Towards Generic Pattern Mining*

Mohammed J. Zaki**, Nagender Parimi, Nilanjana De, Feng Gao,
Benjarath Phoophakdee, Joe Urban, Vineet Chaoji,
Mohammad Al Hasan, and Saeed Salem

Computer Science Department,
Rensselaer Polytechnic Institute, Troy NY 12180

Abstract. Frequent Pattern Mining (FPM) is a very powerful paradigm for mining informative and useful patterns in massive, complex datasets. In this paper we propose the Data Mining Template Library, a collection of generic containers and algorithms for FPM, as well as persistency and database management classes. DMTL provides a systematic solution to a whole class of common FPM tasks like itemset, sequence, tree and graph mining. DMTL is extensible, scalable, and high-performance for rapid response on massive datasets. Our experiments show that DMTL is competitive with special purpose algorithms designed for a particular pattern type, especially as database sizes increase.

1 Introduction

Frequent Pattern Mining (FPM) is a very powerful paradigm which encompasses an entire class of data mining tasks. The specific tasks encompassed by FPM include the mining of increasingly complex and informative patterns, in complex structured and unstructured relational datasets, such as: Itemsets or co-occurrences [1] (transactional, unordered data), Sequences [2, 29] (temporal or positional data, as in text mining, bioinformatics), Tree patterns [30, 3] (XML/semistructured data), and Graph patterns [12, 16, 26, 27] (complex relational data, bioinformatics). Figure 1 shows examples of these different types of patterns; in a generic sense a pattern denotes links/relationships between several objects of interest. The objects are denoted as nodes, and the links as edges. Patterns can have multiple labels, denoting various attributes, on both the nodes and edges.

The current practice in frequent pattern mining basically falls into the paradigm of incremental algorithm improvement and solutions to very specific problems. While there exist tools like MLC++ [15], which provides a collection of algorithms for classification, and Weka [25], which is a general purpose

* This work was supported by NSF Grant EIA-0103708 under the KD-D program, NSF CAREER Award IIS-0092978, and DOE Early Career PI Award DE-FG02-02ER25538.

** We thank Paolo Palmerini and Jeevan Pathuri for their work on an early version of DMTL.

B. Ganter and R. Godin (Eds.): ICFCA 2005, LNCS 3403, pp. 1–20, 2005.

Java library of different data mining algorithms including itemset mining, these systems do not have an unifying theme or framework, there is little database support, and scalability to massive datasets is questionable. Moreover, these tools are not designed for handling complex pattern types like trees and graphs.

Our work seeks to address all of the above limitations. In this paper we describe Data Mining Template Library (DMTL), a generic collection of algorithms and persistent data structures, which follow a generic programming paradigm[4]. DMTL provides a systematic solution for the whole class of pattern mining tasks in massive, relational datasets. The main contributions of DMTL are as follows:

- Isolation of generic containers which hold various pattern types from the actual mining algorithms which operate upon them. We define generic data structures to handle various pattern types like itemsets, sequences, trees and graphs, and outline the design and implementation of generic data mining algorithms for FPM, such as depth-first and breadth-first search.
- Persistent data structures for supporting efficient pattern frequency computations using a tightly coupled database (DBMS) approach.
- Native support for both vertical and horizontal database formats for highly efficient mining.
- Developing the motivation to look for unifying themes such as right-most pattern extension and depth-first search in FPM algorithms. We believe this shall facilitate the design of a single generic algorithm applicable across a wide spectrum of patterns.

One of the main attractions of a generic paradigm is that the generic algorithms for mining are guaranteed to work for **any** pattern type. Each pattern is characterized by inherent properties that it satisfies, and the generic algorithm exploits these properties to perform the mining task efficiently. We conduct several experiments to show the scalability and efficiency of DMTL for different pattern types like itemsets, sequences, trees and graphs. Our results indicate that DMTL is competitive with the special purpose algorithms designed for a particular pattern type, especially with increasing database sizes.

2 Preliminaries

The problem of mining frequent patterns can be stated as follows: Let $\mathcal{N} = \{x_1, x_2, \ldots, x_{n_v}\}$ be a set of n_v distinct nodes or vertices. A pair of nodes (x_i, x_j) is called en edge. Let $\mathcal{L} = \{l_1, l_2, \ldots, l_{n_l}\}$, be a set of n_l distinct labels. Let $L_n : \mathcal{N} \rightarrow \mathcal{L}$, be a node labeling function that maps a node to its label $L_n(x_i) = l_i$, and let $L_e : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{L}$ be an edge labeling function, that maps an edge to its label $L_e(x_i, x_j) = l_k$.

It is intuitive to represent a *pattern* P as a graph (P_V, P_E), with labeled vertex set $P_V \subseteq \mathcal{N}$ and labeled edge set $P_E = \{(x_i, x_j) \mid x_i, x_j \in P_V\}$. The number of nodes in a pattern P is called its *size*. A pattern of size k is called a k-pattern, and the class of frequent k-patterns is referred to as F_k. In some applications P is a symmetric relation, i.e., $(x_i, x_j) \equiv (x_j, x_i)$ (undirected edges),

while in other applications P is anti-symmetric, i.e., $(x_i, x_j) \not\equiv (x_j, x_i)$ (directed edges). A path in P is a set of distinct nodes $\{x_{i_0}, x_{i_1}, x_{i_n}\}$, such that $(x_{i_j}, x_{i_{j+1}})$ in an edge in P_E for all $j = 0 \cdots n-1$. The number of edges gives the length of the path. If x_i and x_j are connected by a path of length n we denote it as $x_i <_n x_j$. Thus the edge (x_i, x_j) can also be written as $x_i <_0 x_j$.

Given two patterns P and Q, we say that P is a *subpattern* of Q (or Q is a super-pattern of P), denoted $P \preceq Q$ if and only if there exists a 1-1 mapping f from nodes in P to nodes in Q, such that for all $x_i, x_j \in P_V$: i) $L_n(x_i) = L_n(f(x_i))$, ii) $L_e(x_i, x_j) = L_e(f(x_i), f(x_j))$, and iii) $(x_i, x_j) \in P_V$ iff (if and only if) $(f(x_i), f(x_j)) \in Q_V$. In some cases we are interested in embedded subpatterns. P is an *embedded subpattern* of Q if: i) $L_n(x_i) = L_n(f(x_i))$, iii) $L_e(x_i, x_j) = L_e(f(x_i), f(X_j))$, and iii) $(x_i, x_j) \in P_E$ iff $f(x_i) <_l f(x_j)$ for some $l \geq 0$, i.e., $f(x_i)$ is connected to $f(x_j)$ on some path. If $P \preceq Q$ we say that P is contained in Q or Q contains P.

A database $\mathcal{D}$ is just a collection (a multi-set) of patterns. A database pattern is also called an *object*. Let $\mathcal{O} = \{o_1, o_2, \ldots, o_{n_o}\}$, be a set of n_o distinct *object identifiers (oid)*. An object has a unique identifier, given by the function $O(d_i) = o_j$, where $d_i \in \mathcal{D}$ and $o_j \in \mathcal{O}$. The number of objects in $\mathcal{D}$ is given as $|\mathcal{D}|$.

The *absolute support* of a pattern P in a database $\mathcal{D}$ is defined as the number of objects in $\mathcal{D}$ that contain P, given as $\pi^a(P, \mathcal{D}) = |\{P \preceq d \mid d \in \mathcal{D}\}|$. The *(relative) support* of P is given as $\pi(P, \mathcal{D}) = \frac{\pi^a(P, \mathcal{D})}{|D|}$. A pattern is *frequent* if its support is more than some user-specified minimum threshold, i.e., if $\pi(P, \mathcal{D}) \geq \pi^{\min}$. A frequent pattern is *maximal* if it is not a subpattern of any other frequent pattern. A frequent pattern is *closed* if it has no super-pattern with the same support. The frequent pattern mining problem is to enumerate all the patterns that satisfy the user-specified $\pi^{\min}$ frequency requirement (and any other user-specified conditions).

The main observation in FPM is that the sub-pattern relation $\preceq$ defines a partial order on the set of patterns. If $P \preceq Q$, we say that P is more general than Q, or Q is more specific than P. The second observation used is that if Q is a frequent pattern, then all sub-patterns $P \preceq Q$ are also frequent. More important is the converse, i.e. if P is infrequent and $P \preceq Q$ then Q shall also be infrequent (follows from the anti-monotonicity of frequency). The different FPM algorithms differ in the manner in with they search the pattern space.

2.1 FPM Instances

Some common types of patterns include itemsets, sequences, trees, and graphs, as shown in Figure 1. In fact, every pattern can be modeled as a graph; the nodes (x_i) are shown under each circle and the node labels ($L_n(x_i)$) are shown inside the circle, whereas edge labels have been omitted.

In an itemset [1] no two nodes have the same label. Let $V = \{x_1, x_2, \cdots x_k\}$ be a node set such that $L_n(x_i) \neq L_n(x_j)$ for all $x_i, x_j \in V$, and $L_n(x_i) < L_n(x_{i+1})$ for all $1 \leq i \leq k-1$. There are several possible formulation of the itemset pattern: i) *vertex-only*: An itemset pattern P is just a of vertices, i.e., $P_V = V$ and $P_E = \emptyset$, this is shown in Figure 1, ii) *linear*: in another formulation the

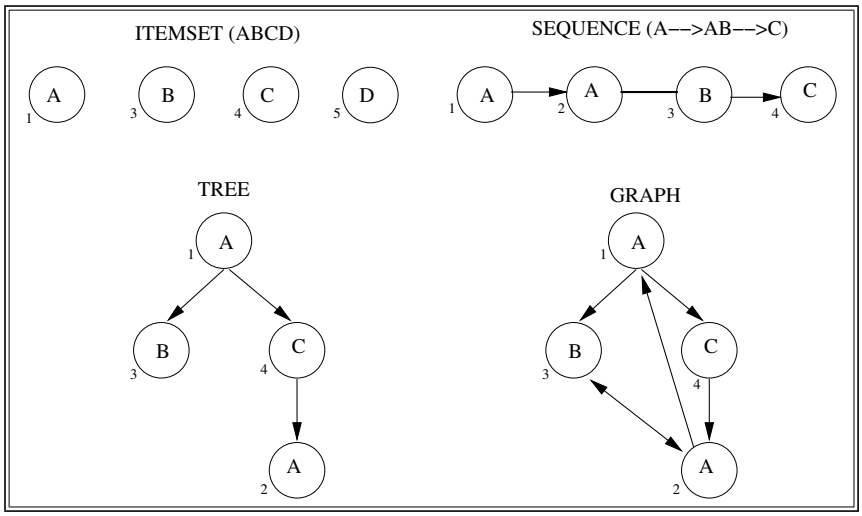


Fig. 1. FPM Instances

itemset is defined as $P_V = V$, and $P_E = \{(x_i, x_{i+1}) | x_i, x_{i+1} \in P_V\}$, iii) *clique*: A third alternative is to represent itemset P as a clique, i.e., $P_V = V$ and $P_E = \{(x_i, x_j) \mid i < j \text{ and } x_i, x_j \in P_V\}$.

In sequence mining [2], a sequence is modeled as an ordered list of itemsets, and thus the different nodes in a sequence can have the same label. We can model a sequence pattern P as being made up of a sequence of n itemsets P^i, $i = 1, \cdots n$, using the linear formulation (as shown in Figure 1); note that using the vertex-only formulation is problematic, since it results in a disconnected pattern. Thus P has a vertex set made up of n disjoint subsets $P_V = \bigcup_{i=1}^{n} P_V^i$. The edge set P_E contains all the edges within P^i (consecutive and undirected), and it also contains a directed edge for every pair of consecutive itemsets, i.e., from the last node of P^i to the first node of P^{i+1}.

In tree mining [30, 3], typically rooted, ordered and labeled trees are considered. Thus a tree pattern P consists of the vertex set $P_V = \{r, x_1, x_2, \cdots\}$, where r is a special node called root. A tree pattern must satisfy all tree properties, namely i) the root has no parent, i.e., $(x_i, r) \notin P_E$ for any $x_i \in P_V$, ii) the edges are directed, i.e., if $(x_i, x_j) \in P_E$, then $(x_j, x_i) \notin P_E$), iii) a node has only one parent, i.e., if $(x_i, x_j) \in P_E$, then $(x_k, x_j) \notin P_E$ for any $x_k \neq x_i$, iv) the tree is connected, i.e., for all $x_i \in P_V$, there exists a path from the root r to x_i, and v) tree has no cycles. Furthermore for ordered trees the order of a nodes' children matters. This means that there is an ordering of edges in P_E, such that (x_i, x_j) comes before (x_i, x_k) in P_E only if x_j is before x_k in the ordering of x_i's children. Embedded trees can be defined by following the definition of embedded patterns introduced earlier.

Finally, by definition a pattern can model any general graph, as well as any special constraints that might appear in graph mining [12, 16, 26], such as connected graphs, or induced subgraphs. It is also possible to model other patterns

such as DAGs (directed acyclic graphs). DMTL currently supports pattern mining of i) itemsets, ii) sequences, iii) embedded, rooted trees with ordered edges and iv) induced, undirected graphs with no single loops or multiple edges. As we shall soon see, the toolkit can be extended to incorporate mining of other user defined patterns as well.

2.2 Database Format

In a typical FPM task, the database is in the *horizontal* format i.e. a set of transactions, where each transaction is an object of the pattern type being mined [1]. Recently, *vertical* database formats have been proposed for mining itemsets, sequences and trees [28, 29, 30]. The vertical format is the more attractive alternative since it enables fast computation of supports by avoiding repeated database accesses. It does so by associating an entity called *Vertical Attribute Table*, VAT with each pattern. For an itemset, the VAT is the list of tids in which it is contained; VATs for sequences and trees are more complex and are described later. There currently does not exist a vertical scheme for graphs; the introduction of a new and efficient VAT scheme for graphs is one of our main contributions. DMTL introduces two modes of persistency: i) the collection of frequent patterns itself may be too large to fit in main memory, and hence persistent containers are provided to hold them, and ii) persistent storage and access to VATs. Both these modes of persistency are entirely transparent to the user.

3 DMTL: Data Structures and Algorithms

The C++ Standard Template Library (STL) provides efficient, generic implementations of widely used algorithms and data structures, which tremendously aid effective programming. Like STL, DMTL is a collection of generic data mining algorithms and data structures. In addition, DMTL provides persistent data and index structures for efficiently mining any type of pattern or model of interest. The user can mine custom pattern types, by simply defining the new pattern types, but the user need not implement a new algorithm - the generic DMTL algorithms can be used to mine them. Since the mined models and patterns are persistent and indexed, this means the mining can be done efficiently over massive databases, and mined results can be retrieved later from the persistent store.

Following the ideology of generic programming, DMTL provides a standardized, general, and efficient implementation of frequent pattern mining tasks by isolating the concept of data structures or containers, as they are called in generic programming, from algorithms. DMTL provides container classes for representing different patterns (such as itemsets and sequences) and collection of patterns, containers for database objects (horizontal and vertical), and containers for temporary mining results. These container classes support persistency when required.

Generic algorithms, on the other hand are independent of the container and can be applied on any valid container. These include algorithms for performing

intersections of the vertical lists [28, 29, 30] for itemsets, sequences or other patterns. Generic algorithms are also provided for mining itemsets, sequences and trees [1, 20, 28, 29], as well as for finding the maximal or closed patterns [11, 31]. Finally DMTL provides support for the database management functionality, pre-processing support for mapping data in different formats to DMTL's native formats, as well as for data transformation (such as discretization of continuous values). It should be noted that some of the algorithms designed for the C++ STL were inherently generic i.e. independent of the underlying datatype or container (e.g. `sort`). However devising a generic algorithm for FPM was a significant design challenge; we present it in Figure 3.

In this section we focus on the containers and algorithms for mining. In later sections we discuss the database support in DMTL as well as support for preprocessing and post-processing.

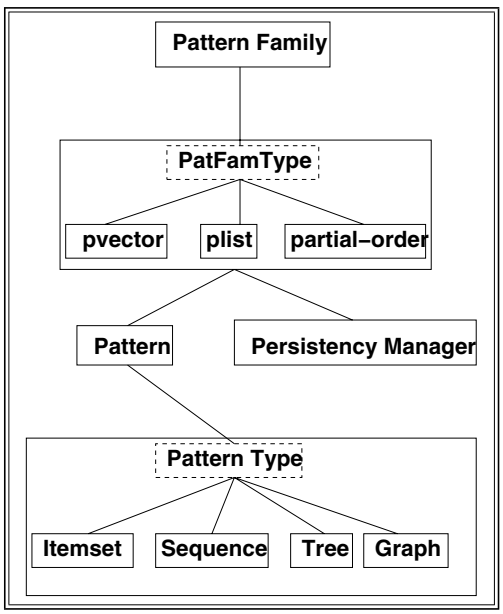


Fig. 2. DMTL Container Hierarchy

3.1 Containers

Figure 2 shows the different DMTL container classes for PMT (the Pattern Mining Toolkit) and the relationship among them. At the lowest level are the different kinds of pattern-types one might be interested in mining. A pattern is a generic container instantiated for one of the pattern-types. There are several pattern family types (such as pvector, plist, etc.) which together with a persistency manager class make up different pattern family classes. More details on each class appears below.

Pattern. In DMTL a pattern is a generic container, which can be instantiated as an itemset, sequence, tree or a graph, specified as `Pattern<class P>` by means of a template argument called Pattern-Type (`P`). A generic pattern is simply a Pattern-Type whose frequency we need to determine in a larger collection or database of patterns of the same type.

Pattern-Type. A pattern type is the specific pattern to be mined, e.g. itemset, and in that sense is not a generic container. DMTL has the itemset, sequence, tree and graph pattern-types defined internally; however the users are free to define their own pattern types, so long as the user defined class provides implementations for the methods required by the generic containers and algorithms. We shall later show how a new pattern type may be added to the library.

Pattern Family. In addition to the basic pattern classes, most pattern mining algorithms operate on a collection of patterns. The pattern family is a generic container `PatternFamily <class PatFamType>` to store groups of patterns, specified by the template parameter `PatFamType`. `PatFamType` represents a persistent class provided by DMTL, that provides seamless access to the members, whether they be in memory or on disk.

Pattern Family Type. This class provides the required persistency in storage and retrieval of patterns. DMTL provides several pattern family types to store groups of patterns. Each such class is templatized on the pattern-type (P) and a persistency manager class `PM`. An example is `pvector <class P, class PM>`, a persistent vector class. It has the same semantics as a STL vector with added memory management and persistency. Another class is `plist<P,PM>`. Instead of organizing the patterns in a linear structure like a vector or list, another persistent family type DMTL class, `partial-order <P,PM>`, organizes the patterns according to the sub-pattern/super-pattern relationship. While pvector and partial-order provide the same interface, certain operations will be more efficient in one class than the other. For example, inserts and deletions are cheaper for `plists`, while the maximality and closed testing functions will be cheaper for partial-orders, since the patterns are already organized according to sub/super-pattern relation.

3.2 Persistency Manager for Patterns

An important aspect of DMTL is to provide a user-specified level of persistency for all DMTL classes. To support large-scale data mining, DMTL provides automatic support for out-of-core computations, i.e., memory buffer management, via the persistency manager class PM. The PatternFamilyType class uses the persistency manager (PM) to support the buffer management for patterns. The details of implementation are hidden from PatternFamily; all generic algorithms continue to work regardless of whether the family is (partially) in memory or on disk. The interface of a persistent container (like pvector) is similar to that of a volatile container (like STL vector), hence encapsulating the implementation

behind the common interface. More details on the persistency manager will be given later.

3.3 Generic Mining Algorithms

The pattern mining task can be viewed as a search over the pattern space looking for those patterns that match the minimum support constraint. For instance in itemset mining, the search space is the set of all possible subsets of items. Within DMTL we attempt to provide a unifying framework for the wide range of mining algorithms that exist today. Figure 3 shows the pseudo-code for the generic mining algorithm, which was devised by combining the unifying aspects of mining itemsets, sequences, trees and graphs [28, 29, 30, 26]. Note that mining F_2 (i.e., level-2) often creates performance and memory bottleneck in FPM tasks, hence we employ a preemptive horizontal scan to accumulate estimated supports of level-2 patterns (line 3). This is an optimization intended for level-2 only, and we use the vertical approach thereon. The *extend* routine outlines the important tasks for mining any pattern: i) systematic candidate generation (line 8), ii) isomorphism checking (line 9) and iii) support counting which we accomplish through the vertical approach (lines 10-11). Partitioning frequent patterns into equivalence classes leads to a $F_k \times F_k$ candidate generation i.e. an F_{k+1} candidate is generated by *joining* two F_k sized patterns. It should also be noted that for graphs $g \in F_k$ implies g has k edges (not k nodes). Some of the salient features of our algorithm's design are:

Search Strategy. Several variants exist, depth-first search (DFS) and breadth-first search (BFS) being the primary ones. BFS has the advantage of providing better pruning of candidates but suffers from the cost of storing all of a given level's frequent patterns in memory. Recent algorithms for mining complex patterns like trees and graphs have focused on the DFS approach, hence it is the preferred choice for our toolkit as well. Nevertheless, support for BFS mining of itemsets and sequences is provided.

Vertical Mining. It has been shown that efficient vertical mining typically outperforms the horizontal approaches [28, 29, 30]. The vertical approach accomplishes fast support counting by intersection of VATs, thereby avoiding repeated database accesses. Section 4 gives details of the support we provide for vertical as well as horizontal mining.

Right-Most Extension. Recent algorithms towards solving tree and graph mining [30, 26] have focused on an approach of *right-most extension* i.e. a new node is added to the pattern only on the right most path from the root. This method has been shown to exhaustively enumerate all candidates for trees and graphs, and we believe that it can be augmented to work for itemsets and sequences as well. Though in the current framework the extension strategy is an internal component of each pattern's specialized routine, part of the proposed future work is devising a completely generic pattern mining algorithm, leveraging

aspects such as right most extension and depth-first search which are common across a wide range of patterns. We believe that developing the motivation to look for such unifying themes in pattern mining is one of the key contributions of this toolkit.

DMTL provides generic algorithms encapsulating these search strategies; by their definition these algorithms can work on any type of pattern: Itemset, Sequence, Tree or Graph. An example is the generic algorithm `DFS-Mine<class PatFamType> (PatternFamily<PatFamType> &pf, DB &db, ...)`, which mines the frequent patterns using a depth-first search (DFS) [28, 29]. The DFS algorithm in turn relies on other generic subroutines for creating equivalence classes, for generating candidates, and for support counting. There is also a generic `BFS-Mine` that performs Breadth-First Search [1, 20] over the pattern space.

dfs_mine (DB,result_pats):
1. F_1 = {level-1 frequent patterns}
2. result_pats = result_pats $\cup F_1$
3. F_2 = {optimized mining of level-2 patterns}
4. result_pats = result_pats $\cup$ F_2
5. F_2 = {partition F_2 into equivalence classes}
6. **for** each equivalence class $[P]_1$ in F_2 **do**
7. extend(DB, result_pats, $[P]_1$)

extend (DB, result_pats, $[P]$):
//DFS, equivalence class-based extension
6. $F_{k+1} = \emptyset$
7. $\forall$ patterns $P_i, P_j \in [P]$ such that $i \neq j$
8. new_pat = $P_i \bigodot P_j$ //generate new candidate
9. **if** new_pat.canonical_code is minimal **then**
//candidate has passed isomorphism test
10. new_pat.vat = $P_i.vat \bigotimes P_j.vat$ //vat intersection
11. **if** |new_pat.vat| $\geq$ minsup **then** //new_pat is frequent
12. result_pats = results_pat $\cup$ new_pat
13. $F_{k+1} = F_{k+1} \cup$ new_pat
14. F_{k+1} = {partition F_{k+1} into equivalence classes}
15. **for** each equivalence class $[P]_k$ in F_{k+1} **do**
16. extend(DB, result_pats, $[P]_k$)

Fig. 3. Generic DFS Pattern Mining

Figure 3 seeks to illustrate the major steps of `DFS-Mine`, our equivalence class-based vertical mining algorithm. The toolkit employs templates to provide for efficient compile time polymorphism based on the pattern type: the underlying algorithm stays the same but each distinct pattern has its specialized implementation of the key steps. For instance, the isomorphism check in line 9 is necessary only for graphs, and is omitted for other simpler patterns. Isomorphism checking

is achieved through the *canonical_code* member of each pattern. Each graph has a canonical code representation, and an ordering is defined on the code such that among all isomorphic graphs only one has the the least canonical code; all other graphs shall be discarded at line 9. DMTL applies the *DFS minimal code* of gSpan [26] but is not constrained by the choice of the canonical code. It is also to be noted that the equivalence class partitioning is omitted for graphs since $F_k \times F_1$ candidate generation does not lend itself easily to equivalence partitions.

3.4 Candidate Generation

We now provide a brief review of our extension routine ($\bigodot$) for the four primary pattern types, details of the VAT intersection follow later.

Itemset: Itemset join is the simplest and DMTL employs a vertical mining approach based on [28]. The join operation is defined on two itemsets Px and Py, belonging to the same equivalence class, $[P]$, which yields $Pxy \in [P_x]$.

Sequence: An equivalence class of sequences can comprise members which are sequence atoms ($P \rightarrow X$) or event atoms (PY). As described in [29], a join of two sequences within the same equivalence class $[P]$ can lead to one of three possibilities – i) joining PB with PD yields PBD (join of two event atoms); ii) join of PB with $P \rightarrow A$ results in $PB \rightarrow A$ (join of event atom with sequence atom) and iii) join of two sequence atoms, $P \rightarrow A$ with $P \rightarrow F$ leads to three outcomes: an event atom $P \rightarrow AF$ and two sequence atoms, $P \rightarrow A \rightarrow F$ and $P \rightarrow F \rightarrow A$.

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 [30] for elaboration on the prefix based representation scheme used for trees.

Graph: 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 [26]. An $F_k \times F_1$ 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. See [26] for more details.

3.5 Isomorphism Checking

Since a graph encompasses other simpler patterns (itemset, sequence, tree) we define the isomorphism problem for graphs: a graph p is isomorphic to q if there exists a mapping $M : p_v \rightarrow q_v$ such that for all $x_i, x_j \in p$, $L_p(x_i) = L_q(M(x_i))$

and $(x_i, x_j) \in p_e$ iff $(M(x_i), M(x_j)) \in q_e$. It has been shown that for itemsets, sequences and ordered trees the isomorphism checking may be averted by intelligent candidate generation, e.g., for the case of itemsets, AB and BA are isomorphic, but the algorithm can avoid generating BA by joining an itemset P_i only with a lexicographically greater itemset P_j (where both belong to the equivalence class $[P]$). Such schemes exist for sequences and ordered trees as well, but more complex patterns like unordered trees, free trees, directed acyclic graphs (DAGs) and generic graphs shall require some form of isomorphism testing.

Isomorphism Checking in Graphs: We follow the scheme outlined in [26] to achieve isomorphism checking for graphs. Based on a linear order on vertex and edge labels, a unique depth-first traversal is defined for any given graph. Each vertex in the graph is assigned a depth-first id, which is its order in the depth-first traversal. Each edge is represented by a 5-tuple (i, j, l_i, l_{ij}, l_j) where i is the DFS id of the first vertex of the edge and j of the second one, and l_i, l_{ij} and l_j are labels of the first vertex, the edge and second vertex respectively. Isomorphism checking is accomplished by defining an order on such 5-tuples.

4 DMTL: Persistency and Database Support

DMTL employs a back-end storage manager that provides the persistency and indexing support for both the patterns and the database. It supports DMTL by seamlessly providing support for memory management, data layout, high-performance I/O, as well as tight integration with database management systems (DBMS). It supports multiple back-end storage schemes including flat files, embedded databases, and relational or object-relational DBMS. DMTL also provides persistent pattern management facilities, i.e., mined patterns can themselves be stored in a pattern database for retrieval and interactive exploration.

DMTL provides native database support for both the horizontal [1] and vertical [28, 29, 30] data formats. It is also worth noting that since in many cases the database contains the same kind of objects as the patterns to be extracted (i.e., the database can be viewed as a pattern family), the same database functionality used for horizontal format can be used for providing persistency for pattern families. It is relatively straightforward to store a horizontal format object, and by extension, a family of such patterns, in any object-relational database. Thus the persistency manager for pattern families can handle both the original database and the patterns that are generated while mining. DMTL provides the required buffer management so that the algorithms continue to work regardless of whether the database/patterns are in memory or on disk.

4.1 Vertical Attribute Tables

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 properties or 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 property `color=red` stores all the transaction identifiers of cars whose color is red. The main advantage of VATs 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. As was mentioned earlier these kinds of vertical representations have proved to be useful in many data mining tasks [28, 29, 30].

In DMTL there is one VAT per pattern-type. Depending on the pattern type being mined the vat-type class may be different. Accordingly, their intersection (line 10, Figure 3) shall vary as well:

Itemset. For an itemset the VAT is simply a `vector <tid>`, where each tid may be stored as an `int`. VAT intersection in this case is straight forward, $new\_pat.vat = \{t | t \in P_i.vat \text{ and } t \in P_j.vat\}$, where $new\_pat = P_i \bigodot P_j$.

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 (refer to the section on sequence extension). 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 $P \rightarrow A \rightarrow F$ from its subsequences $P \rightarrow A$ and $P \rightarrow F$, one needs to match the tid *and* ensure that the time-stamp of A in that tid is less than that of F.

Tree. Define `triple` to 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 and out-scope tests corresponding to the two types of tree extensions described earlier [30].

Graph. The VAT for a graph is defined as a `vector<edge_vat>` where an `edge_vat` is defined as `vector<tid, vids>` where `vids` is a `vector <pair<int, int>>`. A graph may be viewed as a collection of edges; following this approach an `edge_vat` is in essence the VAT for an edge of a graph. It stores the tid of the graph in which the edge is present, and a collection of pair of vertex ids – each pair denoting an occurrence of the edge in that graph. Intersection of graph VATs is complicated due to isomorphism checking, and the details are beyond the scope of this paper.

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.

4.2 Storage and Persistency Manager

The database support for VATs and for the horizontal family of patterns is provided by DMTL in terms of the following classes, which are illustrated in Figure 4. `Vat-type` is a class describing the vat-type that composes the body of a VAT, for instance `int` for itemsets and `pair<int,time>` for sequences. `VAT<class V>` is the class that represents VATs. This class is composed of a collection of records of vat-type V. `Storage<class PM>` is the generic persistency-manager class that implements the physical persistency for VATs and other classes. The class PM provides the actual implementations of the generic operations required by Storage. For example, PM_metakit and PM_gigabase are two actual implementations of the `Storage` class in terms of different DBMS like Metakit [24], a persistent C++ library that natively supports the vertical format, and Gigabase [14], an object-relational database. Other implementations can easily be added as long as they provide the required functionality. `MetaTable<class V, class PM>` represents a collection of VATs. It stores a list of VAT pointers and the adequate data structures to handle efficient search for a specific VAT in the collection. It also provides physical storage for VATs. It is templatized on the vat-type V and on the `Storage` implementation PM. In the figure the H refers to a pattern and B its corresponding VAT. The `Storage` class provides for efficient lookup of a particular `VAT` object given the header. `DB<class V, class PM>` is the database class which holds a collection of `Metatables`. This is the main user interface to VATs and constitutes the database class DB referred to in previous sections. It supports VAT operations such as intersection, as well as

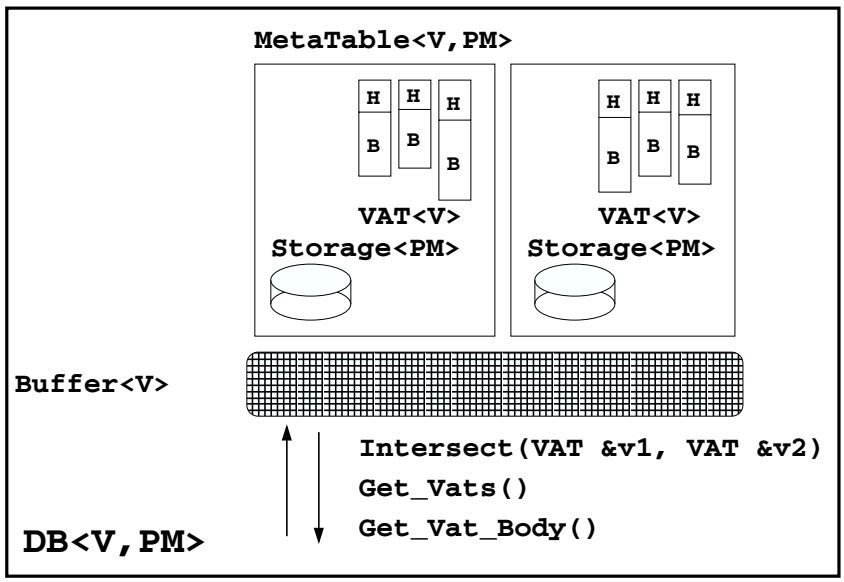


Fig. 4. DMTL: High level overview of the different classes used for Persistency

the operations for data import and export. The `DB` class is a doubly templated class where both the vat-type and the storage implementation need to be specified. An example instantiation of a `DB` class for itemset patterns would therefore be `DB<int,PM_metakit>` or `DB<int, PM_gigabase>`. `DB` has as data members an object of type `Buffer<V>` and a collection of `MetaTables<V,PM>`. `Buffer<class V>` provides a fixed-size main-memory buffer to which VATs are written and from which VATs are accessed, used for buffer management to provide seamless support for main-memory and out-of-core VATs (of type V). When a VAT body is requested from the `DB` class, the buffer is searched first. If the body is not already present there, it is retrieved from disk, by accessing the Metatable containing the requested VAT. If there is not enough space to store the new VAT in the buffer, the buffer manager will (transparently) replace an existing VAT with the new one. A similar interface is used to provide access to patterns in a persistent family or the horizontal database.

5 Extensibility of DMTL

DMTL provides a highly extensible yet potent framework for frequent pattern mining. We provide this flexibility via two central distinctions built into the library by design.

Containers and Algorithms. DMTL makes a clear delineation between patterns and the containers used to store them, and the underlying mining algorithm. This enables us to introduce the concept of a generic pattern mining algorithm, e.g., `dfs_mine`. The algorithms presented are the first step towards that end, and in our conclusions we outline the future challenges. We believe the benefits of a generic framework are at least two-fold: firstly, it provides a single platform for the field of frequent pattern mining and facilitates re-use of mining techniques and methodologies among various patterns, and secondly it may yield insight into discovering algorithms for newer patterns, e.g. DAGs.

Front-End and Back-End. We provide an explicit demarcation between the roles played by the containers and methods used by the actual mining algorithms (called the *front-end* operations) and those employed by the database to provide its functionality (*back-end* operations). FPM algorithms so far have mainly focused on a highly integrated approach between the front-end operations and back-end procedures. Though such an approach leads to efficient mining algorithms, it compromises on their extensibility and scalability. For instance, there is little support for persistency, buffer management, or even adding new DBMSs. DMTL addresses this issue by demonstrating a clean way of seamlessly integrating new pattern types, buffer management techniques or even support for a new DBMS. Furthermore, such a framework also enables us to define distinctly the roles played by its various components, especially in the vertical mining approach, e.g., a pattern need not be aware of its VAT representation at all, and this appeals intuitively too. A pattern is characterized completely by its definition only, and its VAT is an entity defined by us in order to achieve vertical mining.

This concept is again depicted cleanly in our toolkit - the pattern is aware of only the high-level methods `add_vat()` and `get_vat()`; it is not restricted by the specific VAT representation used.

This design enables DMTL to provide extensibility in three key ways.

Adding a New Pattern-Type. Due to the inherent distinction between containers and algorithms, a new pattern type can be added to DMTL in a clean fashion. We demonstrate how it may be extended for unordered, rooted trees [17]. The order of a node's children is relevant in ordered trees, while it is not so in unordered trees. We observe that DMTL already provides tree mining, hence much of the infrastructure may be re-used. The only significant modification required is isomorphism checking. Hence the user can define an unordered tree class, `utree`, similar to the in-built `tree` class. `utree` should provide an implementation for its `canonical_code` member, which our algorithm shall use to determine isomorphism. In addition, a vertical representation (VAT) needs to be provided for `utree`. Since `utree` is essentially a tree itself, it may utilize `tree`'s `vat_body` but needs to provide its distinct implementation of VAT intersection. In this instance, due to its similarity to `tree`, `utree` could utilize many of the common algorithms and routines. We acknowledge that this may not always be the case; nevertheless for a new pattern-type, the user needs to define specialized implementations of the main containers, viz., `utree` and `utree_vat` and their methods, but can reuse the toolkit's infrastructure for vertical/horizontal and DFS/BFS mining, as well as buffering and persistency. This way all algorithms are guaranteed to work with **any** pattern as long as certain basic operations are defined.

Buffering Scheme. The `Buffer` class provides memory management of patterns and VATs. A new buffer manager may be put in place simply by defining an appropriate new class, say `NFU_Buffer` employing a *not frequently used* strategy. `NFU_Buffer` should define methods such as `add_vat(vat_body&)` which shall implement the appropriate buffering of VATs. No other modification to the toolkit is necessary.

DBMS Support. The back-end DBMS and buffer manager are interleaved to provide seamless retrieval and storage of patterns and VATs. The buffer manager fetches data as required from the DBMS, and writes out excess patterns/VATs to the DBMS as the buffering strategy may dictate. In order to provide support for a new DBMS, appropriate methods shall have to be defined, which the toolkit would invoke through templatization of the `DB` class. Again, the design ensures that this new DBMS can be cleanly integrated into the toolkit.

6 Experiments

Templates provide a clean means of implementing our concepts of genericity of containers and algorithms; hence DMTL is implemented using the C++ Standard Template Library [4]. We present some experimental results on the time

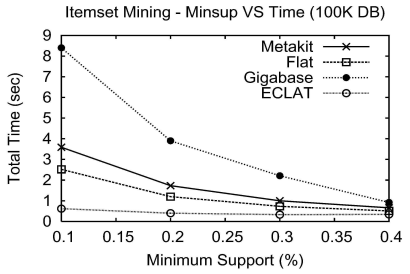
taken by DMTL to perform different types of pattern mining. We used the IBM synthetic database generator [1] for itemset and sequence mining, the tree generator from [30] for tree mining and the graph generator by [16], with sizes ranging from $10K$ to $500K$ (or 0.5 million) objects. The experiment were run on a Pentium4 2.8Ghz Processor with 6GB of memory, running Linux.

Figure 5 shows the DMTL mining time versus the specialized algorithms for itemset mining (Eclat [28]), sequences (Spade [29]), trees (TreeMiner [30]) and graphs (gSpan [26]). For the DMTL algorithms, we show the time with different persistency managers/databases: flat-file (Flat), metakit backend (Metakit) and the gigabase backend (Gigabase). The left hand column shows the effect of minimum support on the mining time for the various patterns, and the column on the right hand size shows the effect of increasing database sizes on these algorithms. Figures 5(a) and 5(b) contrast performance of DMTL with Eclat over varying supports and database sizes, respectively. As can be seen in, Figure 5(b), DMTL(Metakit) is as fast as the specialized algorithm for larger database sizes. Tree mining in DMTL (figures 5(e) and 5(f)) substantially outperforms TreeMiner; we attribute this to the initial overhead that TreeMiner incurs by reading the database in horizontal format, and then converting it into the vertical one. We have accomplished high optimization of the mining algorithm for itemsets and trees; proposed future work is to utilize similar enhancements for sequences and graphs. For graph and sequence patterns, we find that DMTL is at most, within a factor of 10 as compared to specialized algorithms and often much closer (Figure 5(d)). Overall, the timings demonstrate that the performance and scalability benefits of DMTL are clearly evident with large databases. For itemsets, another experiments (not shown here) reported that Eclat breaks for a database with 5 million records, while DMTL terminated in 23.5s with complete results.

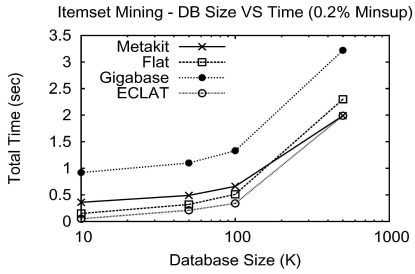
7 Future Work: Generic Closed Patterns

Our current DMTL prototype allows the mining of all frequent patterns. However, in the future we also plan to implement generic mining of other pattern spaces such as maximal patterns, and closed patterns. Informally, a maximal frequent pattern is a pattern which is not contained in another longer frequent pattern, whereas a closed frequent patterns is not contained in a longer frequent pattern which has the same frequency. We are especially interested in closed patterns since they form a lossless representation for the set of all frequent patterns.

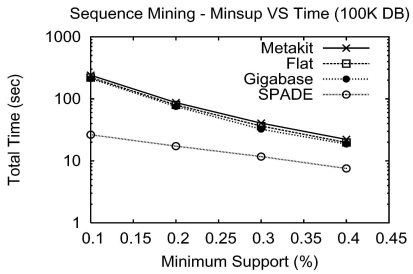
Mining closed patterns has a direct connection with the elegant mathematical framework of formal concept analysis (FCA) [9], especially in the context of closed itemset mining. Using notions from FCA one can define a closure operator [9] between the item ($\mathcal{N}$) and transaction ($\mathcal{O}$) subset spaces, which allows one to define a closed itemset lattice. This in turn provides significant insight into the structure of the closed itemset space, and has lead to the development of efficient algorithms. Initial use of closed itemsets for association rules was studied in [32, 18]. Since then many algorithms for mining all the closed sets have been proposed, such as Charm [31], Closet [19], Closet+ [22] Closure [8], Mafia [6] and Pascal [5]. More recent algorithms have been studied in [10].



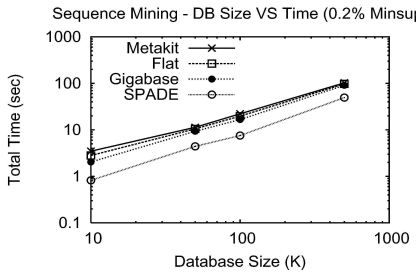
(a) Itemsets: Varying Minsup



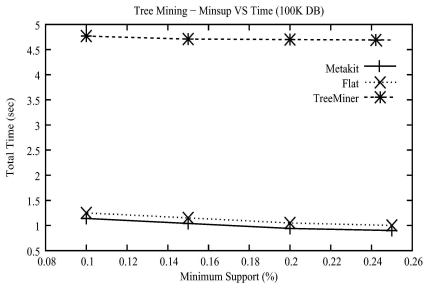
(b) Itemsets: Varying DB size



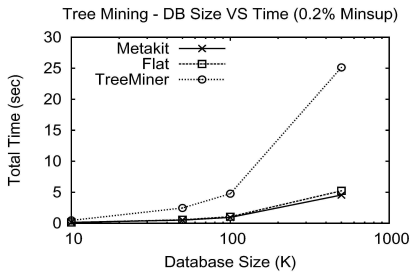
(c) Sequences: Varying Minsup



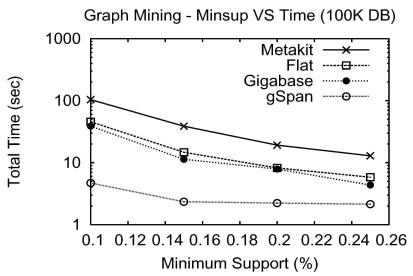
(d) Sequences: Varying DB size



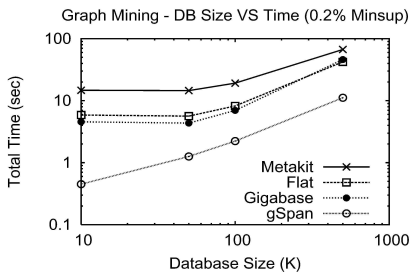
(e) Trees: Varying Minsup



(f) Trees: Varying DB size



(g) Graphs: Varying Minsup



(h) Graphs: Varying DB size

Fig. 5. Itemset, Sequence, Tree and Graph Mining: Effect of Minimum Support and Database Size

Recently, there has also been a surge of interest in mining other kinds of closed patterns such as closed sequences [23], closed trees [21, 7] and closed graphs [27]. For trees and graphs patterns there is currently no good understanding on how to construct the closure operator, and to leverage that to develop more efficient algorithms. The methods cited above use the intuitive notion of closed patterns (i.e., having no super-pattern with the same support) for mining. Recently, for ordered data or sequences, a closure operator has been proposed [13]. In our future work, we would like to develop the theory of a generic closure operator for any pattern and we will also develop generic data structures (e.g., `partial-order` pattern family) and algorithms to efficiently mine the set of all closed patterns.

8 Conclusions

In this paper we describe the design and implementation of the DMTL prototype for important FPM tasks, namely mining frequent itemsets, sequences, trees, and graphs. Following the ideology of generic programming, DMTL provides a standardized, general, and efficient implementation of frequent pattern mining tasks by isolating the concept of data structures or containers, from algorithms. DMTL provides container classes for representing different patterns, collection of patterns, and containers for database objects (horizontal and vertical). Generic algorithms, on the other hand are independent of the container and can be applied on any valid pattern. These include algorithms for candidate generation, isomorphism testing, VAT intersections, etc.

The generic paradigm of DMTL is a first-of-its-kind in data mining, and we plan to use insights gained to extend DMTL to other common mining tasks like classification, clustering, deviation detection, and so on. Eventually, DMTL will house the tightly-integrated and optimized primitive, generic operations, which serve as the building blocks of more complex mining algorithms. The primitive operations will serve all steps of the mining process, i.e., pre-processing of data, mining algorithms, and post-processing of patterns/models. Finally, we plan to release DMTL as part of open-source, and the feedback we receive will help drive more useful enhancements. We also hope that DMTL will provide a common platform for developing new algorithms, and that it will foster comparison among the multitude of existing algorithms.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.
2. R. Agrawal and R. Srikant. Mining sequential patterns. In *11th Intl. Conf. on Data Engg.*, 1995.
3. 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*, April 2002.

4. M. H. Austern. *Generic Programming and the STL.* Addison Wesley Longman, Inc., 1999.
5. Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *SIGKDD Explorations*, 2(2), December 2000.
6. D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: a maximal frequent itemset algorithm for transactional databases. In *Intl. Conf. on Data Engineering*, April 2001.
7. Yun Chi, Yirong Yang, Yi Xia, and Richard R. Muntz. Cmtreeminer: Mining both closed and maximal frequent subtrees. In *8th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2004.
8. D. Cristofor, L. Cristofor, and D. Simovici. Galois connection and data mining. *Journal of Universal Computer Science*, 6(1):60–73, 2000.
9. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer-Verlag, 1999.
10. B. Goethals and M.J. Zaki. Advances in frequent itemset mining implementations: report on FIMI'03. *SIGKDD Explorations*, 6(1), June 2003.
11. K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *1st IEEE Int'l Conf. on Data Mining*, November 2001.
12. A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *4th European Conference on Principles of Knowledge Discovery and Data Mining*, September 2000.
13. J.L.Balcazar and G.Casas-Garriga. On horn axiomatizations for sequential data. In *10th International Conference on Database Theory*, 2005.
14. Konstantin Knizhnik. Gigabase, object-relational database management system. http://sourceforge.net/projects/gigabase.
15. R. Kohavi, D. Sommerfield, and J. Dougherty. Data mining using mlc++, a machine learning library in c++. *International Journal of Artificial Intelligence Tools*, 6(4):537–566, 1997.
16. M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *1st IEEE Int'l Conf. on Data Mining*, November 2001.
17. Siegfried Nijssen and Joost N. Kok. Efficient discovery of frequent unordered trees. In *1st Int'l Workshop on Mining Graphs, Trees and Sequences*, 2003.
18. N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *7th Intl. Conf. on Database Theory*, January 1999.
19. J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD Int'l Workshop on Data Mining and Knowledge Discovery*, May 2000.
20. R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Intl. Conf. Extending Database Technology*, March 1996.
21. A. Termier, M-C. Rousset, and M. Sebag. Dryade: a new approach for discovering closed frequent trees in heterogeneous tree databases. In *IEEE Int'l Conf. on Data Mining*, 2004.
22. J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, August 2003.
23. Jianyong Wang and Jiawei Han. Bide: Efficient mining of frequent closed sequences. In *IEEE Int'l Conf. on Data Engineering*, 2004.
24. Jean-Claude Wippler. Metakit. http://www.equi4.com/metakit/.

25. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann Publishers, 1999.
26. X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *IEEE Int'l Conf. on Data Mining*, 2002.
27. X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. In *ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, August 2003.
28. M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372-390, May-June 2000.
29. M. J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning Journal*, 42(1/2):31–60, Jan/Feb 2001.
30. M. J. Zaki. Efficiently mining frequent trees in a forest. In *8th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, July 2002.
31. M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *2nd SIAM International Conference on Data Mining*, April 2002.
32. M. J. Zaki and M. Ogihara. Theoretical foundations of association rules. In *3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, June 1998.