

DMTL : A Generic Data Mining Template Library *

Mohammad Al Hasan, Vineet Chaoji, Saeed Salem,
Nagender Parimi, Mohammed J. Zaki
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180, USA

{alhasan,chaoji,salems,parimi,zaki}@cs.rpi.edu

ABSTRACT

FPM (Frequent Pattern Mining) is a data mining paradigm to extract informative patterns from massive datasets. Researchers have developed numerous novel algorithms to extract these patterns. Unfortunately, the focus primarily has been on a small set of popular patterns (itemsets, sequences, trees and graphs) and no framework for integrating the FPM process has been attempted. In this paper we introduce DMTL, a generic pattern mining library which fuses theoretical concepts from formal concept analysis and generic programming. It provides a framework that allows mining a large spectrum of patterns. We express each pattern in terms of its relational properties. Describing patterns based on their properties results in a pattern concept hierarchy. This hierarchical model is implemented using principles from generic programming. In this paper, we describe our design considerations and the subsequent implementation. Some of the challenges faced in terms of language features have also been highlighted. Apart from using the library in its entirety, we believe that some of its components, such as isomorphism checking, can be used independently. These components can definitely enrich the existing functionality provided in some of the popular libraries such as the Boost Graph Library.

1. INTRODUCTION

Frequent pattern mining (FPM) is a data mining paradigm to extract informative patterns in massive datasets. Its applications are growing enormously, aided by the availability of high computation power, cheap massive storage, and improved technology for extraction and distribution of data. Researchers have successfully applied FPM to a diverse set of problems in the areas of market basket analysis [1], bioinformatics [25, 24], web mining, fraud detection [4], scientific and medical data mining, etc. In many of these application domains, FPM is not the core component. Hence, availability of the FPM library would allow researchers to save significant effort and would enable them to focus on their core competence. FPM research discovers *patterns* that conceptually represent relations among discrete objects. Depending on the complexity of these relations, different types of patterns originate. The most common type of patterns are sets, where the relation is the co-occurrence

*This work was supported in part by NSF CAREER Award IIS-0092978, DOE Career Award DE-FG02-02ER25538, NSF grant EIA-0103708, and NSF grant EMT-0432098.

of objects. A well known example of the set pattern is a supermarket transaction dataset; the set of items that are bought together by a customer is of interest to the business strategists. Next, there are sequence patterns, where co-occurrence of objects is augmented by the presence of an order between them. 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 object relationship evolves in a hierarchical manner, 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 is an example of graph mining.

1.1 Related Work

Although FPM is a very mature research area, development of an FPM library has mostly been ignored. Since the commencement of FPM research with the legendary *apriori* itemset mining paper [1] over a decade ago, several hundreds different scholarly articles have been published. Some proposed algorithmic improvements, some covered different variations of FPM problems, such as maximal frequent [2] or closed frequent pattern mining [13] and some developed algorithms for mining new patterns, like DAG (Directed Acyclic Graph), Free Tree [3], etc. Several others demonstrated the potential of FPM algorithms by applying them to new fields, like bioinformatics, operations research, intrusion detection, etc. No real effort has concentrated on developing a library targeting different FPM tasks. The closest works are MLC++ [10] and Weka [18]. The former is a collection of classification algorithms. The latter is a general purpose Java library for different data mining algorithms that includes only itemset mining. Besides these, there are some independent application programs developed by researchers in academia, mostly to evaluate the correctness and performance of their proposed mining algorithms. But they are very specific, run on a selected format of datasets and are in no way suitable as a library component. They do not offer any standard interface for end users. A collection of such algorithms specifically for itemset mining is available from the FIMI [6] web site. Moreover, several practical machine learning software, bioinformatics search tools, etc., employ FPM as the core mining engine, for which they usually write their specific FPM programs. The unavailability of a generic FPM library thus wastes enormous time and computation resources for programmers and researchers.

We developed DMTL (Data Mining Template Library), a

frequent pattern mining library, that provides a unified interface to mine a range of patterns. Currently the library has implementations for mining four key patterns—itemset, sequence, tree and graph—but the framework provides the scope to mine new patterns also. DMTL adopts a generic design, inspired by the state-of-the-art generic libraries such as the C++ Standard Template Library (STL) [?, 11] and Boost Graph Library (BGL) [15], and hence it provides widespread usability without compromising on efficiency. The library is generic with respect to the following aspects:

- Pattern to be mined.
- Input data source and format.
- Data structure to be used in the mining algorithm.
- Storage management.
- Mining algorithm/approach.

1.2 Contributions

The major contributions of our work towards the data mining community are as follows:

- DMTL offers algorithms for different pattern mining tasks in a unified platform. To the best of our knowledge this is the first effort of this kind in data mining.
- DMTL offers flexible interfaces to 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 very minimal effort from the end user. Users just 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.

We also believe this work contributes to the library development community in the following ways:

- DMTL adopts the generic software development approach using C++ templates. Due to the limitation imposed by the programming language, it is still very difficult for programmers to design generic software. Few books [?, 15] are available that describe an implementation of a generic library. We believe that the design of DMTL could be an example for other generic library developers to follow.
- Apart from its ultimate purpose of discovering frequent patterns, our library provides several stand-alone utilities for various patterns. This primarily includes the isomorphism checking functionality for different patterns. We believe that these features can complement the features provided in BGL.
- While implementing DMTL, we faced numerous challenges, mostly related to programming language support for generic software development. Most of these issues have already been identified by several researchers [14], but our work stands as another practical example of those limitations.

- DMTL uses several template tricks, which we think could be tremendously useful for any generic software developer.

2. PATTERN MINING PRELIMINARIES

The problem of mining frequent patterns can be stated as follows: let $\mathcal{N} = \{x_1, x_2, \dots, x_{n_v}\}$ be a set of n_v distinct nodes or vertices. A pair of nodes (x_i, x_j) is called an edge. Let $\mathcal{L} = \{l_1, l_2, \dots, l_{n_l}\}$, 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 $L_e(x_i, x_j) = l_k$.

A *pattern* P can be represented as the pair (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 (as defined below) k -patterns is referred to as F_k . Given two patterns P and Q , 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 there exists a label-preserving isomorphism from P to Q ; that is, iff there exists a 1-1 mapping f from nodes in P to nodes in Q , such that for all $x_i, x_j \in P_V$: i) $L_n(x_i) = L_n(f(x_i))$, ii) $L_e(x_i, x_j) = L_e(f(x_i), f(x_j))$, and iii) $(x_i, x_j) \in P_V$ iff $(f(x_i), f(x_j)) \in Q_V$. In some cases we are interested in *embedded* sub-patterns. In embedded patterns we modify condition iii) above to allow an edge (x_i, x_j) in P provided $f(x_i)$ and $f(x_j)$ are connected in Q . In other words, P is an embedded sub-pattern of Q if P is a sub-pattern of the transitive closure of Q . If $P \preceq Q$ we say that P is contained in Q or Q contains P .

A database \mathcal{D} is just a collection of patterns (objects, in database terminology). Let $\mathcal{O} = \{o_1, o_2, \dots, o_{n_o}\}$ be a set of n_o distinct *object identifiers*. An object has a unique identifier, given by the function $O(d_i) = o_j$, where $d_i \in \mathcal{D}$ and $o_j \in \mathcal{O}$. The number of objects in \mathcal{D} is denoted by $|\mathcal{D}|$. 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 $\pi(P, \mathcal{D}) = \frac{\pi^a(P, \mathcal{D})}{|\mathcal{D}|}$. A pattern is *frequent* if its support is greater than a user-specified minimum support (*min_sup*) threshold, i.e., if $\pi(P, \mathcal{D}) \geq \text{min_sup}$. 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 *min_sup* frequency requirement (and any other user-specified conditions).

The main observation in FPM is that the sub-pattern relation \preceq defines a partial order on the set of patterns. If $P \preceq Q$, we say that P is more general than Q , or Q is more specific than P . The second observation used is that if Q is a frequent pattern, then generally all sub-patterns $P \preceq Q$ are also frequent¹. More important is the converse, i.e., if P is infrequent and $P \preceq Q$ then Q shall also be infrequent (follows from the anti-monotonicity of frequency). The *prefix* of a pattern of size k is a sub-pattern that consists of the first $k-1$ nodes of the pattern. For efficiency reasons, many FPM algorithms group (at least conceptually) patterns hav-

¹Note that this property does not hold for induced patterns

ing the same prefix into a *prefix-based equivalence class*. The various FPM algorithms differ in the manner in which they search the pattern space.

3. GENERIC ASPECTS OF DMTL

In this section we outline the generic aspects of the Data Mining Template Library.

3.1 Generic Mining Algorithm

While implementing mining algorithms for different patterns, we noticed that they exhibit considerable similarity, which suggests developing a common framework for implementing them. Figure 1 outlines a generic pattern mining algorithm (pseudo-code) that applies to all commonly explored patterns. In the algorithm (not shown in the figure), k is initialized to zero and DB represents a global database. Similarly, other related pattern mining algorithms (closed or maximal pattern mining) also conform closely with this outline. The algorithm is broken down into the major sub-tasks which includes **candidate generation**, **isomorphism checking** and **support counting** (explained in detail in the implementation section). By implementing generic functions for these sub-tasks, we retain the abstraction shown in this pseudocode. 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 1, 2, and so on. Each iteration discovers frequent patterns sized one greater than the previous till no further frequent patterns exist in the database. The example in figure 2 demon-

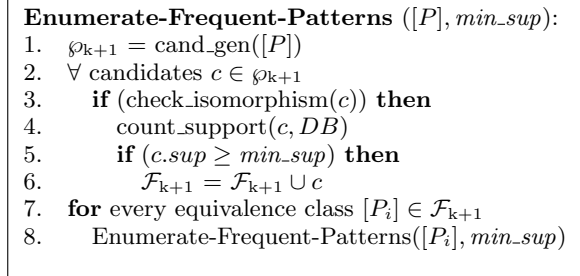


Figure 1: Pattern Mining Algorithm

strates how the generic algorithm works for itemset mining. The database on top left corner of the figure has 4 transactions. Each row contains a collection of items separated by commas. We want to perform itemset mining on this dataset with an absolute minimum support value of 3. The same database is also shown in its vertical format (explained later in subsection 3.1.2). This representation is important in the vertical mining approach. The algorithm first finds all the size-1 frequent itemsets, by making a single database scan. The frequent items from the dataset with a support value 3 or more are A, C, T and W, which are shown in the oval to the right of the dataset. Each of these items is present in at least 3 transactions. Now, the candidate generation step generates six size-2 candidates by joining items from this set. The possible candidates here are shown in the rectangle under the oval. Note that the joining process in itemsets automatically eliminates duplicates. For joining complex patterns (joining two graphs), this may not be the case, and we need to employ isomorphism checking to ensure

that each candidate pattern is generated exactly once. Finally, the support counting step counts the support of each of the candidates from the database. This step drops the itemset AT, as it appears in only 2 (< 3) transactions. The algorithm iterates until the size- k patterns are found. All frequent itemsets produced by this algorithm are shown in the figure. For other patterns, the algorithm follows the exact same approach as detailed in this example.

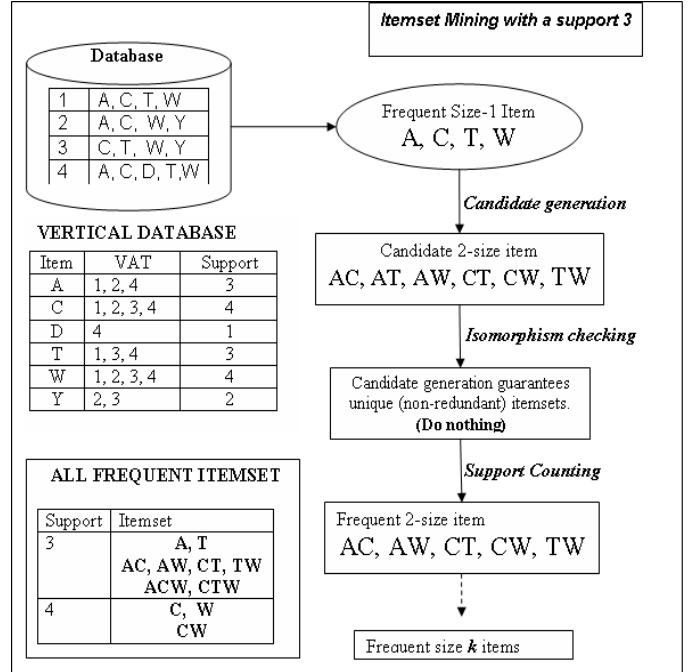


Figure 2: Itemset Mining Example

The sub-tasks of a generic mining algorithm that we referred to in the above two sections can be developed by using generic algorithms expressed with C++ function templates. For example, the **candidate generation** step takes two parent patterns of type T and generates one or more candidate patterns of type T . Here, 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** algorithm takes two input arguments of same type T (a pattern type) and produces a boolean value to indicate whether the arguments are isomorphic patterns or not. The **support count** algorithm takes one input argument of pattern type T , counts its frequency in the entire database and returns an integer value.

In all the above three generic algorithms, the type T models a pattern concept. It has the following requirements:

1. T defines an object that relates some elements.
2. T must adhere to a structure that is defined by a collection of relational properties.
3. T defines a \leq operator.

4. 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 relational properties of a pattern concept that we refer to as *pattern properties* in DMTL are explained in the following subsection.

3.1.1 Pattern Properties

In section 2, we defined patterns in terms of graph abstraction. The choice of graph, indeed comes naturally, since all the patterns are, in a way, specializations of a graph pattern (a set is a special case, which we considered as a graph without any edge). Hence, a graph can represent all the patterns both conceptually and implementation-wise. Using graph implementation for more simpler patterns, like set, sequence or tree introduces inefficiency in the mining algorithm, however, the concept of *pattern property* provides a novel solution to this dilemma. In the implementation section, we explain the way we use *pattern properties* to ensure a generic algorithm that does not compromise efficiency. Here, we explain the different pattern properties that we used.

Relational properties that a pattern type T must conform to, are indeed the graph properties. These properties imposes constraints on graph to formulate patterns like, tree, sequence etc. We analyzed the pattern space and found that the following properties are sufficient to describe the most common patterns, but nevertheless, additional properties may be added seamlessly. The properties are themselves categorized depending on the elements (nodes, edges, etc.) of a graph on which the constraints are imposed.

1. **Edge Relation** The edge set E_g is defined as $E_g \subseteq V_g \times V_g$. Under edge relation category we considered the following properties.
 - **no-edge** Elements in the patterns are not connected with any edge.
 - **directed** Elements in the patterns are connected with directed edge. To put it in another way, we can say, they are asymmetrically related.
 - **undirected** Elements in the patterns are connected with symmetric edges.
 - **cyclic** A pattern is cyclic if at least one vertex is reflexive on edge relation in the transitive closure of the pattern, otherwise the pattern possess the acyclic property.
2. **Vertex**
 - **order** The *ordered* property imposes an ordering on the neighbors of a vertex, or else the pattern is said to be unordered. Ordering is usually relevant for the tree pattern only.
3. **Degree**
 - **indegree_lte_one** This property constrains all vertices of a graph to have indegree ≤ 1 .

- **outdegree_lte_one** This property constrains all vertices of a graph to have outdegree ≤ 1 .

4. Label

- **unique_label** This property requires the labeling function to be one-to-one (injective). Each vertex thus maps to a unique label (a common example of such a pattern is an itemset).

3.1.2 Mining Properties

So far, we discussed that the generic mining algorithm that DMTL advocates can mine any pattern belonging to a pattern concept. But, in data mining research several variations of the core generic mining algorithms exist, by varying the manner in which we perform its sub-tasks. We represent those variations in terms of *mining property*; a user can choose a collection of such mining properties to select the exact kind of algorithm that (s)he would like to choose for the mining process. It is worth noting that, the mining properties are independent from the pattern properties. An analysis of existing FPM tasks revealed the following mining properties that we mention below. As with pattern properties, new mining properties can also be added effortlessly.

1. **Join-type** This category influences the candidate generation phase, in which potentially frequent pattern 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.
 - **$F_k \times F_1$** A $(k + 1)$ -length pattern is constructed by joining a k -length pattern with a unit length pattern.
 - **$F_k \times F_k$** 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** Indicates that the support for a candidate pattern shall be determined by counting its occurrences in the database, testing 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 from what is called a *vertical representation* of a pattern [20]. This vertical representation for a pattern is a list of transactions in which the pattern occurs and is commonly referred to as *Vertical Attribute Table* (VAT). A vertical database lists all the patterns along with their VATs. Figure 2 shows a vertical database in the table titled “Vertical Database”. Support counting using a vertical database is typically faster as it reduces I/O cost.
3. **Transitivity** This category indicates if embedded occurrences of a pattern should be considered in its support counting.

- **induced** Only induced pattern occurrences are counted ².
- **embedded** Transitive closures on the edge relation E are included in the support as well. The transitivity leads to discovery of embedded occurrences of the pattern.

3.2 Generic Storage Manager

Database (back-end) support is an integral part of any pattern mining task. Since pattern mining datasets are typically large in size, back-end management becomes crucial to achieving an efficient implementation. Sometimes a dataset does not even fit in main memory, so part of it needs to be saved on 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's implementation of back-end database support is generic, through a generic storage manager class. Following the STL iterator concept, we decoupled the back-end database from the algorithm using iterators. Any access to the database is done only through the iterators. We also implemented three different storage managers; all provide iterator classes. Discussion about each of them is given in the implementation section.

3.3 Generic Input Data Source

DMTL is implemented with an objective to be widely applicable. However, the format of the input dataset is different for different application domains. For instance, in supermarket transaction databases, items are usually represented by numeric identifiers, whereas in bioinformatics, items may use string representations for protein or DNA sequences. DMTL takes care of these kinds of dataset irregularities by implementing a generic tokenizer, which is templated with various arguments to adapt to a wide variety of input datasets.

4. PATTERN PROPERTY CONCEPT

The generic design of DMTL mining algorithms for all patterns based on the *pattern property* has a foundation in Formal Concept Analysis (FCA) [5]. We explain this next.

4.1 Formal Concept

DEFINITION 1. A **formal context** $(K) := (G, M, I)$ consists of two sets, G and M , and a relation I . The elements of G are called the **objects** and the elements of M are called the **attributes** of the context. In order to express that an object g is in the relation I with an attribute m , we write gIm or $(g, m) \in I$ and read it as "object g has attribute m ."

DEFINITION 2. For a set $A \subseteq G$ of objects we define

$$A' := \{m \in M \mid gIm, \forall g \in A\}$$

(the set of attribute common to the objects in A). Correspondingly, for a set B of attributes we define

$$B' := \{g \in G \mid gIm, \forall m \in B\}$$

(the set of objects which have all the attributes in B .)

²Note that for graphs we actually mine connected sub-graphs, and not only induced sub-graphs

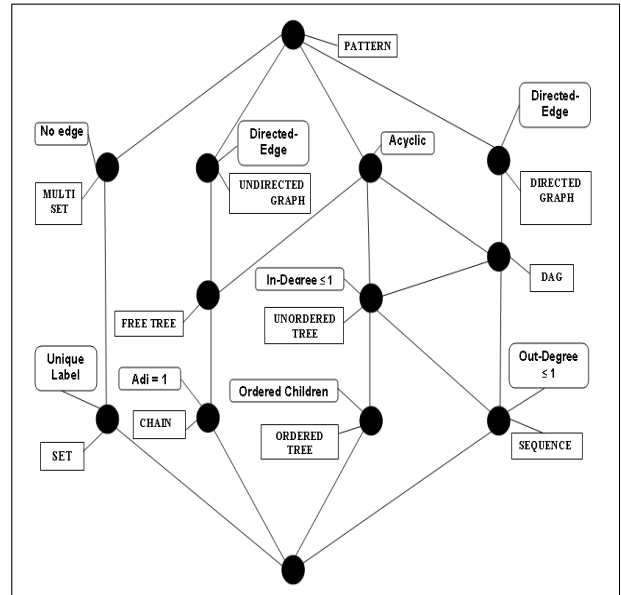


Figure 3: Pattern Property Concept Lattice

DEFINITION 3. A **formal concept** of the context (G, M, I) is a pair (A, B) with $A \subseteq G, B \subseteq M, A' \subseteq B$ and $B' \subseteq A$. We call A the **extent** and B the **intent** of the concepts (A, B) . $\mathcal{B}(G, M, I)$ denotes the set of all concepts of the context (G, M, I) .

In DMTL, we consider M as the set of all patterns that we want to mine, G as the set of all pattern properties and I as the relation that a pattern conforms to a property, then (G, M, I) is a context. Now, if $A \subseteq M$ is maximal a collection of patterns, and $B \subseteq G$ is the set of properties that are common to all the patterns in A , then (A, B) is a formal concept of the context (G, M, I) .

Example: If $A = \{DAG, Sequence, Ordered Tree, Unordered Tree\}$ is the set of patterns and $B = \{Directed, Acyclic\}$ is the set of properties common to members of A , then (A, B) forms a formal concept. The set A , i.e. the set of patterns, is the extent of the concept and B , the set of properties, in the intent of the concept.

The concept in generic programming adheres with definition 3, if the objects equate with abstractions (types, in particular) and the attributes with requirements. In [16], Wilcocks et al. provide a precise definition for concepts, as they are used in practical generic programming. That definition is an extended form of the above definition, where the extensions clarify several issues related to generic software design and programming languages.

4.2 Formal Concept Lattice

DEFINITION 4. If (A_1, B_1) and (A_2, B_2) are concepts of a context, (A_1, B_1) is called a **sub-concept** of (A_2, B_2) , provided that $A_1 \subseteq A_2$ (which is equivalent to $B_2 \subseteq B_1$). In this case, (A_2, B_2) is a **superconcept** of (A_1, B_1) , and we write $(A_1, B_1) \leq (A_2, B_2)$. The relation \leq is called the **hierarchical order** of the concepts. The set of all concepts of (G, M, I) ordered in this way is denoted by $\mathcal{B}(G, M, I)$ and is called the **formal concept lattice** of the context (G, M, I) .

Example: The set of all *pattern-property formal concepts* form a concept lattice as illustrated in Figure 3. In this figure, every node is a formal concept. The corresponding set of objects and attributes of that concept are shown next to it, in boxes with rectangular and rounded edges, respectively. Every box only list those objects or attributes that are not implicitly inherited through the refinement relation (discussed in next paragraph). We can retrieve the entire set of extents (objects) by tracing all paths which lead down from that node. On the other hand, the intents (attributes) can be obtained by tracing all paths leading upward from that node.

If we consider the node labeled with the formal object DAG, it represents a formal concept with objects

{DAG, Sequence, Unordered Tree, Ordered Tree}

and with properties {Acyclic, DirectedEdge}

4.3 Concept Refinement

DEFINITION 5. *Concept refinement is the process of obtaining a sub-concept from a concept. Adding one or more attributes in the intent removes objects from the extent that do not conform to that property.*

Example: We can refine the concept in the above example by adding one property named `indegree_lte_1`. In the refined concept, the pattern DAG is omitted, as DAG does not conform to this property.

4.4 Concept Refinement in DMTL Design

In our generic library implementation, we employed understanding of formal concept hierarchy to develop mining algorithms that can handle different types of patterns. Any algorithm that works for patterns in a pattern-property concept automatically works for the sub-concept. For patterns in sub-concepts, a list of pattern properties that is passed as template arguments matches partially and automatically invokes the algorithm for the patterns belonging to the immediate super-concept. However, there could exist a more efficient implementation for the patterns in the sub-concept as they might be comparably easier to mine. For those cases, we provide a more efficient implementation of the algorithm as an overloading of the template function.³ We discuss the implementation details in the following section.

5. IMPLEMENTATION ISSUES

This section describes the implementation details of DMTL. Three major subsections cover the architecture, data and algorithms of DMTL respectively.

5.1 Architecture

Figure 4 provides a quick look at the various architectural components (in rectangular boxes) of DMTL. We partitioned the components into two main segments—the *front end* and the *back end*. The front end deals with the core mining process while the back end provides the necessary storage support.

³If we were expressing algorithms with classes we would provide the more efficient algorithms as partial template specializations, but in the case of function templates one must currently use overloading instead. Proposals to add partial specialization of function templates to the language standard have been made but to date have not been accepted.

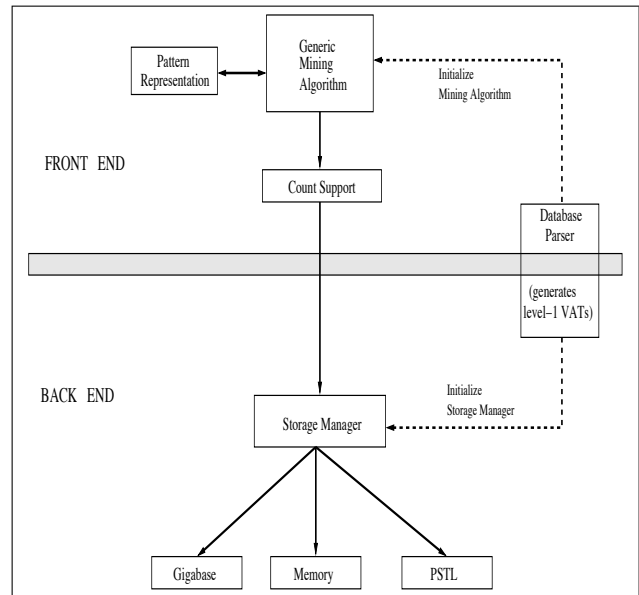


Figure 4: High-level Architecture Diagram of the Data Mining Template Library

5.1.1 Front-end: The Mining Engine

The mining task is initiated with all frequent patterns of length one. This step is performed by reading the data from a source. The source could either be a database, a flat file or another process that is generating the data. This functionality is performed by the *Database Parser* module (see figure 4). Then the generic algorithm generates unique candidate patterns through candidate generation and isomorphism checking, as we explained in section 3.1. The task of finding the support of each candidate pattern is delegated to the back end through the *Count Support* module.

5.1.2 Back end: The Storage Manager

Frequent pattern mining is often performed on very large datasets. Each iteration of the algorithm generates increasingly larger patterns, and the number of candidate patterns also grows enormously (especially, with low support) and does not fit in memory on most machines. In a vertical mining paradigm, associated with each pattern, a VAT also needs to be stored. Most mining algorithms do not provide explicit means of memory management nor is the issue addressed within the algorithm. The DMTL back end is dedicated to storage management, which stores the patterns, VATs, and the associated one-to-one mapping from patterns to their VATs. The back end also determines the support count of candidate patterns and returns it to the front end.

The current state of DMTL has multiple implementations of the back end—memory, Gigabase [9] and PSTL [7]—each one exporting the same interface. The Count Support module can select any one of these by using template arguments. Gigabase is an embedded object relational database which has its own storage management. It also stores elements (patterns, VATs) in its database file. PSTL is a library of persistent containers, akin to STL in its design. PSTL also achieves persistence by maintaining memory-mapped data

files. In both the above cases, the mining results and intermediate data (like VATs) are stored on disk and are available for processing at a later point. Thus, DMTL provides an elegant solution when a memory-based back end fails due to enormous growth of data. A flexible interface makes addition of a new storage manager type quite easy. We also considered using third party object stores as storage managers. Lack of flexible libraries for object storage prompted us to develop our own storage manager.

5.2 Data Types

The most vital data in DMTL are the patterns and their associated VATs. Patterns are implemented with a graph structure. Elements of a pattern are the vertex or edge labels of that graph. VATs are implemented using `std::vector`, as they store a list of transaction identifiers. And for the mapping between pattern and VAT, we use `std::map`. However, pattern structure plays the most important role in our generic mining algorithm, so we describe it further in the following section.

5.2.1 Pattern Structure

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 directed edges between vertices. Figure 5 shows the C++ class interface of the pattern concept that we mentioned in 3.1. It consists of the most basic operations expected from a type modeling such concept. A specific pattern (set, sequence, tree, etc.) is defined by enlisting the respective pattern properties (`pattern_props`). The `canonical_code` template parameter maintains a unique code corresponding to each pattern and is employed for isomorphism checking. It also provides binary inequality testing operations that can be used to implement the \leq operator for the pattern concept. The `graph_model` is the underlying data structure used for storing the above representation. A typical example of such a data structure is an adjacency list. This design decision to parameterize the storage type 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. Our design underlines the fact that loose coupling between key design components is crucial for the extensibility of a large software system. 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_out_edge(v1, v2, e)` methods. The Boost Graph Library (BGL) [15] provides a more complete set of graph representations and graph algorithms. At this moment we have refrained from using BGL's graph representations, primarily to keep the design flexible and open to various possibilities. In the future, we aim to utilize BGL's graph primitives to standardize our library. As seen in figure 3, the specific patterns are instantiations of the abstract pattern concept. Each such concrete concept is represented 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, from the implementation perspective we treat them as an ordered list of properties. This ordering of properties is necessary for the compiler to match a specialized pattern

```
template<class pattern_props, class graph_model,
        class canonical_code>
class pattern {

public:
    typedef vector<V_TYPE> VERTICES;
    typedef typename VERTICES::const_iterator
        CONST_VIT;

    bool add_vertex(const V_TYPE& v);
    bool add_out_edge(const V_TYPE& v1,
                     const V_TYPE& v2,
                     const E_TYPE& e);
    bool add_in_edge(const V_TYPE& v1,
                    const V_TYPE& v2,
                    const E_TYPE& e);
    CONST_VIT get_neighbors(const V_TYPE& v);
    CONST_VIT get_rmost_path();
};
```

Figure 5: Pattern Class Interface

to an appropriate super-pattern, if any algorithmic implementation is not available for that specialized pattern. This leads to the pattern hierarchy tree in figure 6. Note that in figure 3, a node can have multiple parents while in the pattern tree each pattern has a single parent. The importance of the single-parent characteristic becomes evident when we realize that selecting a super-pattern would lead to ambiguities in case of multiple super-patterns. Using this 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, figure 3 represents the conceptual (theoretical) side of the pattern mining problem whereas figure 6 represents the practical (implementation) side of the problem.

We had the following goals while constructing 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, and
3. Propagate absence of a lower-level concept implementation to a higher-level concept implementation.

The last objective above is a logical extension 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 directed graphs. Apart from being the most popular patterns, the core patterns can be considered to mark the 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. The following paragraphs describe the challenges faced in designing the library to achieve the first two goals.

5.2.2 Pattern Properties Implementation

In order to enable dispatching to the appropriate pattern we use the set of pattern properties as template parameters.

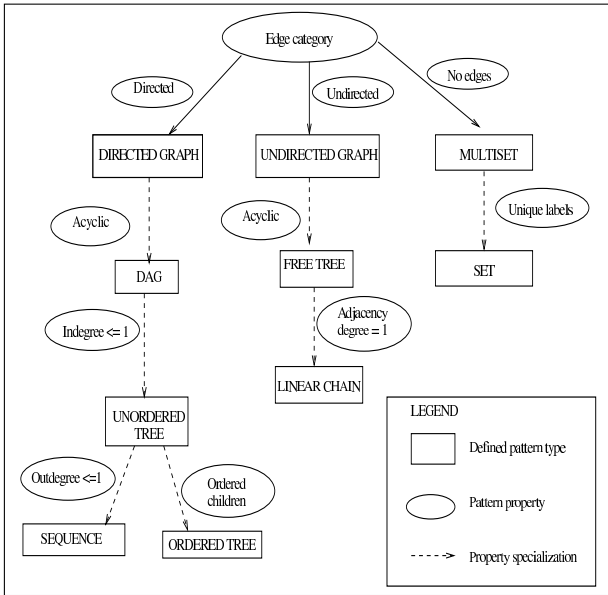


Figure 6: Pattern Hierarchy

This set of pattern properties is encapsulated in a **proplist**. Since we model properties as types, the *proplist* is a static list of types provided for collecting properties. 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. The type list was designed by borrowing ideas from two of the C++ Boost libraries—the Boost Graph Library and the Metaprogramming Library [?]. Since it is simply a container of types, the class itself is not complicated and is given in Figure 7. The class `null_prop` is used

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

Figure 7: proplist Class Interface

as the terminator of a type list. In addition to its utility as a type list, the **proplist** possesses the nice feature of facilitating upward propagation of properties. This behavior is demonstrated in Figure 8. To keep the example simple, we have stripped function parameters and return types that are not relevant for the example. In this example, we create property classes and give the prototype of a function that generates candidates from a given pattern. As pointed out above, candidate generation is one of the three tasks a mining algorithm must undertake. In the figure, two prototypes of the `candidates` function are provided—one for directed graphs and one for DAGs. DAGs do not possess cycles, hence the specialized `candidates` function does not generate cyclic graphs as candidate DAGs. On the other hand, the generic function generates all possible digraphs, includ-

```
/// Property class definitions ///
class directed {};
class acyclic {};
class planar {};
class null_prop {};

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

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

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

/// Following function call compiles ///
/// to generic function. ///
candidates(digraph);

/// Following function call compiles ///
/// to specialized function. ///
candidates(dag);

/// Following function call compiles ///
/// to generic function ///
candidates(planar_graph);
```

Figure 8: Application of Property Hierarchy

ing cyclic ones. This relation between DAGs and directed graphs is reinforced by the pattern hierarchy in figure 6). Hence, as expected, method calls with directed graph and DAG as their input parameter types would invoke the appropriate methods. The `planar_graph` property list is now introduced. It should be noted at this point that the pattern property `planar` is not defined in our library. Hence, it is a new pattern property for representing planar graphs. Let \mathcal{P}_1 denote the pattern type, digraphs, and \mathcal{P}_2 denote directed, planar graphs. Since the properties defining \mathcal{P}_1 are a subset of the properties defining \mathcal{P}_2 we can say $\mathcal{P}_1 \preceq \mathcal{P}_2$. As a result a `candidates` method call with `planar_graph` as input parameter will invoke the method with `digraph` as the formal parameter. Had there been a more efficient implementation for planar digraphs, that would have been invoked. To summarize, we have shown how the **proplist** can be used to select the most appropriate implementation and how a new pattern can be easily introduced into the framework.

5.3 Generic Algorithms

The core FPM algorithm shown in Figure 1 was introduced in section 3.1. 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.⁴ The pseudocode in figure 1 is implemented in the `freq_pat_mine` method.

⁴FP-tree is another approach for FPM. Since it is not as widespread as the apriori based approach, DMTL does not currently support it.

```

template<class PATTERN, class MINE_PROPS,
        class SM_TYPE>
void
freq_pat_mine(const pat_fam<PATTERN>& Fk,
              const pat_fam<PATTERN>&, int& min_sup,
              pat_fam<PATTERN>& freq_pats,
              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 and can be implemented as an STL vector or a list. The third parameter, `freq_pats`, which is passed by reference, is used to collect the final set of frequent patterns. Our customized containers either retain the same interface as the popular STL containers or are simply wrappers around STL containers. Note that in the above example `PATTERN` is the pattern representation. Hence it is not just a container parameter but is used to pick the most efficient implementation along the pattern hierarchy. The actual template argument could represent any pattern. As the name suggests, the `count_support` class is used for finding the support of the candidate patterns in the dataset. `count_support` is templated on the mining properties and back-end database type. The former is necessary because counting support differs for embedded and induced mining (which is a mining property). The later (`SM_TYPE`) is necessary for querying the appropriate storage manager to find the number of occurrences of a pattern. Let us take a closer look at some of the key steps inside `freq_pat_mine`.

5.3.1 Candidate Generation

Pattern types differ in how they generate candidates. However, there does exist significant commonality among the varying pattern types. This was explored by us in a previous work [23]. 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:

```

template<class PAT_PROPS,
        class MINE_PROPS,
        class SM_TYPE>
pattern<PAT_PROPS,
        MINE_PROPS,
        SM_TYPE>**
join(const
     pattern<PAT_PROPS, MINE_PROPS,
            SM_TYPE>* pat_i,
     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 is created from the join operation). Note that both the pattern properties and the mining properties are associated with the pattern type. Using pattern properties, the `join` method chooses the most appropriate algorithmic implementation to perform the join for this pattern type. Note that a join between patterns is associated with an intersection of the corresponding VATs. For example, if a pattern A is a set $\{a, b, c\}$ and another pattern B

is a 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.

5.3.2 Isomorphism Checking

For itemsets and sequences we can circumvent generating isomorphic patterns by intelligent candidate generation [1, 21]. 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 [22]. However, unordered trees [12], free trees [3] and graphs [19, 8] require isomorphism testing. The isomorphism checker is provided by the `check_isomorphism` method and it is templated on the pattern properties. Our library provides specialized isomorphism routines for various patterns—directed graphs and unordered trees, to name a few. The isomorphism checker can be used as a stand-alone component and we believe that it could further enrich the isomorphism checking support provided in BGL.

```

template<class PAT_PROPS,
        class MINE_PROPS,
        class SM_TYPE>
bool
check_isomorphism(pattern<PAT_PROPS,
                  MINE_PROPS,
                  SM_TYPE>* cand_pat)

```

5.3.3 Support Counting

The last step in an iteration is to determine the support of candidates, and discard ones that do not pass the `min_sup` (minimum support) criterion. The support counting functionality is supported by the *Count Support* block in figure 4. Since support counting needs to query the back end, this block acts as a liaison between the front end and the back end. The support counting module is common across all the pattern types, since it does not need to know anything about a specific pattern. At the same time the `count` method is independent of the back end since the `count_support` class is templated on the storage type. The interface for the `count` method is given below:

```

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

```

As we mentioned under Candidate Generation above, a join of patterns in the front end triggers an associated VAT intersection in the back-end. We provided different back-end implementations, all storing the same VAT but may be in different formats. For example, the VAT stored in the Gigabase database is necessarily different than that stored in the memory 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. Figure 9 shows how iterators hide the

```

template<typename InIter,
        typename OutIter>
void intersection(pair<InIter, InIter> itr_i,
                pair<InIter, InIter> itr_j,
                OutIter cand_vats);

```

Figure 9: Using Iterators with Generic Algorithms

data representation from the algorithms. The figure shows the signature of the `intersection` method, which joins two VATs to generate the VATs for new candidate patterns. The first parameter is a pair of iterators pointing to the beginning and end of the container that corresponds to the first VAT. Similarly, the second parameter is for the second VAT. The two iterators use the same `InIter` parameter since patterns have to be of the same type to be intersected. The third parameter represents an output iterator and is used to collect the set of generated VATs. Note that, depending on the pattern, more than one VAT could be generated.

To reiterate, the design of DMTL consists primarily of three challenging components:

1. pattern structure,
2. pattern algorithms, and
3. back end storage facility.

Along with the above key components, the library contains multiple smaller utilities for reading in data from multiple sources, parsing data in multiple formats, and many others.

5.4 Incorporating new patterns

Representing patterns as property-based concepts allows users to introduce new properties, and hence new patterns, with minimal changes to the code. This effectively allows us to mine any type of pattern. This idea of mining arbitrary patterns is novel and extremely desirable in the data mining community. Let us walk through an example to see how a completely new pattern can be mined. At this time we would like to remind the reader that our library currently implements only four key kinds of patterns—sets, sequences, trees and graphs. Each of these marks a new strata of pattern complexity. For this example let us say we want to mine all frequent cliques, given an input dataset containing graphs. A clique of a graph is a maximal complete subgraph. Suppose we want to mine all frequent k -cliques, where k is the number of nodes in the clique. Since a clique is a specialized graph, we can guess that the process of mining cliques might resemble that of mining graphs. Let us reconsider the three core steps required for mining any patterns and compare the functionality in each of those for the two patterns. 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. The alert reader might note that the task of mining cliques is similar to the task of mining itemsets. Although they are similar there is a subtle difference—itemsets are guaranteed to have unique labels whereas this is not the case with cliques. This argument reinforces our claim that

```

typedef proplist<directed,
                proplist<connected> > CLIQUE;

typedef proplist<directed > DI_GRAPH;

// Specialization for the clique pattern. //
template<class PAT, class MINE_PROPS,
         class SM_TYPE>
void
cand_gen(const pat_fam<proplist<directed,
                        proplist<connected, PAT> >& Fk,
         ....);

// Specialization for directed graphs //
// Can be used by cliques. //
template<typename T>
bool
check_isomorphism(pattern<proplist<directed,
                          T> >* cand_pat);

```

Figure 10: Adding a new pattern

cliques just differ in the isomorphism-checking step. Even though this example might seem contrived, it helps us see that a similar approach can be taken for any other pattern. In the worst case, the user will need to provide implementations for all three stages of pattern mining. From our experience with pattern mining, we can confidently claim that all the patterns in figure 6 along with many others need very few modifications on the part of the user. This has been the motivation behind the library design and implementation. Figure 10 shows the interface for the specialized candidate generation method for cliques. The first parameter is specialized to match a clique or any of its sub-concepts. The rest of the parameters have been omitted as they are not relevant to the example. Clique mining can borrow the remaining methods that are specialized for directed graphs.

6. CHALLENGES AND FUTURE WORK

The design and implementation of DMTL has helped us appreciate some of the language features provided by C++. While specialization by overloading, iterator categories, and similar powerful concepts are extremely important for generic programming, there are other aspects that are not equally well explored. Features such as *concept checking* and *named parameters* are features that would benefit our implementation. Moreover, dispatching based on concepts rather than pure type checking would allow partial specialization based on concepts. Even though some of these features have been implemented via template metaprogramming and made available in Boost libraries, our experience suggests advantages of including these features in the language standard.

The current design of DMTL has substantial scope for improvement. For example, our implementation of static lists to manage the pattern properties is not necessarily the best design choice. Such a property-list-based mechanism enforces a strict ordering of the properties in order for the compiler to select the appropriate specialization. Ideally, we would have benefited from the support for *named parameters* in C++. With such a feature we could omit the

properties that did not apply for a specific pattern and provide property in any order. While *named parameters* seems like a good option, it might result in changes to the interface while introducing newer properties in our framework. A different approach to handling dispatching in this scenario would necessitate support for concept based dispatching as against type matching based dispatching. Additionally, support for concept checking [17] in the language specifications would enhance development efforts. We also explored using the **PropertyGraph** concept in BGL to represent a set of properties but it did not fit well into our framework at that point without compromising flexibility. The `enable_if` family of templates is an approach for enabling certain function templates and class template specialization. It could be used to achieving the same effect as our property list approach. We hope to explore this opportunity with other ongoing development in DMTL. From the data mining perspective, DMTL provides quite an extensive set of FPM algorithms which perform better than existing stand-alone algorithms. Since DMTL has been an evolving idea, now it is ready for its first public release after undergoing numerous refinements to the design. Some performance results based on an earlier version of DMTL are presented in our previous work [23]. In the long term, we plan to incorporate mining algorithms in other pattern spaces such as maximal patterns and closed patterns. Our eventual goal is to extend DMTL to other data mining tasks like classification, clustering, and so on.

7. ACKNOWLEDGMENT

We would like to thank Professor David Musser for his suggestions and feedback at various stages of this project.

8. REFERENCES

- [1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *21st Int'l Conference on Very Large Data Bases*, 1994.
- [2] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.
- [3] Y. Chi, Y. Yang, and R. Muntz. Indexing and mining free trees. In *3rd IEEE International Conference on Data Mining*, 2003.
- [4] T. E. Fawcett and F. Provost. Fraud detection. pages 726–731, 2002.
- [5] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.
- [6] B. Goethals. Frequent pattern mining implementations repository. <http://fimi.cs.helsinki.fi/>.
- [7] T. Gschwind. PSTL—A C++ Persistent Standard Template Library. In *6th USENIX Conference on Object-Oriented Technologies and Systems*, 2001.
- [8] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. Technical Report TR03-021, University of North Carolina, 2003.
- [9] K. Knizhnik. Gigabase. <http://sourceforge.net/projects/gigabase>.
- [10] R. Kohavi, D. Sommerfield, and J. Dougherty. Data mining using MLC++, a Machine Learning Library in C++. In *8th Int'l Conference on Tools with Artificial Intelligence*, 1996.
- [11] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Second edition, 2001.
- [12] S. Nijssen and J. Kok. Efficient discovery of frequent unordered trees. In *1st Int'l Workshop on Mining Graphs, Trees and Sequences*, 2003.
- [13] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM/SIGMOD Int. Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 21–30, 2000.
- [14] J. Siek, D. Gregor, R. Garcia, J. Willcock, J. Järvi, and A. Lumsdaine. Concepts for c++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005.
- [15] J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.
- [16] J. Willcock, J. Järvi, A. Lumsdaine, and D. Musser. A formalization of concepts for generic programming. In *Concepts: a Linguistic Foundation of Generic Programming at Adobe Tech Summit, San Jose, CA*. Adobe Systems, Apr. 2004.
- [17] J. Willcock, J. Siek, and A. Lumsdaine. Caramel: A concept representation system for generic programming. In *Second Workshop on C++ Template Programming*, Tampa, Florida, October 2001.
- [18] I. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufman, 1999.
- [19] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. Technical Report UIUCDCS-R-2002-2296, University of Illinois at Urbana-Champaign, 2002.
- [20] M. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.
- [21] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42:31–60, 2001.
- [22] M. Zaki. Efficiently mining trees in a forest. In *8th ACM SIGKDD Int'l Conference on Knowledge Discovery and Data Mining*, 2002.
- [23] M. Zaki, N. Parimi, N. De, F. Gao, B. Phoophakdee, J. Urban, V. Chaoji, M. Hasan, and S. Salem. Towards generic pattern mining. In *International Conference on Formal Concept Analysis (Invited Paper)*, 2005.
- [24] M. J. Zaki, S. Jin, and C. Bystrhoff. Mining residue contacts in proteins. In *BIBE*, pages 168–175, 2000.
- [25] L. Zhao and M. J. Zaki. Tricluster: An effective algorithm for mining coherent clusters in 3d microarray data. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 694–705, New York, NY, USA, 2005. ACM Press.