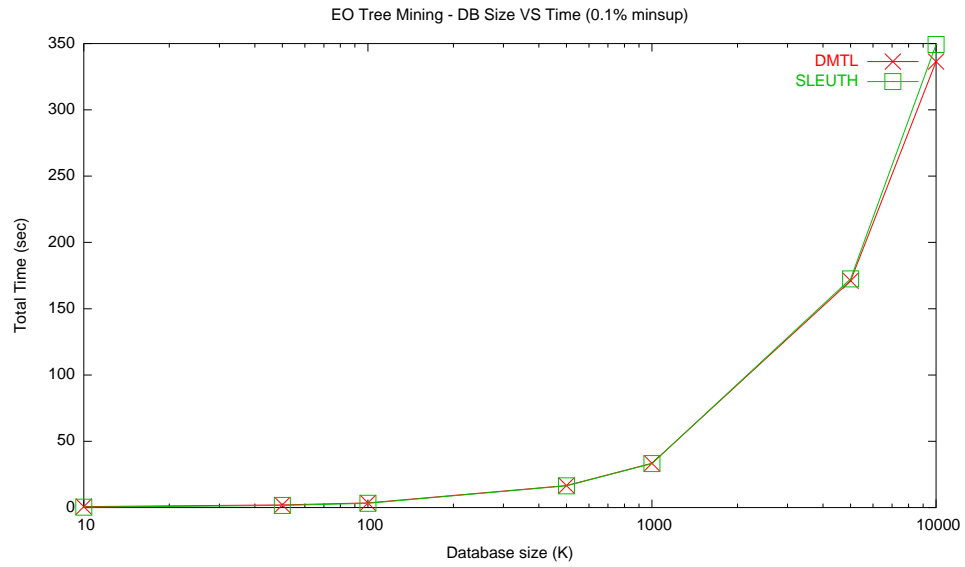


Figure 6.6: Comparison of DMTL with SLEUTH for mining embedded, ordered trees over varying database sizes

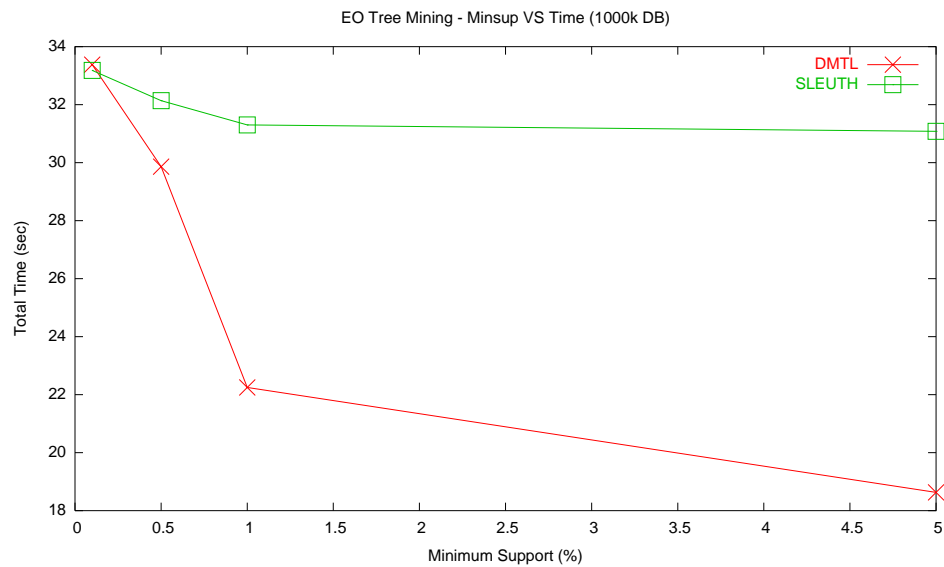


Figure 6.7: Comparison of DMTL with SLEUTH for mining induced, unordered trees for varying minsup

DB Size	Fraction of mining time taken by level-1
10K	0.235
50K	0.43
100K	0.47
500K	0.5
1M	0.517
5M	0.517
10M	0.519

Table 6.3: Fraction of mining time spent in level-1 with varying DB size for induced, unordered trees

6.4 Level-1

As stated earlier in the chapter, level-1 in our framework has two primary tasks: i) read and parse input database, and ii) generate frequent patterns and store their VATs. This startup cost is paid by any algorithm employing vertical mining. It would therefore be interesting to note what fraction of the total time level-1 computations take. Table 6.3 presents this analysis for induced, unordered tree mining. We compute the fraction of total mining time taken by level-1 only, and compare the value of the fraction for different database sizes.

Table 6.3 brings to attention two significant points: i) the fraction represents a substantial value, and ii) it increases with increasing database size. The second observation is particularly noteworthy. It implies that for large datasets, the I/O and VAT generation costs are significant. If we isolate these tasks into a *preprocessing* step, it could significantly speed up the mining process. Once a dataset is preprocessed, its VATs can be stored persistently. Any further mining queries can then be addressed by retrieving the required VATs and hence may reduce I/O costs.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

DMTL addresses an important subset of data mining problems - frequent pattern mining. We follow the ideology of generic programming in finding a unifying framework among all FPM tasks. DMTL attempts to find similarity amongst FPM algorithms based on the correspondence between inherent pattern types. We introduced the notion of properties with this motivation, and formulated a property hierarchy to identify relations between properties. We also demonstrated how this framework can be extended to new pattern types by the static type-list mechanism. Our implementation also has the desirable feature of upward propagation of properties, facilitating re-use of components. We discussed the core generic classes of the toolkit, as well as the specializations needed to perform specific tasks. It is hoped that DMTL will provide a common platform for developing new algorithms, and that it will enable equitable comparisons among the multitude of existing algorithms.

7.2 Future Work

DMTL is a work in progress, and is expected to undergo several changes shortly. We are working on enabling (undirected) graph mining in DMTL. Thereafter, extensions may be made towards mining digraphs and free trees. The second iteration (level-2) often represents a significant bottleneck for vertical mining algorithms. This may be overcome by performing horizontal mining for level-2. Extending the framework for new pattern types may be an interesting task in itself.

BIBLIOGRAPHY

- [1] Boost c++ libraries. <http://www.boost.org>.
- [2] Standard template library programmer's guide. <http://www.sgi.com/tech/stl>.
- [3] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD Conference on Management of Data*, 1993.
- [4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *21st Int'l Conference on Very Large Data Bases*, 1994.
- [5] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th Int'l Conference on Data Engineering*, 1995.
- [6] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Satamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *2nd SIAM Int'l Conference on Data Mining*, 2002.
- [7] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms : bagging, boosting, and variants. *Machine Learning*, 36:105–139, 1999.
- [8] P.A. Boncz. *Monet: A Next-Generation DBMS Kernel for Query-Intensive Applications*. PhD thesis, Univeriteit van Amsterdam, 2002.
- [9] Y.-L. Chen, H.-P. Kao, and M.-T. Ko. Mining dag patterns from dag databases. In *Web Age Information Management Conference*, 2004.
- [10] Y. Chi, Y. Yang, and R.R. Muntz. Indexing and mining free trees. In *3rd IEEE International Conference on Data Mining*, 2003.
- [11] Y. Chi, Y. Yang, and R.R. Muntz. Hybridtreeminer: An efficient algorithm for mining frequent rooted trees and free trees using canonical forms. In *16th*

International Conference on Scientific and Statistical Database Management, 2004.

- [12] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Second edition, 2001.
- [13] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. Wiley Interscience, Second edition, 2001.
- [14] T. Gschwind. Pstl-a c++ persistent standard template library. In *6th USENIX Conference on Object-Oriented Technologies and Systems*, 2001.
- [15] J. Han and M. Kamber. *Data Mining: Concepts and Principles*. Morgan Kaufmann, 2000.
- [16] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Conference on Management of Data*, 2000.
- [17] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. Technical Report TR03-021, University of North Carolina, 2003.
- [18] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *4th European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2000.
- [19] A.K. Jain and R.C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.
- [20] G. Karypis. Pafi software package. <http://www.cs.umn.edu/karypis/pafi>.
- [21] K. Knizhnik. Gigabase. <http://sourceforge.net/projects/gigabase>.
- [22] R. Kohavi, D. Sommerfield, and J. Dougherty. Data mining using mlc++, a machine learning library in c++. In *8th Int'l Conference on Tools with Artificial Intelligence*, 1996.

- [23] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *1st IEEE International Conference on Data Mining*, 2001.
- [24] D.R. Musser, G.J. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Second edition, 2001.
- [25] S. Nijssen and J.N. Kok. Efficient discovery of frequent unordered trees. In *1st Int'l Workshop on Mining Graphs, Trees and Sequences*, 2003.
- [26] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1996.
- [27] R. Srikant. Quest synthetic data generator.
<http://www.almaden.ibm.com/software/quest/Resources/datasets/syndata.html>.
- [28] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Int'l Conference Extending Database Technology*, 1996.
- [29] I.H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kauffman, 1999.
- [30] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. Technical Report UIUCDCS-R-2002-2296, University of Illinois at Urbana-Champaign, 2002.
- [31] M.J. Zaki. Treegen: Synthetic tree dataset generator.
<http://www.cs.rpi.edu/zaki/software>.
- [32] M.J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.
- [33] M.J. Zaki. Sequence mining in categorical domains: Incorporating constraints. In *9th Int'l Conference on Information and Knowledge Management*, 2000.
- [34] M.J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42:31–60, 2001.

- [35] M.J. Zaki. Efficiently mining trees in a forest. In *8th ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining*, 2002.
- [36] M.J. Zaki. Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae*, 66(1-2):33–52, 2005.
- [37] M.J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *9th ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining*, 2003.
- [38] M.J. Zaki and C.-J. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *2nd SIAM Int'l Conference on Data Mining*, 2002.
- [39] M.J. Zaki, N. Parimi, N. De, F. Gao, B. Phoophakdee, J. Urban, V. Chaoji, M.A. Hasan, and S. Salem. Towards generic pattern mining. In *International Conference on Formal Concept Analysis (Invited Paper)*, 2005.