Data Min Knowl Disc
DOI 10.1007/s10618-008-0098-x

An integrated, generic approach to pattern mining: data mining template library

Vineet Chaoji · Mohammad Al Hasan · Saeed Salem · Mohammed J. Zaki

Received: 8 May 2007 / Accepted: 29 April 2008

Abstract Frequent pattern mining (FPM) is an important data mining paradigm to extract informative patterns like itemsets, sequences, trees, and graphs. However, no practical framework for integrating the FPM tasks has been attempted. In this paper, we describe the design and implementation of the Data Mining Template Library (DMTL) for FPM. DMTL utilizes a *generic data mining* approach, where all aspects of mining are controlled via a set of *properties*. It uses a novel *pattern property hierarchy* to define and mine different pattern types. This property hierarchy can be thought of as a systematic characterization of the pattern space, i.e., a meta-pattern specification that allows the analyst to specify new pattern types, by extending this hierarchy. Furthermore, in DMTL *all* aspects of mining are controlled by a set of different *mining properties*. For example, the kind of mining approach to use, the kind of data types and formats to mine over, the kind of back-end storage manager to use, are all specified as a list of properties. This provides tremendous flexibility to customize the toolkit for various applications. Flexibility of the toolkit is exemplified by the ease with which support for a new pattern can be added. Experiments on synthetic and public dataset are conducted to demonstrate the scalability provided by the

Responsible editor: Hannu Toivonen.

V. Chaoji · M. Al Hasan · S. Salem · M. J. Zaki (✉)
Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY 12180, USA
e-mail: zaki@cs.rpi.edu

V. Chaoji
e-mail: chaojv@cs.rpi.edu

M. Al Hasan
e-mail: alhasan@cs.rpi.edu

S. Salem
e-mail: salems@cs.rpi.edu

persistent back-end in the library. DMTL been publicly released as open-source software (http://dmtl.sourceforge.net/), and has been downloaded by numerous researchers from all over the world.

Keywords Frequent pattern mining · Itemset mining · Sequence mining · Tree mining · Graph mining · Generic programming

1 Introduction

Frequent pattern mining (FPM) is an important data mining paradigm that helps to discover *patterns* that conceptually represent relations among discrete entities (or items). Depending on the complexity of these relations, different types of patterns arise. The most common type of patterns are sets, where the relation is the co-occurrence of items. A well-known example of a set pattern is a market-basket, the set of items that are bought together by a customer, say at a supermarket. Next, there are sequence patterns, where we require an ordering (temporal or positional) between items. Examples include time-series data in financial markets, genome sequence data in bioinformatics, etc. Data mining researchers also work with tree and graph patterns. In tree patterns the item relationship takes a hierarchical form, and in graph patterns the relationship is mostly arbitrary. Mining web log data, XML, or semi-structured data are examples of tree mining, and mining chemical compounds for drug discovery, or web communities in a web graph, are examples of graph mining. It is thus clear that different applications require the ability to define and mine different types of patterns; some may even require the ability to define custom pattern types (Horváth et al. 2006). All of these scenarios require efficient and flexible FPM algorithms and supporting data/index structures, which can be reused in a variety of domains.

It is worth noting that there exist open source data mining suites such as Weka (Witten and Frank 1999), which span commonly used data mining methods like association rules, clustering, classification, and regression. For the specific case of itemset mining, there also exist repositories of separate methods, such as Frequent Itemset Mining Implementations (FIMI) (Goethals and Zaki 2003). However, no unified framework for various FPM tasks currently exists.

The Data Mining Template Library (DMTL) is the first such effort in realizing a unified framework for mining various pattern types. DMTL is implemented in C++, which was chosen due to its support for generic programming. These features include template classes and functions, partial specialization, and so on. The most appealing part of C++ template features is that the binding is resolved at compile time, hence no runtime overhead is incurred. In contrast, while an object-oriented approach provides re-usability through inheritance, sometimes it suffers from runtime inefficiency, due to virtual table lookup for dynamic binding. In addition, most data mining libraries do not provide support for persistence. As a result they are restricted by the size of available main memory. For instance, Weka (Witten and Frank 1999) does not provide any mechanism to store intermediate results so that it can scale to much larger datasets. However, there has been some recent work to address this limitation (Zou et al. 2006).

The major contributions of our work are as follows:

- DMTL adopts the generic software development approach using C++ templates, inspired by the state-of-the-art generic libraries such as the C++ Standard Template Library (STL) (Musser et al. 2001) and Boost Graph Library (Siek et al. 2002), and hence provides widespread usability without compromising on efficiency.
- DMTL offers algorithms for different pattern mining tasks in a unified platform. Currently the library has implementations for mining four key patterns—itemsets, sequences, trees, and graphs.
- DMTL offers flexible interfaces for each of the algorithms, including each of its sub-tasks so that it is very simple for end users to use it as a library component in their software development.
- DMTL is extensible; new patterns can be mined with minimal effort from the end user. Users need to define some template parameters to ensure that the library selects the proper mining algorithm to mine that pattern successfully. Some additional specialized code may be required for efficiency reasons.
- In DMTL *all* aspects of mining are controlled by *properties*. *Pattern-properties* and *mining-properties* are used in DMTL to specify the following aspects of the mining process: (a) pattern to be mined, (b) input data source and format, (c) data structure to be used in the mining algorithm, (d) storage management, and (e) mining algorithm/approach.
- Support for multiple back-ends, which enables the library to scale to much larger datasets. The current implementation includes a file-based back-end that provides *transparent persistency* for mining out-of-core datasets.

The DMTL is available as open-source software from the world-wide Sourceforge repository (http://dmtl.sourceforge.net/). It has already been downloaded by numerous researchers and practitioners from all over the world.[1] Below, we describe the main elements of the design and implementation of DMTL. We explicitly show via an example, how the DMTL can be easily extended to mine custom defined patterns. For instance, we extend DMTL to mine multiset patterns. Finally, we perform extensive experimental studies to demonstrate the scalability of DMTL for mining various pattern types, and for mining very large, out-of-core datasets.

2 Related work

Frequent structure mining refers to an important class of exploratory mining tasks, namely those dealing with extracting patterns in massive databases representing complex interactions between entities. It encompasses mining techniques like itemsets (Agrawal et al. 1996), sequences (Agrawal and Srikant 1995), trees (Zaki 2002), and graphs (Inokuchi et al. 2003; Kuramochi and Karypis 2004). As one increases the complexity of the structures to be discovered, one extracts more informative patterns. Here we briefly review the existing methods for FPM.

[1] As of Nov 2007, DMTL has been downloaded by around 3,000 researchers from the Sourceforge site (it is averaging about 2,000 hits and 100 downloads a month).

Itemset mining The itemset mining problem is to discover frequently co-occurring sets of items (or attributes). Since its introduction by Agrawal et al. (1993), over the past decade many interesting algorithms have been proposed for mining frequent itemsets (Agrawal et al. 1996; Zaki et al. 1997; Savasere et al. 1995; Brin et al. 1997; Han et al. 2000a; Zaki and Gouda 2003). It continues to be an active area of research. Methods for mining maximal (those that have no frequent superset) and closed patterns (those that have no superset with the same frequency) have appeared in (Pasquier et al. 1999; Zaki and Hsiao 2002; Zaki and Hsiao 2005; Wang et al. 2003; Burdick et al. 2001a). Recent advances are described in the comparative study on Frequent Itemset Mining Implementations (FIMI) by Goethals and Zaki (2003).

Sequence mining Sequence mining helps discover frequent sequential patterns across time or positions in a given data set. Within data mining, the problem of mining sequential patterns was introduced by Agrawal and Srikant (1995). Many other approaches have followed since then (Srikant and Agrawal 1996; Mannila et al. 1995; Mannila and Toivonen 1996; Oates et al. 1997; Zaki 2001; Ayres et al. 2002; Pei et al. 2001). Methods that consider constraints like maximum/minimum gaps, sliding windows, regular expressions, and taxonomies have also been proposed (Srikant and Agrawal 1996; Zaki 2000b; Garofalakis et al. 1999). Methods for mining closed sequences appear in Wang and Han (2004), Balcazar and Casas-Garriga (2005).

Tree mining Several algorithms for tree mining have been proposed recently, starting with the earlier work in Wang and Liu (1998), Asai et al. (2002), and Zaki (2002). The new methods mine different kinds of tree patterns, such as ordered/unordered embedded trees (Zaki 2005b; Wang et al. 2004; Termier et al. 2002; Termier et al. 2004; Zaki 2005a) or induced trees (Chi et al. 2003; Shasha et al. 2004; Xiao et al. 2003; Nijssen and Kok 2003; Asai et al. 2003; Chi et al. 2004a). Methods for mining closed and maximal tree patterns appear in Chi et al. (2004b).

Graph mining Given a database of graph objects, the goal of graph mining is to find all the commonly occurring sub-graph patterns. Some of the early work in graph mining include Cook and Holder (1994), Yoshida and Motoda (1995), and Dehaspe et al. (1998). Many recent methods have been proposed which improve the efficiency of mining, these include Inokuchi et al. (2000), Kramer et al. (2001), Kuramochi and Karypis (2001), Yan and Han (2002a), Huan et al. (2003a), and Nijssen and Kok (2004). Closed graph mining methods have also been proposed (Yan and Han 2003).

As reviewed above, there have been many stand-alone algorithms to mine different types of patterns. On closer examination, certain common themes and common algorithmic paradigms permeate all of the existing methods. The goal of the DMTL is to abstract these common elements into generic primitives both in terms of the data structures used and in terms of the algorithms. In addition, DMTL explicitly handles the issue of persistency, i.e., the ability to mine out-of-core datasets. An initial version of DMTL was described in Zaki et al. (2004), however that design was not based on the property-based framework we present here. In a recent paper on DMTL (Hasan et al. 2005), we focused on the software design; neither did we emphasize the data mining aspects, nor did we present any experiments results. This paper describes the novel property-based DMTL framework, it shows how DMTL can be extended to mine new pattern types, and it presents a comprehensive experimental evaluation demonstrating DMTL's scalability to large datasets.

We would like to point out that the work by the authors in Mannila and Toivonen (1997), is similar in concept to that of ours. In Mannila and Toivonen (1997), the authors provide an abstract theoretical analysis for the problem of traversing the partial order in a levelwise fashion. In the pattern space, they mainly focus on the itemset and sequence patterns and their variations. The authors also discuss bounds on the number of evaluations of the "interestingness criterion" in terms of the size of the border. Our work lends an empirical grounding for their theoretical work. Moreover, our framework supports both levelwise and depthwise traversal techniques. We also explore more complex patterns such as trees and graphs.

3 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 an 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 $\mathcal{F}_k$. In some applications P is comprised of undirected edges, i.e., the edges define a symmetric relation: $(x_i, x_j) \equiv (x_j, x_i)$, while in other applications P is comprised of directed edges, i.e., the edges define an anti-symmetric relation: $(x_i, x_j) \not\equiv (x_j, x_i)$. A path in P is a set of distinct nodes $\{x_{i_0}, x_{i_1}, \ldots, x_{i_n}\}$, such that $(x_{i_j}, x_{i_{j+1}})$ is an edge in P_E for all $j = 0, \ldots, n-1$. x_{i_0} is the start node and x_{i_n} is the end node. The number of edges gives the length of the path. If x_i and x_j are connected by a path of exactly length n we denote it as $x_i <_n x_j$. Thus the edge (x_i, x_j) can also be written as $x_i <_1 x_j$.

Given two patterns $P = (P_V, P_E)$ and $Q = (Q_V, Q_E)$, let f: $P_V \rightarrow Q_V$ be an injective function. We say that P is a *sub-pattern* of Q (or Q is a super-pattern of P), denoted $P \preceq Q$ if and only if (iff) for all $x_i, x_j \in P_V$:

(i) Nodes labels are preserved by f, i.e., $L_n(x_i) = L_n(f(x_i))$.
(ii) Edge labels are preserved by f, i.e., $L_e(x_i, x_j) = L_e(f(x_i), f(x_j))$.
(iii) $(x_i, x_j) \in P_E \implies (f(x_i), f(x_j)) \in Q_E$, i.e., $P_E \subseteq Q_E$.

If $P \preceq Q$ we also say that P is contained in Q or Q contains P. Note that with the exception of itemsets, we are generally interested only in *connected sub-patterns*, where we require that there exists a path between x_i and x_j for all $x_i, x_j \in P_V$. Furthermore, in some data mining applications, we desire *embedded sub-patterns*; P is called an embedded sub-pattern of Q iff:

(i) Nodes labels are preserved by f, i.e., $L_n(x_i) = L_n(f(x_i))$.
(ii) $(x_i, x_j) \in P_E \implies f(x_i) <_l f(x_j)$ in Q_E, where $l \geq 1$. In other words, an edges (x_i, x_j) exists in P if $f(x_i)$ is connected to $f(x_j)$ by a path in Q. Note that if $l = 1$, then $f(x_i) <_1 f(x_j) \implies (f(x_i), f(x_j)) \in Q_E$. In this case

$P_E \subseteq Q_E$, and if we require that edge labels be preserved, then P becomes a *regular sub-pattern* of Q.

A database $\mathcal{D}$ is just a collection (a multi-set) of patterns. A database pattern is also called an *object* or a *record*. Let $\mathcal{R} = \{t_1, t_2, \ldots, t_{n_r}\}$, be a set of n_r distinct *transaction identifiers (tid)*. An object has a unique identifier, given by the function $R(d_i) = t_j$, where $d_i \in \mathcal{D}$ and $t_j \in \mathcal{R}$. 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 $\pi^{\min}$. A frequent pattern is *maximal* if it is not a sub-pattern 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).

3.1 FPM instances

Some common types of patterns include itemsets, sequences, trees, and graphs, as shown in Fig. 1. 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 no two nodes have the same label. Let $V = \{x_1, x_2, \ldots, x_k\}$ be a node set such that $L_n(x_i) \neq L_n(x_j)$ for all $x_i, x_j \in V$, and without loss of generality, we may assume that $L_n(x_i) < L_n(x_{i+1})$ for all $1 \leq i \leq k-1$. There are several possible graph representations for itemset patterns: (i) *vertex-only*: An itemset pattern P is just a set of vertices, i.e., $P_V = V$ and $P_E = \emptyset$, this is shown in Fig. 1, (ii) *linear*: in another formulation the itemset is defined as $P_V = V$, and $P_E = \{(x_i, x_{i+1}) | x_i, x_{i+1} \in P_V\}$,

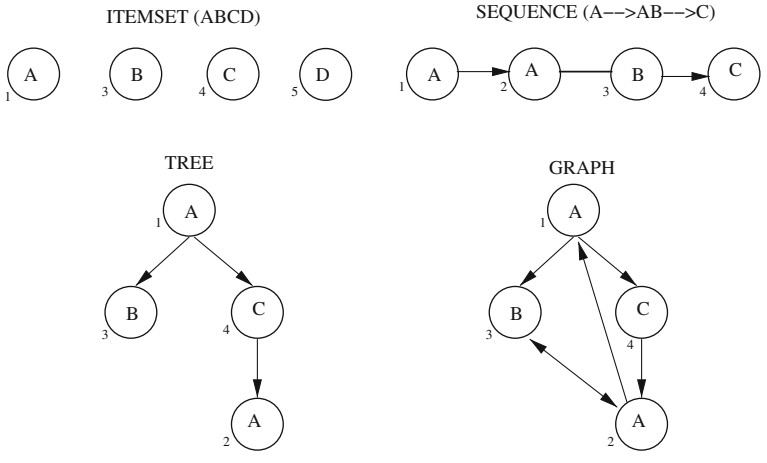


Fig. 1 FPM instances

(iii) *clique*: A third alternative is to represent itemset P as a clique, i.e., $P_V = V$ and $P_E = \{(x_i, x_j) \mid i < j \text{ and } x_i, x_j \in P_V\}$. All edges in itemsets are undirected.

A simple sequence pattern P is easily modeled as a set of directed edges $P_E = \{(x_i, x_{i+1}) | (x_i, x_{i+1}) \text{ is ordered, and } x_i, x_{i+1} \in P_V\}$. Moreover, for sequences, the node labeling function L_n need not be injective, i.e., two nodes can have the same label. The sequential patterns defined by Agrawal and Srikant (1995) are ordered lists of itemsets. We can model a sequential pattern P as being made up of a sequence of n itemsets $P^i, i = 1, \ldots n$, using the linear representation, with directed edges from one itemset to the next. For example, Fig. 1 shows the sequential pattern $A \rightarrow \{A, B\} \rightarrow C$. Note that using the vertex-only formulation for the itemsets in a sequential pattern would be problematic, since it would result in a disconnected pattern. In summary, a sequential pattern P has a vertex set made up of n disjoint subsets $P_V = \bigcup_{i=1}^{n} P_V^i$, and 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}.

A tree pattern P consists of the vertex set $P_V = \{r, x_1, x_2, \ldots\}$, where r is a special node called root. A tree pattern must satisfy the following 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. 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 general pattern can be modeled as a graph, along with any special constraints that typically arise in graph mining (e.g., connected graphs, or induced subgraphs). It is also possible to model other patterns such as directed acyclic graphs (DAGs) or free trees (undirected acyclic graphs). DMTL currently implements direct support for the mining of (i) itemsets, (ii) regular or embedded simple sequences, (iii) regular/embedded, rooted trees with ordered/unordered edges, and (iv) undirected graphs with no self loops or multiple edges. As we shall describe below, the toolkit can be extended to incorporate mining of other user defined patterns as well.

3.2 Database format

In a typical FPM task, the database is in a *horizontal* format, i.e. a set of transactions, where each transaction is an object of the pattern type being mined (Agrawal et al. 1996). Recently, *vertical* database formats have been proposed for mining itemsets, sequences, and trees (Zaki 2000a, 2001, 2002). The vertical format is an attractive alternative since it enables fast computation of support 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. Currently a vertical scheme does not exist for graphs; the introduction of a new and efficient VAT scheme

for graphs is one of the contributions of this paper. DMTL introduces two modes of persistence: (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 persistence are entirely transparent to the user.

4 DMTL design and implementation

4.1 Design motivation

While implementing mining algorithms for different patterns, we noticed that they exhibit considerable similarity, which suggested developing a common framework for implementing them. Figure 2 outlines a generic pattern mining algorithm (pseudo-code) that applies to all commonly explored patterns in the literature. The algorithm can be broken down into key sub-tasks, which include *candidate generation*, *isomorphism checking*, and *support counting*. By implementing generic algorithms for these sub-tasks, we retain the abstraction shown in this pseudo-code. The overall idea of the algorithm is as follows: the mining process searches incrementally in the pattern space by iteratively applying these sub-tasks in each iteration to enumerate patterns of size one, two, and so on. Each iteration discovers frequent patterns extended by one more item (or edge) than in the previous step, until no further frequent patterns exist in the database.

The sub-tasks of a generic mining algorithm can be developed using generic methods (expressed using C++ function templates). For example, the candidate generation method generates candidate patterns of generic type T, by combining two parent patterns of type T. The algorithm strictly requires that both the input arguments, together with the output argument, are of the same type T (e.g., we cannot join a set pattern with a tree pattern to produce a tree candidate pattern). The isomorphism checking method takes as input a pattern P of type T and produces a boolean value to indicate whether P is generated from a branch of the candidate generation tree where its canonical code is minimum. A minimum canonical code is a unique signature of a pattern, thus it guarantees that each candidate is enumerated only once. The support counting

Fig. 2 Generic frequent pattern mining algorithm

```
// [P] is a group of patterns (that share a prefix)
// DB is the database
// k is initialized to 0
Enumerate-Frequent-Patterns ([P], π^min):
1.  C_{k+1} = candidate_generation([P], DB)
2.  ∀ candidates c ∈ C_{k+1}
3.    if (isomorphism_checking(c)) then
4.      support_counting(c, DB)
5.      if (c.sup ≥ π^min) then
6.        F_{k+1} = F_{k+1} ∪ c
7.  for every group [P_i] ∈ F_{k+1}
8.    Enumerate-Frequent-Patterns([P_i], π^min)
```

method takes as input a pattern of type T, counts its frequency in the entire database (via sub-pattern, i.e., subgraph isomorphism, checking). Typically, the support of entire sets of candidates are determined in one database access.

In all the above three generic methods, the type T models a pattern concept. It has the following requirements: (i) T defines an object that relates some elements. (ii) T must adhere to a structure that is defined by a collection of relational properties. (iii) T defines a comparison ($\leq$) operator. (iv) Associated with type T there exists a pattern iterator, which is used to iterate through the elements of the pattern. All commonly known patterns in data mining, like set, sequence, tree, or graph are refinements of a pattern concept. The entire pattern mining process can be represented in terms of abstract objects and operations, that can be captured easily using the C++ template mechanism.

Figure 3 shows the key architectural components of DMTL. The components are partitioned into two main segments—the *front end* and the *back end*. The front end manages the core mining process while the back end provides the necessary storage support. The dotted rectangular block in the front end corresponds to the generic pattern mining algorithm shown in Fig. 2 and the three generic subtasks (candidate generation, isomorphism checking, and support counting) are represented by solid boxes inside it. The arrows in this figure show the data/control flow among different components. For instance, the down arrows among the tasks in the dotted rectangle indicate the order in which they were called and the loop back arrow (count support

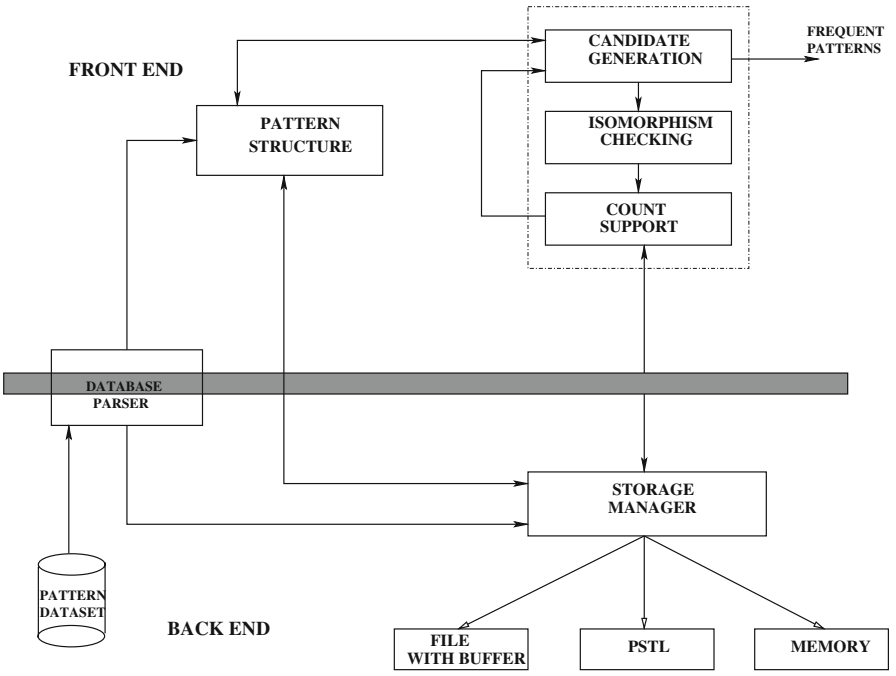


Fig. 3 High-level architecture diagram of the Data Mining Template Library. The boxes denote different functional units and the arrows connecting different boxes shows the data flow

to candidate generation) illustrates the recursive enumerative nature of a generic frequent pattern mining algorithm. The count support task delegates its responsibility to the back-end which manages the storage and can return the support of candidate patterns by efficient back-end operations (intersecting the VAT of the parent patterns). The separation between the front-end and the back-end, as shown in this figure, is explicitly retained in the implementation, which makes DMTL flexible, extensible, and also widely usable. Since the enumeration based algorithm can generate a large number of frequent patterns, efficient data structures are required such that the desired patterns and their corresponding VATs can be accessed efficiently from the container where they are stored. These data structures are implemented in the back-end, whose implementation is hidden to the front-end. The VAT for a pattern is accessed from the back-end via the *Pattern Structure* module. Note that direct access to VATs is not allowed (for modularity and abstraction). Instead pattern identifiers are passed to the back-end, which in turn computes the VAT for a candidate pattern and returns the support value to the front-end. The candidate pattern, if found frequent, is saved in the back-end along with its VAT. The *Database Parser* initiates the mining process by reading all frequent patterns of length one from the input source which could either be a database, a flat-file or another process generating the data. The *Database Parser* generates two objects for each level-one pattern—*pattern object* and the *VAT object* and stores those in the back-end storage manager.

4.2 The front-end mining engine

The front-end consists of the core mining engine which implements the enumerative mining process.

4.3 Data types

The key data structure in DMTL is the pattern, along with its associated VAT. The reader can more or less consider that the pattern is associated with the front-end and the VAT is associated with the back-end. This is because majority of the operations on the pattern structure are performed in the front-end, while the VAT is operated upon primarily in the back-end.

4.3.1 Expressing patterns as properties

Any pattern is conceptualized as a group of elements that can be represented as vertices of a graph. The relationship between the elements is captured by the presence of edges between the vertices. For instance, a set is a specialized graph which has no edges between its vertices. Whereas graphs provide an effective pattern abstraction, using a general purpose graph mining implementation to mine simpler patterns (itemsets, sequences or trees) is inefficient. The concept of *pattern property* provides an effective solution to this dilemma. With this idea of expressing patterns in terms of primitive properties, we can ensure that a generic algorithm can pick the most appropriate sub-tasks. In Sect. 4.3.3, we explain the use of these pattern properties to ensure

a generic algorithm that does not compromise efficiency via the use of appropriate pattern-specific specializations. Below we explain the different pattern properties that we used.

The pattern properties constrain the graph to form the desired pattern. We analyzed the pattern space and found that the following properties are sufficient to describe the most common patterns, but nevertheless, additional properties can be added to DMTL seamlessly. The properties are themselves categorized depending on the elements (nodes, edges, etc.) of a graph on which the constraints are imposed.

1. *Edge properties*: The edge set P_E is defined as $P_E \subseteq P_V \times P_V$, where P_V is the set of vertices in the pattern. Under edge relation category we consider the following properties:
   - *no-edge* means that elements in the patterns are not connected with any edge.
   - *directed* means that elements in the patterns are connected with directed (asymmetric) edges.
   - *undirected* means that elements in the patterns are connected with undirected (symmetric) edges.
   - *cyclic* means that at least one vertex is reflexive on the edge relation in the transitive closure of the pattern; otherwise, the pattern possesses the acyclic property.
2. *Vertex properties*: Here we consider only one property, *ordered*, which imposes an ordering on the neighbors of a vertex, or else the pattern is said to be unordered. Ordering is usually relevant for only tree patterns.
3. *Degree-related properties*: These relate to the degree constraints placed on the nodes of the pattern.
   - *indegree_lte_one* constrains all vertices of a graph to have indegree ≤ 1.
   - *outdegree_lte_one* constrains all vertices of a graph to have outdegree ≤ 1.
4. *Label properties*: Here the *unique_label* 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).

Figure 4 shows the relationship among the different pattern types, based on the pattern properties, which define a concept lattice over the property space and pattern types. Each node of the lattice defines a formal concept (Ganter and Wille 1999), which consists of an intent and extent pair. The extent spells out the maximal set of pattern types that share the properties noted in the intent. Likewise, the intent consists of the maximal set of properties that apply to the patterns in the extent. The lattice of pattern property concepts is shown in Fig. 4 using minimal labeling (Ganter and Wille 1999). For each node, its extent is made up of all nodes reachable below it, and its intent is made up of all nodes reachable above it. For example, the minimal concept (`{Indegree <= 1}, {Unordered Tree}`) represents the full concept (`{Directed, Acyclic, Indegree<= 1},{Sequence, Ordered Tree, Unordered Tree}`).

Concept refinement is the process of obtaining a sub-concept from a concept. Adding one or more properties in the intent removes patterns from the extent that do

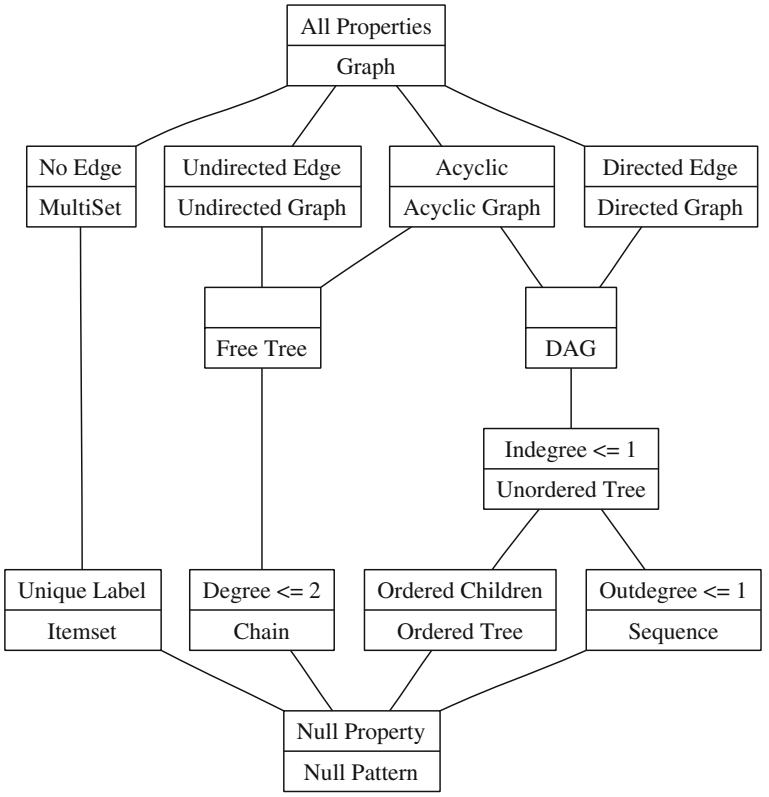


Fig. 4 Pattern property concept lattice and pattern types. Each node shows a pattern property and the specific pattern type defined. Each node inherits all the properties from nodes above it. Free Trees and DAGs do not define any new properties; Free Tree is an undirected, and DAG is a directed, acyclic graph

not conform to that property. For example, we can refine the (`{Indegree ≤ 1}, {Unordered Tree}`) concept by adding another property `Outdegree <= 1`, to yield the concept (`{Directed, Acyclic, Indegree <= 1, Outdegree <= 1}, {Sequence}`).

In our generic library implementation, we employ the formal concept hierarchy to develop mining algorithms that can handle different types of patterns. Any algorithm that works for patterns in a concept automatically works for the patterns in a sub-concept (i.e., those nodes below it). The list of pattern properties in a concept is passed as a template argument, which partially matches and automatically invokes the correct algorithm for the patterns belonging to the concept. There are two possible cases: (i) An efficient implementation exists for the current concept, in which case, the properties will match that implementation, or (ii) An implementation of that concept does not exist, in which case, the template will match some super-concept which has an implementation. For example, assume we want to mine DAG patterns, but no specific implementation exists for mining DAGs. In this case, DMTL will automatically use the implementation for general graph mining (assuming that DMTL also does not have specific implementations for acyclic or directed graphs).

```
template<
    class pattern_props,
    class canonical_code,
    class graph_model>
class pattern {
   public:
      typedef vector<V_TYPE> VERTICES;
      typedef typename VERTICES::const_iterator CONST_VIT;

   bool add_vertex(const V_TYPE& v);
   bool add_edge(const V_TYPE& src, const V_TYPE& dest,
                                    const E_TYPE& e_lbl);
   CONST_VIT get_neighbors(const V_TYPE& v);
   CONST_VIT get_rmost_path();
};
```

Fig. 5 Pattern class interface

4.3.2 Pattern container implementation

As mentioned earlier, in DMTL, vertices and edges are the basic structural building blocks of every pattern. The most basic interface for a pattern should thus provide methods for adding labeled vertices and edges between vertices. Figure 5 shows the C++ class interface of the pattern concept. It consists of the most basic operations required to construct a complex pattern. It also shows the template arguments that we used to construct the pattern concept. The first argument `pattern_props` lists the pattern properties that define a specific pattern. The second argument `canonical_code` maintains a pattern signature and is used for isomorphism checking. It is also used to implement the comparison ($\leq$) operator for the pattern. The last argument `graph_model` is the underlying data structure used for storing the pattern structure. A typical example of such a data structure is an adjacency list. This design decision to parameterize the pattern data structure aims at decoupling the pattern storage from the pattern concept, such that an adjacency list based storage could be substituted by a sparse adjacency matrix structure. From the above interface, a sequence such as $A \rightarrow B$ can be constructed by invoking the *add_vertex(A)* method followed by the *add_vertex(B)* and *add_edge(A, B, e)* methods (*e* is some label).

As seen in Fig. 4, the specific patterns are instantiations of the abstract pattern concept. We identify each such concept by a set of properties (or constraints) that define the pattern. For instance, a directed acyclic graph (as the name suggests) has `{acyclic,directed}` as its property set. The notion of having a set of properties to represent a concept is crucial for the implementation of our library. Even though conceptually the properties are considered to be a set, for implementation we treat them as an ordered list of properties. This ordering of properties is necessary for the compiler to match a specialized pattern to an appropriate super-pattern, if any algorithmic implementation is not available for that specialized pattern. This leads to the pattern hierarchy tree in Fig. 6. Note that in Fig. 4, a node can have multiple parents whereas in Fig. 6 each pattern has a single parent. Multiple parents for a pattern would lead to an ambiguity when a super-pattern functionality has

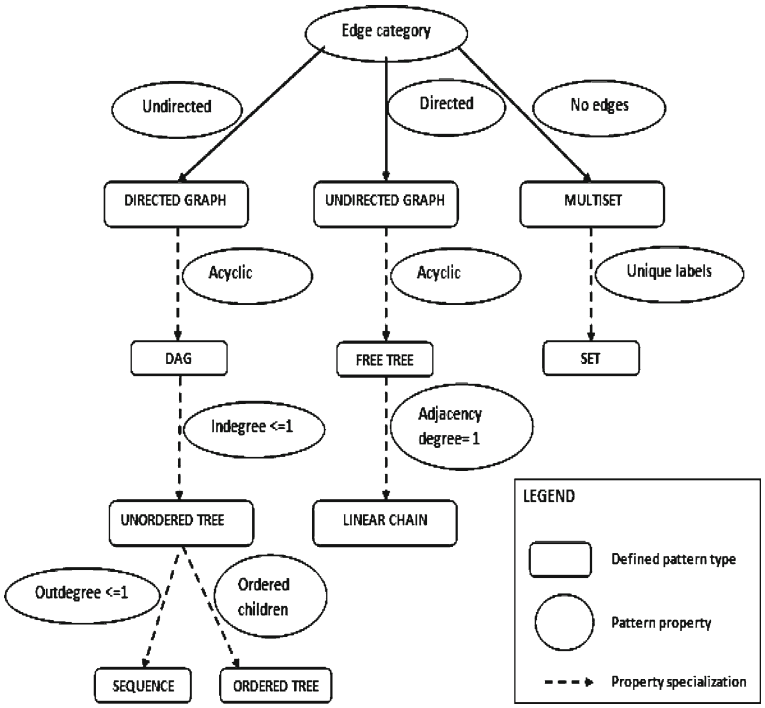


Fig. 6 Pattern hierarchy

to be invoked. Using the pattern hierarchy tree, the ordering of the properties for a pattern is automatically enforced. They are ordered along the path from the root to a pattern node. In a nutshell, Fig. 4 represents the conceptual relationships among the patterns, whereas Fig. 6 represents the practical implementation in terms of an (unambiguous) hierarchy. We had the following goals while implementing the hierarchy of patterns:

1. Abstract out the common aspects between the pattern types and the algorithms.
2. Allow new patterns to be added to the hierarchy by introducing new properties.
3. Propagate absence of a lower-level concept implementation to a higher-level concept implementation.

The last objective above is an outcome of using partial specialization (via function template overloading). The presence of a single parent in the hierarchy tree enables finding the right pattern to which control should be dispatched. Our library provides implementations for what we call the four core patterns—sets, sequences, trees, and undirected graphs. Apart from being the most popular patterns, the core patterns can informally be considered to mark complexity classes in frequent pattern mining. Sets are at the simpler end of the spectrum with sequences and trees (in that order) before graphs at the other extreme.

4.3.3 Pattern properties implementation

In order to identify the appropriate pattern we use the set of pattern properties as template parameters. This set of pattern properties is encapsulated in a class `proplist`, which is an ordered list of properties. Since it is simply a holder of types, the class itself is not complicated and is given below.

```
1  template<
2   class prop,
3   class next_property=null_prop>
4  class proplist{
5   public:
6     typedef prop FIRST;
7     typedef next_property SECOND;
8  };
```

The class `null_prop` is used as a delimiter in the type-list. 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 run-time overhead. A type-list gives us the flexibility to append properties to it, making the design generic and extensible. In addition to its utility as a type list, the `proplist` possesses the nice feature of facilitating upward propagation of properties. Pattern properties also play an important role in introducing pattern mining for a new pattern. These aspects are discussed in further detail in Sect. 6.

4.3.4 Mining properties

Section 4.3.3 focused on one aspect of enabling genericity; using `proplist` to define patterns. The generic behavior of pattern mining algorithms that DMTL advocates is not restricted to the types of patterns it mines. This generic behavior extends to the different ways in which algorithms perform the sub-tasks in Fig. 2. The choices are captured using the *mining properties*; the user can choose a collection of such mining properties to select the kind of mining algorithm to run. In this way, our generic software implements a family of algorithms and gives the user the freedom to choose one that best suites the application at hand. The mining properties are independent of the pattern properties. Analysis of existing FPM algorithms reveals the following mining properties (as with pattern properties, new mining properties can be incorporated):

1. *Join-type*: This category influences the candidate generation phase, in which potential frequent patterns are generated. During candidate generation, the algorithm typically constructs a new pattern by *joining* two parent patterns. The nature of this join is a property itself. A suitably correct algorithm has to be provided for the chosen property:
   - $\mathcal{F}_{\mathbf{k}} \times \mathcal{F}_{\mathbf{1}}$: Here a $(k+1)$-length pattern is constructed by joining a k-length pattern with a pattern of size one (i.e., a single element).

- $\mathcal{F}_{\mathbf{k}} \times \mathcal{F}_{\mathbf{k}}$: Here a $(k+1)$-length pattern is constructed by joining two k-length patterns. 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 common approaches are:
   - *horizontal*: This indicates that the support for a candidate pattern is determined by counting its occurrences in the database, testing for the sub-pattern relationship against each database object. This method usually involves significant I/O overhead for large databases.
   - *vertical*: In this approach, support for a pattern is determined by defining an appropriate *join* (or intersection) operation over the vertical attribute tables of the two parent patterns.
3. *Transitivity*: This property controls the inclusion of either induced or embedded occurrences of a pattern in its support counting step. One might think that this property should belong to the set of pattern properties (Sect. 4.3.1), however, a pattern itself is unaware of its occurrences in the database, since it can have both induced and embedded occurrences within a database object. The distinction between an induced and and embedded *occurrence* of a pattern within database objects is entirely up to the support counting phase of the algorithm. As a result, this property is included under mining properties.
   - *induced*: When transitivity is not considered, only regular sub-pattern occurrences are counted.
   - *embedded*: For embedded patterns, transitive closures on the edge relation are included in the support counting. The transitivity leads to discovery of embedded occurrences (see Sect. 3) of the pattern.

4.4 The back-end storage manager

As mentioned earlier, the enumerative pattern mining algorithm generates new patterns and the associated VATs dynamically. While mining large datasets, it is most likely that the newly generated patterns will not fit in the main-memory (especially, for low minimum support values). Most mining algorithms do not provide explicit means of memory management nor is the issue addressed within the algorithm. The DMTL back-end manages the storage of patterns and their VATs; it allows constant time access to a pattern and its corresponding VAT through a pattern-key (which is a hash-value computed on the pattern). The back-end is also the facilitator for the front-end to determine the support of a candidate pattern.

We designed DMTL back-end to be similar to that of an STL allocator. Each allocator implements a different memory management strategy. Moreover, each allocator provides the same interface, thus hiding the details from the user. We currently implement three allocator classes—the mining algorithm can choose any of them, by using the allocator name as a template argument in the *Count Support* module. A new allocator can be introduced without affecting the remaining modules of

the system since the mining algorithms are independent of the allocator. In Fig. 3, *Storage Manager* is the functional block that represents a generic allocator. The three allocators implemented in DMTL are shown as specializations of the Storage Manager.

Out of the currently implemented allocators, pure memory-based one is naturally the most time efficient, as long as the data structures fit in memory. As expected, it does not scale to larger datasets. The persistent allocator (PSTL) (Gschwind 2001) is based on memory-mapped files, but does not scale very well. This lack of flexible libraries for persistent object storage prompted us to develop our own storage manager based on flat-files. The custom disk-based allocator, that uses a memory cache and a disk file, provides scalability to very large datasets. Thus, DMTL provides an elegant solution when a memory-based back-end fails due to enormous growth of the number of patterns generated.

4.4.1 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 a *Vertical attribute table* (VAT). 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; vertical representations have proved to be useful in several pattern mining tasks (Zaki 2000a, 2001, 2002).

In DMTL there is specialized VAT per pattern-type. Depending on the pattern type being mined the vat-type class may be different, and the VAT intersection 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 is simply the intersection of the two vectors, consisting of all the common tids (Zaki 2000a).
- *Sequence*: The VAT for a sequence is defined as: `vector <pair<tid, vector <time>>>`. For each item, the VAT records the tid where the item occurs, as well as a vector of associated time-stamps. The intersection operation has to compare for matching tids, and in addition requires comparison of the timestamps when doing sequence joins. For instance, when computing the VAT intersection for a sequence $A \rightarrow B$ from the VATs for items A and B, one needs to match the tid *and* ensure that the timestamp of A in that tid is less than that of B (see Zaki 2001 for details of vertical sequence joins).
- *Tree*: Define `triple` to be `(tid, scope, match-label)`, then the VAT for a tree pattern is defined as: `vector<triple>`. The `tid` identifies a tree in the input database; `scope` is an interval `[l,u]` which denotes the range of depth-first-search (DFS) vertex ids which lie under the last node of the tree, and `match-label` is a list of DFS positions at which the last tree node occurs.

Intersection of tree VATs is an involved operation, comprising in-scope and out-scope tests (refer to Zaki 2002 for more details on vertical tree mining).

- *Graph*: DMTL implements a novel VAT representation for graphs. The VAT for a graph is defined as a `vector<edge_vat>` where an `edge_vat` is defined as `vector<pair<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 the graph. In Sect. 5, we show an example of how DMTL mines graph patterns using the graphs VATs.

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

4.4.2 Generic memory management via allocators

Memory management is an integral part of any pattern mining task. Since pattern mining datasets are typically large, efficient memory allocation becomes crucial to achieve a superior performance. Sometimes a dataset does not even fit in main memory, so parts of it need to be staged into the memory from the disk for the algorithm to continue. Since back-end access is tightly embedded in the mining algorithm, it is very difficult for the user to modify the back-end to obtain scalability or persistence.

DMTL implements a generic memory manager, by adopting the allocator concept of STL. Every data structure gets its memory from the allocator that is passed to it as a template parameter. The generic allocator concepts provide the freedom to choose a memory manager. Such a framework is very helpful in situations where heterogeneous data sources (DBMS, flat file, shared memory, etc.) are used for data management. Furthermore, it allows users to allocate memory in such a way that facilitates the deployment of mining-aware caching mechanisms. Such mechanisms combine the knowledge of access patterns to devise cache replacement strategies that improve spatial locality. It has been recently demonstrated in Ghoting et al. (2005) that a customized cache mechanism can dramatically enhance performance. Currently, DMTL implements three allocation mechanisms: purely memory-based, memory-map-based (mmap) persistent allocator and finally a file-based allocator with a memory buffer that provides scalability for a process that would otherwise exceed the virtual memory size (on a 32 bit machine, 2^{32} is the virtual address space size). Work to integrate relational and object-relational databases is in progress.

4.5 Generic algorithms

The core FPM algorithm shown in Fig. 2 was introduced in Sect.4.1. Even though we do not enforce a pattern to conform to this precise formulation of the mining process, most FPM algorithms (including the ones in our library) conform closely

to this outline. The pseudo-code in Fig. 2 is implemented in the `freq_pat_mine` method.

```
1   template<
2     class PATTERN,
3     class MINE_PROPS,
4     class SM_TYPE>
5   void freq_pat_mine(const pat_fam<PATTERN>& Fk,
6       pat_fam<PATTERN>& freq_pats,
7       int& min_sup,
8       count_support<MINE_PROPS, SM_TYPE >& cs)
```

The first parameter to this method, `pat_fam`, is a collection of patterns that belong to the same prefix-based equivalence class. The second parameter, `freq_pats` is used to collect the final set of frequent patterns. The rest are related to the support counting and are passed to the back-end routines. Note that in the above example `PATTERN` is a concept, thus the algorithm automatically chooses the most efficient implementation to mine it by matching pattern properties along the pattern hierarchy.

4.5.1 Candidate generation

Pattern types differ in the way they generate candidates. The number of candidates generated or the operation involved (adding a node or adding an edge) differ for one pattern type versus another. Despite these differences there are fundamental similarities. Each pattern has a set of locations (called extension points) where an extension operation, such as add edge or add node, leads to a new candidate pattern. The `freq_pat_mine` method calls the `join` method to generate new candidates by joining two frequent patterns. The interface for the join method is as shown below:

```
1   template<
2     class PAT_PROPS,
3     class MINE_PROPS,
4     class SM_TYPE>
5   pattern<PAT_PROPS, MINE_PROPS, SM_TYPE>** join(
6       const pattern<PAT_PROPS, MINE_PROPS, SM_TYPE>*
        pat_i,
7       const pattern<PAT_PROPS, MINE_PROPS, SM_TYPE>*
        pat_j)
```

This method takes two pattern pointers and outputs an array of pattern pointers. An array is chosen, as sometimes more than one pattern can be created from the join operation. For example, if a pattern A is the set $\{a, b, c\}$ and another pattern B is the set $\{a, b, d\}$ and their VAT (list of transactions they occur in) are $\{1, 4, 10\}$ and $\{1, 10, 12\}$, respectively, a join (set union operation) produces one pattern $\{a, b, c, d\}$, and the corresponding intersection of VATs (set intersection operation) produces $\{1, 10\}$, which is the VAT of the new pattern. However, the join method shown here materializes the pattern join only; the associated VAT intersection is done in the back-end.

4.5.2 *Isomorphism checking*

For itemsets and sequences we can circumvent generating isomorphic patterns by intelligent candidate generation (Agrawal et al. 1996; Zaki 2001). Essentially, we exploit the 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 (Zaki 2002). However, unordered trees (Nijssen and Kok 2003), free trees (Chi et al. 2003) and graphs (Yan and Han 2002a; Huan et al. 2003a) require isomorphism testing. The isomorphism checker is provided by the `check_isomorphism` method and it is templatized on the pattern properties. Our library provides specialized isomorphism routines for various patterns—general graphs and unordered trees, to name a few.

```
1  template<
2    class PAT_PROPS,
3    class MINE_PROPS,
4    class SM_TYPE>
5  bool check_isomorphism (
6    pattern<PAT_PROPS, MINE_PROPS, SM_TYPE>*
     cand_pat)
```

4.5.3 *Support counting*

The support counting functionality is supported by the *Count Support* module in Fig. 3. Since support counting needs to query the back-end, this module acts as a liaison between the front-end and the back-end. The interface for the `count` method is given below:

```
1  template<class PATTERN>
2  void count(PATTERN* p1, PATTERN* p2, int min_sup)
```

A join of patterns in the front-end triggers an associated VAT intersection in the back-end. We provide different back-end implementations, all storing the same VAT, but which may be in different formats. For example, the VAT stored in persistent STL (Gschwind 2001) is necessarily different than that stored in the memory-based back-end. Nevertheless, the VAT intersection algorithm is the same. Inspired by STL's design, we used iterator concepts to decouple the algorithm from the actual data structure. To reiterate, the design of DMTL consists primarily of three challenging components: pattern structure, pattern algorithms, and back-end storage facility. More details of the generic design of DMTL are available in Hasan et al. (2005).

5 Graph mining: a novel vertical approach

The VAT-based mining approach followed by DMTL is based on the vertical database representations previously proposed in Zaki (2000a) for itemset mining, in

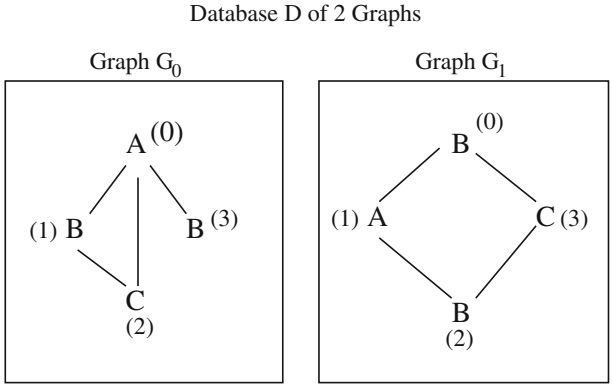
Zaki (2001) for sequence mining and in Zaki (2002, 2005a) for tree mining. However, a vertical representation for graph mining has previously not been described.

In this section we discuss the VAT structure for graphs, and show how it captures each occurrence of a graph in the dataset. VAT intersection operation for graphs is defined thereafter. A graph VAT for a graph g is defined to be a collection (vector) of `edge_vats` (as mentioned in Sect. 4.4.1), corresponding to each edge in the graph. The VAT distinctly identifies each occurrence of g in any graph g_d, where g_d denotes a graph from the dataset D. This is essentially the *subgraph isomorphism* problem: we wish to determine if a graph in the database $g_d \in D$ contains a subgraph g_s isomorphic to the given graph g. Subgraph isomorphism is a known NP-complete problem, and the method we have proposed solves it by utilizing the additional information (the graph VAT) associated with each graph. The basic idea is to avoid computing the full isomorphism each time a candidate pattern is extended; instead we solve the problem in an incremental fashion. Suppose g were an extension of g_k (by adding another edge to g_k); then (i) g can be present in only those $g_d \in D$ which contain a subgraph isomorphic to g_k, and thus (ii) efficiently storing occurrences of g_k in D shall circumvent the need to perform the costly subgraph isomorphism for g from scratch. Our VAT scheme for graphs is built upon these ideas.

A distinct vertex id (vid) is assigned to each vertex in a graph $g_d \in D$; the vid is local to g_d and the same label may be mapped to different $vids$ across various graphs in the dataset. The `tid` member of `edge_vat` identifies a distinct g_d in D which contains the edge e, and `vids` maintains a pair of vertex ids for *each* occurrence of e in g_d. Thus the `edge_vat` is made up of a pair: the graph id (tid), and a collection of edges, which are given as pairs of vertex ids (edge labels can easily be added, but we omit the details here). The graph VAT is a collection of `edge_vats` of all the edges that are on the rightmost path (Yan and Han 2002a) of the graph. Since, the candidate generation step in graph mining algorithm generates patterns only by a rightmost extension, it suffices to store the graph VAT for the rightmost path.

Figure 7 shows a sample dataset D of two graphs. The nodes are represented by their labels (A, B or C) and each node's vid is indicated beside it in parenthesis. For ease of demonstration, we have ignored edge labels (all edges have the same dummy label) in this example. Dealing with labeled edges is however a simple extension of the methodology described in the example, and is implemented in DMTL. The edge $(A - B)$ occurs twice in both G_0 and G_1. Hence its VAT has two entries, one for each graph. The first record in the VAT for $(A - B)$, namely $(0, [(0, 1), (0, 3)])$, in its `edge_vat` corresponds to $(A - B)$'s occurrence in G_0: vertex ids 0 and 1, and between 0 and 3. Moving to higher level graphs such as g_1 in Fig. 8, the VAT now comprises two `edge_vats`, for the two edges $(A - B)$ and $(B - C)$. $(A - B)$'s `edge_vat`, for g_1, in Fig. 8, lists its occurrences as part of g_1, and hence those occurrences are a subset of those shown in Fig. 7. In G_0 the edge between vertex ids 0 and 3 constitutes an $A - B$ edge, but it does not appear as supporting g_1, since g_1 represents the linear chain $(A - B - C)$ and $(0, 3)$ cannot be extended to support g_1. Thus the pair $(0, 3)$ is not contained in g_1's `edge_vat`.

We now give details of the intersection operation for two graph VATs. There exist two types of intersections, corresponding to the two kinds of extensions:



VATs of level−1 graphs (single edges)

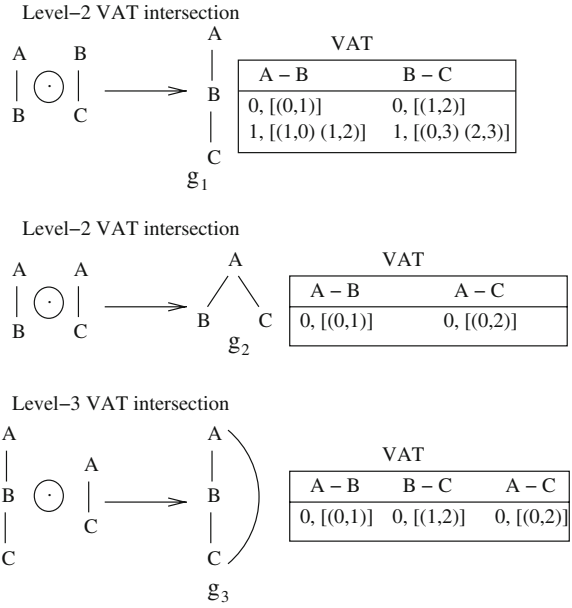
A – B	A – C	B – C
0, [(0,1) (0,3)] 1, [(1,0) (1,2)]	0, [(0,2)]	0, [(1,2)] 1, [(0,3) (2,3)]

Fig. 7 Sample graph database & level-1 VATs

- *Forward intersection*: This corresponds to extending a graph g by adding a new vertex, along with a new edge. For example, in Fig. 8, g_1 is constructed by the forward extension of the graph $g = (A - B)$ with the edge $e = B - C$. The intersection operation first checks for a matching tid, and then checks if for an edge (v_i, v_j) along the rightmost path (Yan and Han 2002a) in the `edge_vat` of graph g, there exists an edge (v_k, v_l) in the `edge_vat` of edge e, such that $v_j = v_k$ and v_l does not already exist in g. In the given example of g_1, the edge $(0, 1)$ for graph $g = (A - B)$, and $(1, 2)$ for edge $e = B - C$ satisfy these conditions, and thus become entries in the `edge_vat` of g_1.
- *Backward intersection*: Backward intersection leads to a cycle in the graph and corresponds the addition of a new edge between vertices already in the graph. For example, g_3 in Fig. 8 is formed by the backward extension of g_1 with edge $A - C$. Here the intersection operation first checks for a matching tid, and then checks if for an edge (v_i, v_j) along the rightmost path, in the `edge_vat` a graph g, there exists an edge (v_k, v_l) in the `edge_vat` of edge e, such that $v_j = v_k$, and that v_l already exists in g. For g_3, we find that for the edge $(1, 2)$ in graph $g = g_1 = (A - B - C)$, we have an edge $(2, 0)$ in the `edge_vat` for $e = A - C$, which satisfies the above conditions, and yields the VAT entry shown for g_3.

The candidate generation step and the isomorphism checking for graph mining in DMTL follow the same principle suggested by gSpan(Yan and Han 2002a). The main difference is in the support counting via VAT intersections, which we outlined above.

Fig. 8 Graph VAT intersection



Space analysis of graph VAT: Let us assume that a pattern P has k edges on its rightmost path and that the pattern occurs in t objects from the database D. Object o_i has e_i embeddings of the pattern, where $i = \{1, \ldots, t\}$ and $e_i \geq 1$. To analyze the size of the VAT let us break down each component of the VAT. Since the rightmost path of P has k edges, the size of the `vector` of `edge_vat` is k. And as the pattern occurs in t objects in the database, the `vector` corresponding to each `edge_vat` will have t pairs of `<tid, vids>`. For e_1 embeddings in object o_1, the size of the vector corresponding to the `vids` structure will contain e_1 `<int, int>` pairs. So the total size of the VAT is of the order of $O(kt \sum_{i=1}^{t} e_i)$. The VAT for a graph pattern is quite space intensive, prompting algorithms to pay keen attention on rapid pruning during pattern enumeration.

6 Incorporating new patterns

In order to mine a pattern of type `T` in our framework, the pattern has to define certain data structures and operations specific to that pattern. The following is a list of such structures and patterns:

1. Data structure that can capture occurrences of the pattern in a database object of type `T`. We call it the VAT for the pattern.
2. Ability to generate the next level of candidate patterns, given patterns from a previous level.
3. Mechanism to perform isomorphism checking on a pattern.

4. Counting the number of occurrences of the pattern within a dataset of such patterns.
5. Comparison operators such 'greater than,' 'less than' and 'not equal' that can order the patterns. Sometimes it is hard to define the 'less than' (and 'greater than') operators for complex patterns. As a result defining these operators is optional. On the other hand, defining the 'not equal' operator is mandatory. The 'not equal' operator, which is true only if the patterns are not isomorphic, can be performed by assigning a unique signature for each instance of a pattern. We call this signature the canonical code for the pattern.
6. Utility functions for parsing datasets containing instances of `T` and for serializing and de-serializing an instance of `T`.

Specialized patterns can invoke functionality provided by a parent pattern. Representing patterns as property-based concepts allows users to represent such a sub/super-pattern relationship. Hence adding a new pattern involves adding new properties to the framework or defining a new subset of properties from the existing ones. Subsequently, one may need to redefine (or specialize) functions for this new pattern. In most cases, only a few key functions need to be redefined or specialized, and this effectively allows us to mine a new pattern with minimal effort.

Let us walk through an example to see how a completely new pattern can be mined. Note that the current implementation consists of the four key patterns—sets, sequences, trees, and graphs. We describe now how to mine all frequent *multisets*, given an input dataset containing multisets. A multiset is a special type of set wherein each item is associated with a number indicating the number of times the item is repeated in the set. Such a pattern can be useful in market-basket analysis if the quantities of items bought by a customer are to be taken into account. An itemset is a special case of the multiset where the quantity is 1 for each item. Figure 9a shows a sample multiset dataset and Fig. 10 outlines the operations and data structures that have to be redefined for mining multisets.

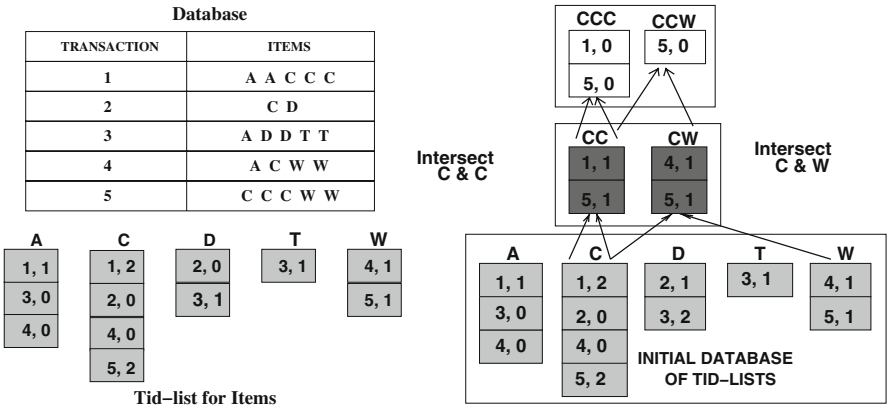


Fig. 9 Multiset mining

```
typedef proplist<no_edges> MSET;
typedef vector<int> ITEMSET_VAT;
typedef vector<pair<int, int> > MSET_VAT;

/* Generates multiset candidates */
template<
   class PAT,
   class MINE_PROPS,
   class SM_TYPE>
void candidates(const pat_fam<MSET>& Fk, ....);

MSET_VAT* intersection(pattern<MSET>* cand_pat);
```

Fig. 10 Adding a new pattern

6.1 Mining multiset patterns

Since multiset is a generalization of an itemset the candidate generation method for multisets is quite similar to that for itemsets. For example, for itemset mining, `ABC` and `ABD` result in `ABCD` as the new candidate pattern, where `AB` is the common prefix for both the patterns. Similarly, new candidate multiset `ABBCD`, can be generated from frequent multisets `ABBC` and `ABBD`. The main change in candidate generation is that we allow multiple occurrences of items in the set. Thus `AA` and `AB` can join to produce `AAB`, and so on. Note also that we keep the items in the multiset sorted (lexicographically). For example, all occurrences of `A` come before those of `B`, and so on.

The main difference for multiset mining compared to itemset mining is really in the the VAT structure (see Figs. 9 and 10) and the associated VAT intersections. Figure 9a shows the vertical VAT representation for each length one pattern (item). The VATs consist of a list of a pair of integers. The first integer is the transaction id and the second integer is the number of *remaining occurrences* of the item in the multiset transaction. For example, `A` occurs twice in transaction 1, therefore there is one remaining occurrence, resulting in the first entry in the VAT, namely (1, 1). In contrast, an itemset VAT only records the transaction id.

In the intersection of two multiset VATs (Fig. 9b) we traverse their VATs trying to match the transaction id. This is common for both itemsets and multisets. The resulting VAT for itemsets is simply the intersection of the tid-list of the initial VATs. In addition, the multiset VAT intersection needs to update the remaining occurrences field. For self-join (e.g., intersecting the VATs of `C` and `C`), the remaining occurrences gets reduced by one. For other intersections (e.g., intersection the VATs for `C` and `W`), the remaining occurrence field of the candidate VAT takes the value of the second VAT. In other words, we always record the remaining occurrences of the last item in a pattern.

Incorporating multisets into the DMTL framework was accomplished in only a few hours, highlighting the extensibility of our approach. Multisets were a case of building a mining algorithm for a more general pattern given a specialized pattern. Likewise, given a generalized algorithm we can build a specialized algorithm. For example, given a graph mining algorithm, we can easily build an algorithm for mining cliques, since a

clique is a constrained graph. As a result, the process of mining cliques resembles that of mining graphs. While the candidate generation step for graphs generates multiple candidates, the candidate generation step for cliques needs to generate only fully connected graphs. This is much simpler than generating all possible candidates. The isomorphism checking and support counting for cliques does not change from regular graphs since cliques are specialized graphs. Note that the task of mining cliques is similar to the task of mining multisets, since like multisets, cliques can have repeated labels. Also, while cliques are fully connected, and multisets are fully disconnected, the mining approach is remarkably similar in the two cases; cliques just differ in the isomorphism-checking step. This example further illustrates the inherent extensibility of the DMTL framework. In general, to add a new pattern type, the user will need to provide implementations for all three stages of mining, namely candidate generation, isomorphism checking and support counting. However, based on our experience, we expect that relatively little effort will be required to add the remaining patterns from the pattern hierarchy shown in Fig. 6.

6.2 Mining closed and maximal patterns

Even though the current version of DMTL addresses the frequent mining problem, it can be easily modified to accommodate more complex mining tasks such as closed and maximal mining.

The algorithm for frequent pattern mining in Fig. 2 can be modified to capture the maximal patterns. The modified version is presented in Fig. 11. In the algorithm, if any frequent candidate pattern is found (line 6) then the current pattern, denoted by $c.parent$, cannot be maximal. This satisfies the first condition for maximality of a pattern (line 9). The second condition determines if a super-pattern exists in the current $\mathcal{M}$ that subsumes $c.parent$ (line 10). When both these conditions are satisfied the pattern is added to the maximal set (line 11). The rest of the algorithm is quite similar to that of Fig. 2. We modified the graph mining code to build a maximal graph mining implementation. The subset/superset checking can be performed in two stages. In the first stage, the graphs are treated as multi-sets and subsumption checking is done for each pair of multi-sets. If multi-set MS_A for a graph A is not a subset of multi-set MS_B of another graph B, then A cannot be a subgraph of B. A large number of candidate subgraphs are eliminated in this first stage. Note that this approach is much cheaper as compared to a subgraph isomorphism check between A and B. In formal notation, $MS_A \not\preceq MS_B \Rightarrow A \not\preceq B$, but $MS_A \preceq MS_B \not\Rightarrow A \preceq B$. As a result, subgraph isomorphism check has to be performed for the multi-sets that satisfy the subset condition. This is the second stage of the subset/superset checking. The subgraph isomorphism checking was implemented using the algorithm proposed in Ullmann (1976).

Similar modifications to the basic algorithm in Fig. 2 can be performed to accomplish closed pattern mining. Of course, we have to admit that the high level algorithm in Fig. 11 does not capture the optimizations that are implemented by specific algorithms to achieve superior performance. At the same time, we claim that most of the optimizations are performed either in the candidate generation or the support counting stages. And those can be incorporated into the framework based on the degree

Fig. 11 Generic maximal pattern mining algorithm

```
// [P] is a group of patterns (that share a prefix)
// DB is the database
// k is initialized to 0
Enumerate-Maximal-Patterns ([P], π^min):
1.  C_{k+1} = candidate_generation([P], DB)
2.  is_max=true
3.  ∀ candidates c ∈ C_{k+1}
4.    if (isomorphism_checking(c)) then
5.      support_counting(c, DB)
6.      if (c.sup ≥ π^min) then
7.        F_{k+1} = F_{k+1} ∪ c
8.        is_max=false
9.  if (is_max) then
10.   if (!(∃M_i ≻ c.parent)) then
11.     M = M ∪ c.parent
12. if ((∃M_i ≺ c.parent)) then
13.   Remove M_i from M
14. for every group [P_i] ∈ F_{k+1}
15.   Enumerate-Maximal-Patterns([P_i], π^min)
```

of optimizations one desires to obtain. For instance, MAFIA (Burdick et al. 2001b) uses a vertical bitmap representation and performs efficient pruning by exploiting the relationship between different components of the HUT (Head Union Tail) of a given pattern. DMTL provides the same vertical representation which can be modified to include the bitmap optimization. Similarly, the support counting stage can be modified to include the pruning achieved in MAFIA. CloseGraph (Yan and Han 2003) is built around the gSpan algorithm with additional pruning of the search space. Since the graph miner in DMTL is also built along the lines of gSpan, a small set of changes can provide an implementation for CloseGraph.

A large number of variations have been proposed at different stages of the pattern mining tasks. Some of these enhancements target efficient storage of underlying data structures (Ayres et al. 2002), while others try to prune the search space effectively (Roberto J Bayardo 1998; Zaki et al. 1997). Naturally, the existing framework does not provide all these enhancements, but it is flexible enough to allow users to customize the library fairly easily. Since the basic data structures and algorithms are cleanly abstracted, a newer (and more efficient) implementation can be substituted as long as it adheres to the interface. Furthermore, existing components (candidate generation, support counting, etc.) can be broken down if desired by a specialized algorithm. Finally, we would like to emphasize that the framework was designed to provide a jump-start for users who would like to customize existing algorithms to their specific needs.

6.3 Handling different database formats

Although we have described the entire paper in the context of the vertical mining approach, the DMTL framework is flexible enough to incorporate horizontal mining

as well. In fact, previous versions of DMTL provided both vertical and horizontal mining approaches. The horizontal mining was removed while restructuring the framework, in the interest of focusing on just one of the approaches. DMTL can also be extended to handle the projected database approaches (Pei et al. 2001; Han et al. 2000b; Yan and Han 2002a). Even though the projected database approach might seem quite different from the VAT based approach, they are essentially similar. The *prefix projection* operation in the *pattern growth* approach essentially constructs a data structure similar to the VAT for the pattern. A more thorough comparison between levelwise and pattern-growth methods (for sequence mining) is presented in Antunes and Oliveira (2004).

7 Experiments and results

We showed above that a new pattern can be incorporated into the DMTL framework with minimal effort. In addition to this flexibility, the ability to handle substantially very large datasets is another key aspect of the framework. In the past, efforts have primarily focused on the scalability and speedup of specific itemset mining algorithms (Buehrer et al. 2006). Even though the primary focus of our work has been the ability to mine a broad range of patterns, we cannot ignore the fact that scalability is important for any real world use of the framework. In this section, we conduct experiments to test the scalability of the system.

As noted before, DMTL provides a file-system based storage mechanism. In this approach, a restricted size main-memory buffer is allocated. As the algorithm progresses, the memory buffer is used to store intermediate patterns. When the memory buffer space is exhausted patterns are shipped to secondary storage. This ensures that frequently used patterns are kept in the memory buffer, reducing page swapping. Moreover, tailored cache replacement mechanisms can be deployed for the file-based implementation. Such flexibility allows us to utilize the knowledge of the access patterns of the mining algorithms in designing the cache replacement algorithm. As such, the file-based back-end scales to much larger dataset sizes (e.g., 60 GB) at a small additional cost. The patterns are serialized to the file when they need to be evicted from the cache and de-serialized when they need to be fetched into the memory buffer. The time to serialize and de-serialize depends on the complexity of the pattern structure.

We now empirically show the scalability and performance of the DMTL algorithms. All experiments noted below were performed on a Dual 2.7 Ghz PowerPC Apple G5, with 4 GB of main memory, and 400 GB of disk space. We used gcc 4.0 with -O3 optimization[2] to compile our code. Unless otherwise stated, all experiments are conducted in 32-bit mode.

7.1 Flexibility of file-based back-end

In this first set of experiments, we highlight the generic aspect of the file-based back-end. The same back-end implementation can be used with a wide range of

[2] We noted that using gcc 3.3 results in 1.5–2.0 times the speedup when compared to gcc 4.0. In spite of this fact, we chose to use gcc 4.0 due to issues with 64-bit compilation on `Mac OS X` with gcc 3.3.

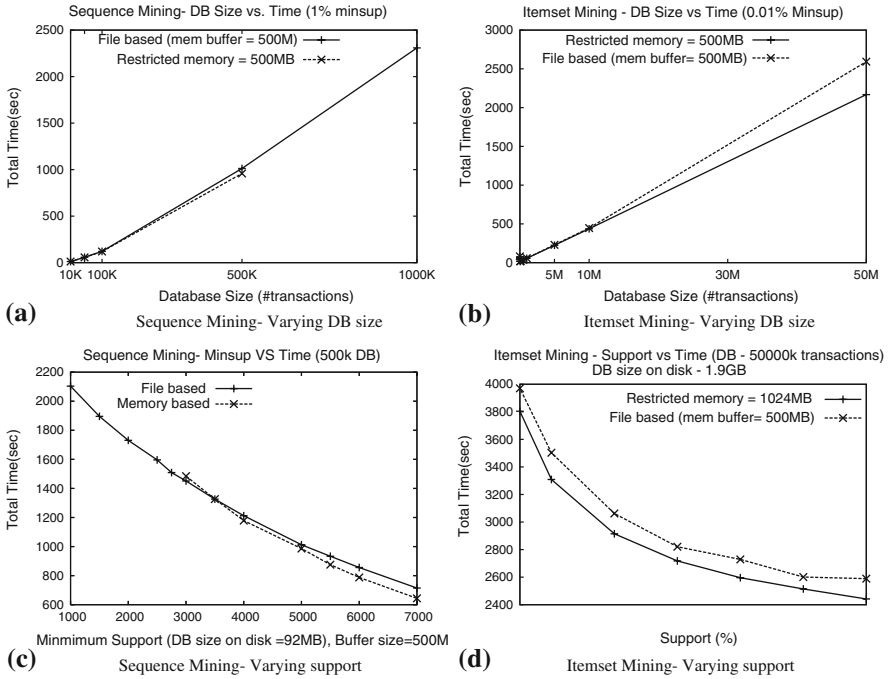


Fig. 12 Scalability results for itemset and sequence patterns

patterns. At the same time, we emphasize the need for a file-based back-end by showing that even a moderately sized dataset can exhaust the physical memory limits of a regular desktop computer. The datasets in this section range from medium to large in terms of their sizes. In the following experiments, parameters such as dataset size, minimum support, and buffer size are altered to bring out different aspects of the framework. We used the IBM synthetic database generator (Agrawal et al. 1996) for itemset and sequence mining, the tree generator from Zaki (2002) for tree mining and the graph generator by Kuramochi and Karypis (2001).

First, we mine induced sequence patterns on datasets of varying sizes and compare the performance of a memory-based implementation with a file-based back-end implementation. The buffer size for the file-based back-end is set to 500 MB and the physical memory for the memory based implementation is also restricted to 500 MB.[3] For both settings, the minimum support is set to 1%. Figure 12a shows that the in-memory implementation cannot execute beyond a dataset size of 500 K transactions, which is when the process reaches the virtual memory limit of 4 GB. On the other hand, the file-based implementation scales linearly to larger dataset sizes. A similar experiment was conducted on an itemset dataset with minimum support set to 0.01%. The results are shown in Fig. 12b. Since itemsets have a smaller VAT structure as compared to sequence VATs, the memory based implementation scales to larger dataset sizes.

[3] Physical memory can be restricted on `Mac OS X` using the `nvram` command.

Figure 12c compares the execution time for induced sequence mining with varying support on the dataset with 500 K transactions. For this execution, the file-based buffer size is set to 500 MB, whereas for the memory-based implementation the memory is restricted to 1 GB. This setting gives the memory based implementation some advantage over the file based approach. For an absolute support of 3500, the execution times for both the back-ends are the same. As the support is reduced, the memory based implementation starts to thrash, resulting in higher execution time. The thrashing behavior can be observed by measuring the page-outs (e.g., using the `sc_usage` utility in Mac OS X). While the memory based approach crashes as the absolute support goes below 3000, the file-based back-end continues to execute for much lower supports. On the other hand, as we increase the support the execution time for the memory-based approach becomes lower than the file-based approach. This can be explained by the fact that all the patterns fit into memory as the support is increased. Moreover, the memory based approach has 1 GB of physical memory whereas the buffer for the file-based approach is restricted to 500 MB. Results for itemsets can be seen in Fig. 12d.

Figure 13a shows the effect of changing the buffer size for file-based back-end. This result confirms to our intuition that as the buffer size increases the execution time decreases. The plots in this figure depend on two factors—the size of the dataset and the replacement strategy used. For larger datasets, the effect is more pronounced, that is, increasing the buffer size reduces the execution time considerably. In our

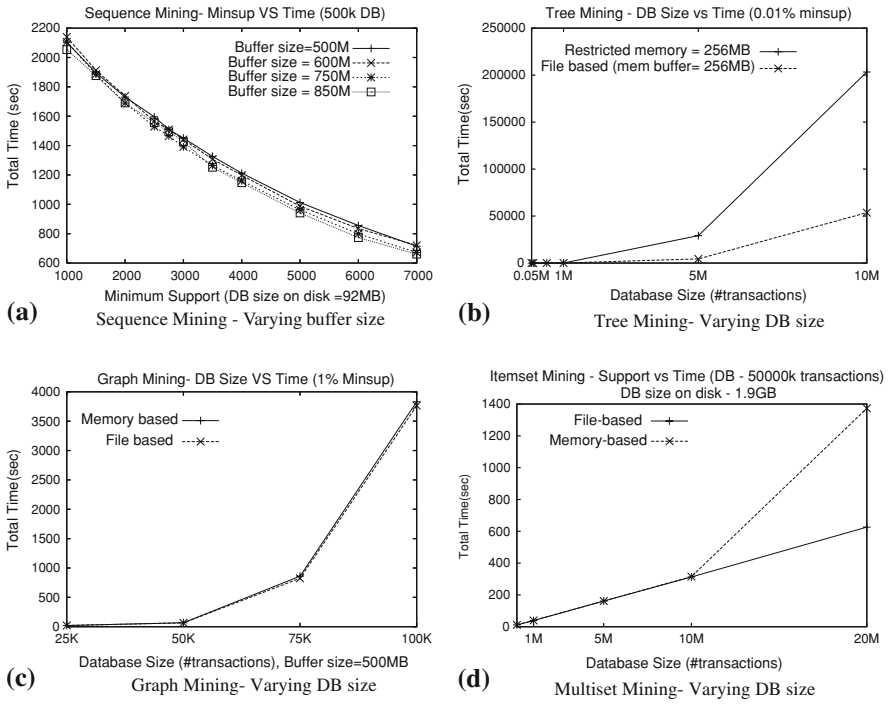


Fig. 13 Scalability results for different patterns

implementation of the file-based back-end, we use the least recently used (LRU) replacement strategy. For algorithms that traverse the pattern lattice in a depth-first fashion, the LRU strategy proves to be a good one. The depth-first strategy encounters smaller patterns first when it traverses a branch of the lattice. From the object lattice, we know that smaller patterns have larger VAT structures. A LRU strategy results in these large VATs being evicted from memory. This allows a larger number of newer patterns to fit in memory since these patterns are much smaller than the evicted pattern. On the contrary, a most recently used (MRU) strategy, would remarkably degrade the performance. Note that the above observations regarding the eviction strategy are applicable only to a depth first mining algorithm and some other strategy might be more successful for alternate methods. Results for more complex patterns such as trees, graphs, and multisets are given in Fig. 13(b), (c), and (d), respectively. In all of these, the advantage of the file-based back-end is clearly seen, especially for larger database sizes.

7.2 Scalability for larger datasets

This section focuses on DMTL's performance while dealing with large itemset datasets (described in Table 1). Even though the results are shown only for itemset mining, they act as an indicator of the framework's scalability for other pattern types, since all the patterns share the same back-end implementation.

The *webdocs* dataset is the largest dataset in the FIMI repository (Goethals and Zaki 2003) and the *D60* dataset is generated from a synthetic dataset generator with parameters shown in Table 1. Figure 14a shows the execution time versus varying support for both file-based and restricted memory approach, using the webdocs dataset. The file-based approach performs better than restricting memory when the support is lowered to 7.5%. The reason for this performance can be explained by observing the process page-ins in Fig. 14b. Page-ins count the cumulative number of memory pages that are swapped in during the course of execution of the process. The page-ins for the restricted memory approach clearly indicate that the system is thrashing. This in turn explains the difference in the execution time. Similar conclusion can be drawn from Fig. 14c and d, which show the virtual memory usage as the execution progresses. The (restricted) memory-based approach relies on the virtual memory system in order to allocate (virtual) memory and hence the process can consume virtual memory as high as 4 GB on a 32-bit machine. Along the way, the virtual memory system swaps pages in order to allocate memory for the active page. In the file-based approach, the process takes control of ensuring that memory is available for new patterns as execution continues. As a result, the moment the physical memory reaches the set (buffer size) limit, which is also the virtual memory at that time, the VAT is shipped

Table 1 Dataset characteristics

Dataset	Size	# Trans	# Items	Avg. Len.
Webdocs	1.48 GB	1,692,082	5,267,656	117
D60	60.5 GB	250,000,000	100,000	40

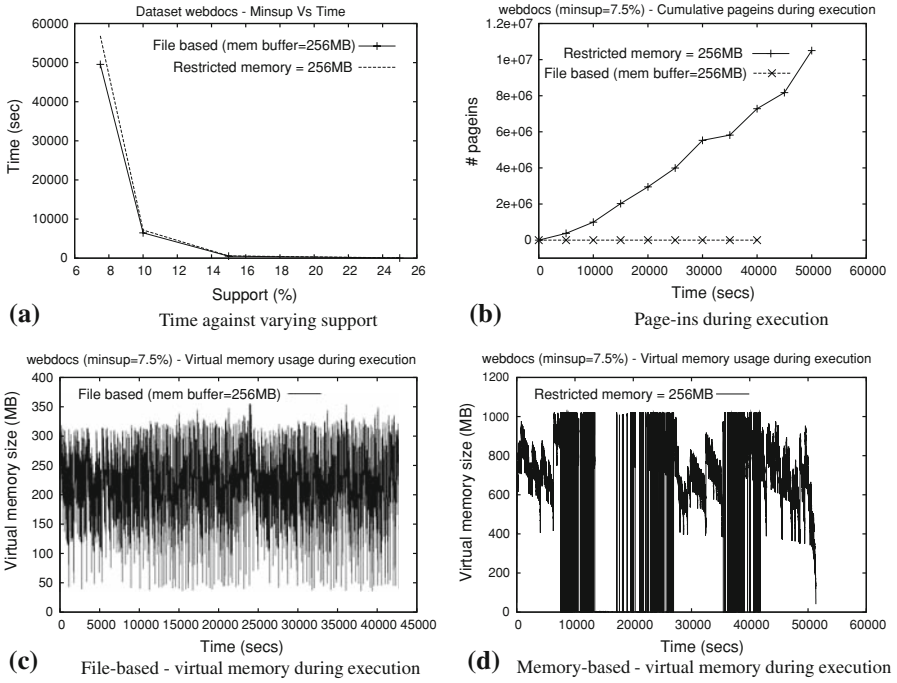


Fig. 14 Dataset webdocs, 32-bit execution, min_sup = 7.5%

to the disk, pulling down both the physical and virtual memory. This can be seen from Fig. 14c, wherein, the maximum virtual memory consumed does not go much beyond the memory buffer size. The maximum virtual memory does reach around 350 MB even though the memory buffer size is set to 256 MB. This difference is attributed to the size of static blocks and the call stack of the process that we do not account for during execution.

Figure 15 shows the results for the *D60* dataset. A memory-based implementation for this dataset does not run for supports lower than 1%. Figure 15c and d show the virtual memory consumption and page-ins, respectively. Another aspect of the process state is shown in Fig. 15b in the form of number of context switches. The context switches for the memory based implementation in Fig. 15b indicate that the scheduler needs to be invoked frequently in order for the process to proceed, which is related to the page-ins in Fig. 15d. In order to bring out the difference between the two methods, only the first 1000 s of the execution are shown. For more complex patterns, as long as the process does not hit the 4 GB barrier the restricted memory approach performs better than the file-based approach beyond which the restricted memory implementation crashes.

7.3 Scalability in 64-bit mode

With the popularity of 64-bit machines, many traditional mining tasks can now be performed due to the much larger virtual memory limit. In the previous section, the

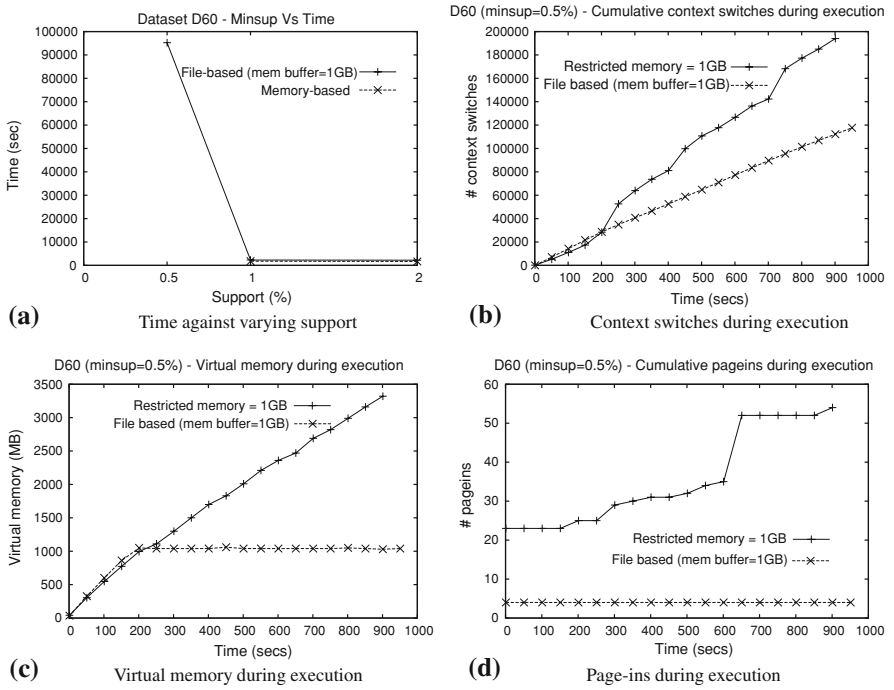


Fig. 15 Dataset D60, 32-bit execution, min_sup = 0.5%

results on 32-bit showed that the memory-based implementation would often crash due to the (comparatively) limited 4 GB virtual memory.

In order to enable the memory-based implementation to run past the 4 GB virtual memory barrier, we compared the performance of both the back-end implementations in 64-bit mode. In 64-bit mode, the virtual space is of the order of 16 terabytes because of which neither the memory-based nor the file-based implementation crashes. But due to the limited physical memory the restricted memory implementation starts to thrash resulting in better execution times in favor of the file-based approach. The results for 64-bit execution are shown in Fig. 16. The restricted memory implementation took about twice as long to execute as compared to the file-based implementation. Again, the page-ins and virtual memory plots explain the thrashing effect; for example the number of page-ins for the memory-based version continues to increase linearly as time progresses.

Our experiments show that the back-end is scalable to much larger dataset sizes as compared to the memory-based implementation. We can improve the performance further by integrating smarter caching mechanisms and by exploring alternative serialization techniques.

7.4 Comparison with stand-alone implementations

In this section we compare the DMTL file based implementation with non-generic implementation of the corresponding pattern mining tasks. Even though we compare

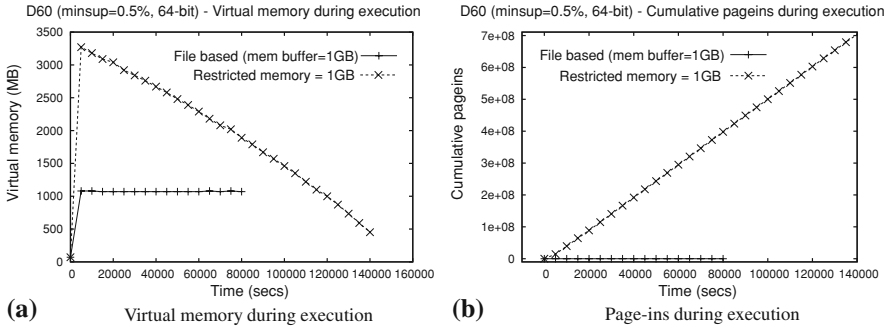


Fig. 16 Dataset D60, 64-bit execution, min_sup = 0.5%

the run time of the DMTL and stand-alone pattern mining algorithms, competing with specialized mining algorithms was not the primary goal of our work. We would like to reiterate that the key focus of our work is to provide an extensible and scalable framework for a wide range of pattern mining tasks. The results in Fig. 17, as we discuss below, corroborate this design principle. For itemset, sequence and tree mining we used Eclat (Zaki 2000a), SPADE (Zaki 2001) and SLEUTH (Zaki 2005a). These algorithms are generally comparable with other existing algorithms and moreover we have ready access to the source code for these algorithms. Access to source code in turn allows us to better evaluate the algorithms by running them under different environments. The same luxury was not available in the case of graph mining, as none of the implementations (Yan and Han 2002b; Huan et al. 2003b; Kuramochi and Karypis 2004) provide the source code. As a result, graph mining was evaluated under a different set of conditions as compared to rest of the experiments.

For all the experiments, other than graph mining, the physical memory was restricted to 256 MB for the non-generic versions of the algorithm. Similarly, the buffer size for file based DMTL was restricted to 256 MB. Figure 17a shows the run time of DMTL itemset miner versus Eclat for different database sizes. Eclat is very promising for smaller dataset sizes, when the entire dataset fits in main memory. For such datasets, Eclat is around 4–5 times faster as compared to DMTL. For larger datasets Eclat ran out of memory, whereas DMTL continues to run with the help of the file based backend. Figure 17b shows comparison with the FP-growth (Han et al. 2000b). The experiments for FP-growth were performed on a machine with 1.6 GHz processor and 512 MB RAM running Linux, since the source code was not available for the algorithm. As with Eclat, FP-growth also fails to run beyond a certain point (5M transactions). On the other hand, DMTL easily scale to a database an order of magnitude larger (50M transactions).

Results comparing SPADE with the DMTL implementation of sequence mining is shown in Fig. 17c, where we mine induced patterns on datasets of varying sizes. SPADE is much faster faster as long as the algorithm can run in-memory, which happens for dataset sizes 100 K and smaller. This difference can be seen in the inset in Fig. 17c. The plot for SPADE almost overlaps with the x-axis. Once again, DMTL scales to datasets that are an order of magnitude larger than those that can be mined

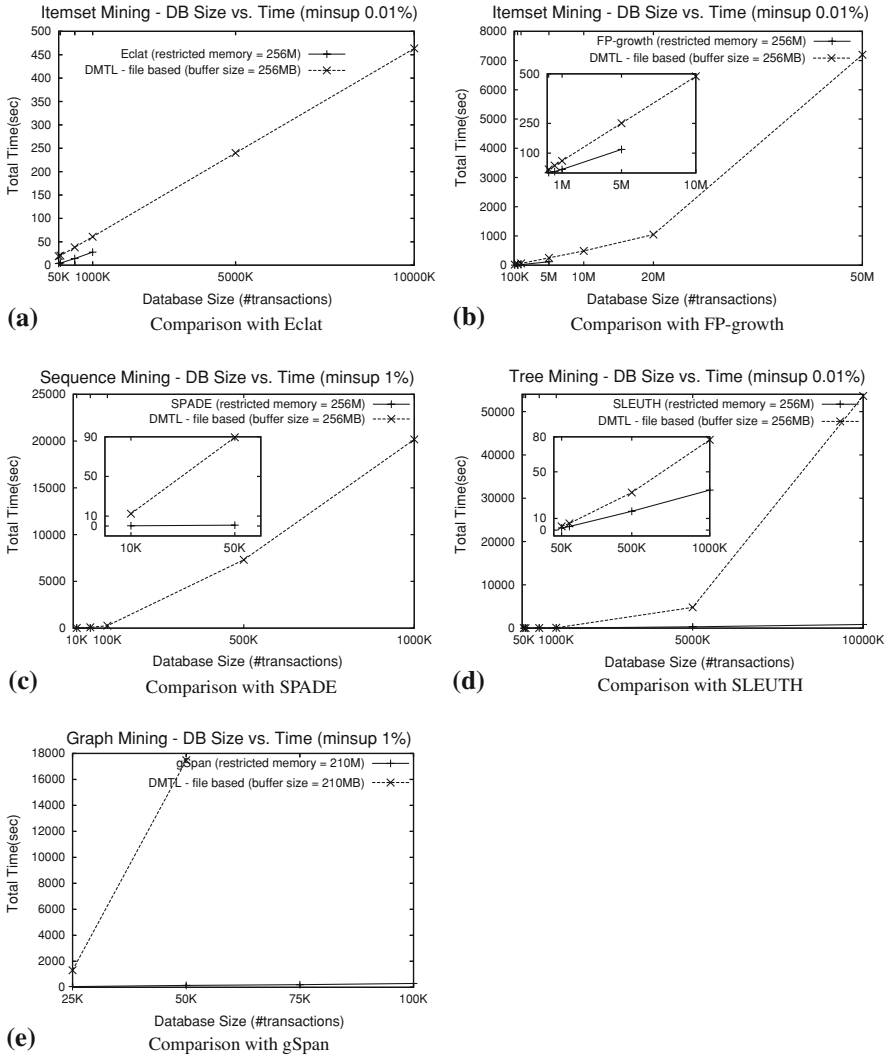


Fig. 17 DMTL comparison with stand-alone (non-generic) mining algorithms for varying database sizes

with SPADE. Figure 17d shows the results on tree mining. For tree mining, on the datasets shown, SLEUTH is much faster than DMTL. We mine embedded and ordered patterns from the tree datasets. The difference in performance between SLEUTH and our file-based implementation becomes larger with increasing number of writes to the file. The current version of the file-based implementation is not optimized and we believe that this performance gap will be reduced as we optimize the file-based implementation. Finally, Fig. 17e shows the results of gSpan versus DMTL. For sets, sequences, and trees, DMTL has a relatively good performance compared to stand-alone implementations, and it even outperforms them as we increase the database size. However, for graph mining, when we compare with gSpan, we find that DMTL

is several orders of magnitude slower. This can be attributed to the fact that gSpan is highly optimized. The gSpan implementation provided by the authors restricts the size of the graphs thus allowing them to perform highly efficient bit-operations. DMTL on the other hand is entirely general purpose, placing no restrictions on the graph sizes, and thus does not leverage such optimizations. gSpan also achieves significant pruning due some other optimizations that are not incorporated in our vertical mining approach (Yan and Han 2002b). Furthermore, DMTL has not really been optimized fully; graph mining seems a good place to start doing that, given the large difference in the performance of the stand-alone algorithm and DMTL. It is worth emphasizing that none of the common graph mining algorithms are available as source code; DMTL provides the only open-source graph mining algorithm (with one exception, Nijssen and Kok 2004).

8 Future work

The current design of DMTL has substantial scope for improvement. There are two different aspects of improvements: data mining aspects and generic design aspects. In data mining, one improvement is to provide more FPM implementations. We would like to add generic mining algorithms for maximal and closed pattern mining. From the back-end perspective, improvements are already in progress, i.e., to design interfaces so that DMTL can work with a relational/object relational database. From a generic design perspective, the improvement is not so simple. In fact, the DMTL design has been challenging because of lack of support for certain generic programming features in C++. For example, our implementation of static lists to manage the pattern properties is the best what we can get from current state of support from the language. But, the property-list based mechanism enforces a strict ordering of the properties in order for the compiler to select the appropriate specialization. As the language evolves in future, we will improve DMTL. Finally, performance evaluation and improvement of the DMTL framework remain ongoing aims.

Acknowledgment We would like to thank the anonymous reviewers for their inputs which have greatly helped us improve the quality of the paper.

References

Agrawal R, Imielinski T, Swami A (1993) Mining association rules between sets of items in large databases. In: ACM SIGMOD conference on management of data

Agrawal R, Mannila H, Srikant R, Toivonen H, Verkamo AI (1996) Fast discovery of association rules. In: Advances in knowledge discovery and data mining. AAAI Press, Menlo Park, CA, pp 307–328

Agrawal R, Srikant R (1995) Mining sequential patterns. In: 11th International conference on data engineering

Antunes C, Oliveira AL (2004) Sequential pattern mining algorithms: Trade-offs between speed and memory. In: 2nd International workshop on mining graphs, trees and sequences with ECML/PKDD

Asai T, Abe K, Kawasoe S, Arimura H, Satamoto H, Arikawa S (2002) Efficient substructure discovery from large semi-structured data. In: 2nd SIAM international conference on data mining

Asai T, Arimura H, Uno T, Nakano S (2003) Discovering frequent substructures in large unordered trees. In: 6th International conference on discovery science

Ayres J, Flannick J, Gehrke JE, Yiu T (2002) Sequential pattern mining using a bitmap representation. In: ACM SIGKDD international conference on knowledge discovery and data mining

Balcazar JL, Casas-Garriga G (2005) On horn axiomatizations for sequential data. In: 10th International confererence on database theory

Bayardo RJ Jr (1998) Efficiently mining long patterns from databases. In: SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data. ACM, New York, USA, pp 85–93

Brin S, Motwani R, Ullman J, Tsur S (1997) Dynamic itemset counting and implication rules for market basket data. In: ACM SIGMOD conference on management of data

Buehrer G, Parthasarathy S, Ghoting A (2006). Out-of-core frequent pattern mining on a commodity PC. In: ACM SIGKDD international conference on knowledge discovery and data mining

Burdick D, Calimlim M, Gehrke J (2001a) MAFIA: a maximal frequent itemset algorithm for transactional databases. In: IEEE international conference on data engineering

Burdick D, Calimlim M, Gehrke J (2001b) MAFIA: a maximal frequent itemset algorithm for transactional databases. In: 17th International conference on data engineering

Chi Y, Yang Y, Muntz RR (2003) Indexing and mining free trees. In: 3rd IEEE international conference on data mining

Chi Y, Yang Y, Muntz RR (2004a) HybridTreeMiner: an efficient algorihtm for mining frequent rooted trees and free trees using canonical forms. In: 16th International conference on scientific and statistical database management

Chi Y, Yang Y, Xia Y, Muntz RR (2004b) CMTreeMiner: mining both closed and maximal frequent subtrees. In: 8th Pacific-Asia conferernce on knowledge discovery and data mining

Cook D, Holder L (1994) Substructure discovery using minimal description length and background knowledge. J Arti Intell Res 1:231–255

Dehaspe L, Toivonen H, King R (1998) Finding frequent substructures in chemical compounds. In: 4th ACM SIGKDD international conference knowledge discovery and data mining

Ganter B, Wille R (1999) Formal concept analysis: mathematical foundations. Springer-Verlag

Garofalakis M, Rastogi R, Shim K (1999) SPIRIT: sequential pattern mining with regular expression constraints. In: 25th International conference on very large data bases

Ghoting A, Buehrer G, Parthasarathy S, Kim D, Nguyen A, Chen Y-K, Dubey P (2005) Cache-conscious frequent pattern mining on a modern processor. In: 31st International conference on very large data bases

Goethals B, Zaki MJ (2003) Advances in frequent itemset mining implementations: report on FIMI'03. SIGKDD Explor 6:109–117

Gschwind T (2001) PSTL—A C++ Persistent Standard Template Library. In: 6th USENIX conference on object-oriented technologies and systems

Han J, Pei J, Yin Y (2000a) Mining frequent patterns without candidate generation. In: ACM SIGMOD conference on management of data

Han J, Pei J, Yin Y (2000b). Mining frequent patterns without candidate generation. In: ACM SIGMOD conference on management of data

Hasan MA, Chaoji V, Salem S, Zaki M (2005) DMTL: A generic Data Mining Template Library. In: 1st Workshop on library-centric software design (with OOPSLA)

Horváth T, Ramon J, Wrobel S (2006) Frequent subgraph mining in outerplanar graphs. In: 12th ACM SIGKDD international conference on knowledge discovery and data mining

Huan J, Wang W, Prins J (2003a) Efficient mining of frequent subgraphs in the presence of isomorphism. In: IEEE international conference on data mining

Huan J, Wang W, Prins J (2003b) Efficient mining of frequent subgraphs in the presence of isomorphism (Technical report TR03-021). University of North Carolina

Inokuchi A, Washio T, Motoda H (2000) An apriori-based algorithm for mining frequent substructures from graph data. In: 4th European conference on principles of knowledge discovery and data mining

Inokuchi A, Washio T, Motoda H (2003) Complete mining of frequent patterns from graphs: Mining graph data. Machine Learn 50:321–354

Kramer S, Raedt LD, Helma C (2001) Molecular feature mining in HIV data. In: ACM SIGKDD international conference on knowledge discovery and data mining

Kuramochi M, Karypis G (2001). Frequent subgraph discovery. In: 1st IEEE international conference on data mining

Kuramochi M, Karypis G (2004) An efficient algorithm for discovering frequent subgraphs. IEEE Transactions on Knowledge and Data Engineering 16:1038–1051
Mannila H, Toivonen H. (1996) Discovering generalized episodes using minimal occurences. In: 2nd International conference knowledge discovery and data mining
Mannila H, Toivonen H. (1997) Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery 1:241–258
Mannila H, Toivonen H, Verkamo I (1995) Discovering frequent episodes in sequences. In: 1st International conference knowledge discovery and data mining
Musser D, Derge G, Saini A. (2001) STL tutorial and reference guide, 2nd edition. Addison-Wesley
Nijssen S, Kok J (2003) Efficient discovery of frequent unordered trees. In: 1st Internationall workshop on mining graphs, trees and sequences
Nijssen S, Kok J (2004) A quickstart in frequent structure mining can make a difference. In: ACM SIGKDD international conference on knowledge discovery and data mining
Oates T, Schmill MD, Jensen D, Cohen PR (1997) A family of algorithms for finding temporal structure in data. In: 6th International workshop on AI and statistics
Pasquier N, Bastide Y, Taouil R, Lakhal L (1999) Discovering frequent closed itemsets for association rules. In: 7th International conference on database theory
Pei J, Han J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu M-C (2001). PrefixSpan: mining sequential patterns efficiently by prefixprojected pattern growth. In: IEEE international conference on data engineering
Savasere A, Omiecinski E, Navathe S (1995) An efficient algorithm for mining association rules in large databases. In: 21st International conference on very large data bases
Shasha D, Wang J, Zhang S (2004) Unordered tree mining with applications to phylogeny. In: IEEE international conference on data engineering
Siek J, Lee L, Lumsdaine A (2002). The boost graph library. Addison-Wesley
Srikant R, Agrawal R (1996) Mining sequential patterns: Generalizations and performance improvements. In: 5th International conference extending database technology
Termier A, Rousset M-C, Sebag M (2002) TreeFinder: a first step towards xml data mining. In: IEEE international conference on data mining
Termier A, Rousset M-C, Sebag M (2004) Dryade: a new approach for discovering closed frequent trees in heterogeneous tree databases. In: IEEE international conference on data mining
Ullmann JR (1976) An algorithm for subgraph isomorphism. J ACM 23:31–42
Wang C, Hong M, Pei J, Zhou H, Wang W, Shi B (2004) Efficient pattern-growth methods for frequent tree pattern mining. In: Pacific-Asia conference on knowledge discovery and data mining
Wang J, Han J (2004) BIDE: efficient mining of frequent closed sequences. In: IEEE international conference on data engineering
Wang J, Han J, Pei J. (2003). CLOSET+: searching for the best strategies for mining frequent closed itemsets. In: ACM SIGKDD international conference on knowledge discovery and data mining
Wang K, Liu H (1998) Discovering typical structures of documents: a road map approach. In: ACM SIGIR international conference on information retrieval
Witten I, Frank E (1999) Data mining: practical machine learning tools and techniques with java implementations. Morgan Kauffman
Xiao Y, Yao J-F, Li Z, Dunham MH (2003) Efficient data mining for maximal frequent subtrees. In: IEEE international conference on data mining
Yan X, Han J (2002a) gSpan: graph-based substructure pattern mining. In: IEEE international conference on data mining
Yan X, Han J (2002b) gSpan: graph-based substructure pattern mining (Technical report UIUCDCS-R-2002-2296). University of Illinois at Urbana-Champaign
Yan X, Han J (2003) CloseGraph: mining closed frequent graph patterns In: ACM SIGKDD international conference on knowledge discovery and data mining
Yoshida K, Motoda H (1995) CLIP: concept learning from inference patterns. Artif Intel 75:63–92
Zaki MJ (2000a) Scalable algorithms for association mining. IEEE Trans Knowl Data Eng 12:372–390
Zaki MJ (2000b) Sequences mining in categorical domains: Incorporating constraints. In: 9th International conference on information and knowledge management
Zaki MJ (2001) SPADE: An efficient algorithm for mining frequent sequences. Machine Learn J 42:31–60
Zaki MJ (2002) Efficiently mining frequent trees in a forest. In: 8th ACM SIGKDD international conference on knowledge discovery and data mining

Zaki MJ (2005a) Efficiently mining frequent embedded unordered trees. Fundamenta Informaticae 66: 33–52

Zaki MJ (2005b) Efficiently mining frequent trees in a forest: algorithms and applications. IEEE Trans Knowledge Data Eng 17:1021–1035

Zaki MJ, Gouda K (2003) Fast vertical mining using Diffsets. In: 9th ACM SIGKDD international conference on knowledge discovery and data mining, pp 326–335

Zaki MJ, Hsiao C-J (2002) CHARM: an efficient algorithm for closed itemset mining. In: 2nd SIAM international conference on data mining

Zaki MJ, Hsiao C-J (2005) Efficient algorithms for mining closed itemsets and their lattice structure. IEEE Trans Knowl Data Eng 17:462–478

Zaki MJ, Parimi N, De N, Gao F, Phoophakdee B, Urban J, Chaoji V, Hasan M, Salem S (2004) Towards generic pattern mining. In: International conference on formal concept analysis (Invited paper)

Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997) New algorithms for fast discovery of association rules. In: 3rd International conference on knowledge discovery and data mining

Zou B, Ma X, Kemme B, Newton G, Precu D (2006) Data mining using relational database management systems. In: Pacific-asia conference on knowledge discovery and data mining