DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION THEORY
BUDAPEST UNIVERSITY OF TECHNOLOGY AND ECONOMICS

APRIORI-BASED FREQUENT PATTERN MINING

by
Ferenc Bodon

PhD Thesis Summary

Supervised by
Dr. Lajos Rónyai
Tadeusz P. Dobrowiecki

Budapest, Hungary
2006
1 Introduction

Frequent pattern mining (FPM) is a relatively young subfield of data mining. It aims to find frequently occurring patterns in an input data set. We focus on the largest family of FPM, where the input data set is given as a sequence of transactions and the frequency of a pattern is given by the number of transactions that contain the pattern. The frequency threshold is set by the user. The types of the patterns and transactions may be of many kind, such as sets, sequences, trees, graphs.

Agrawal et al. presented the problem that appeared as a subproblem in analyzing supermarket transaction data to better understand customers’ behavior. The customers’ habits were modeled by association rules. This new approach turned out to be a very useful tool for marketing purposes. An association rule shows a connection between sets of products. For example the rule “digital camera $\Rightarrow$ memory card (85%, 1300, 3)” states that 85% of the people who bought a digital camera also bought a memory card. The two products were contained in 1300 baskets. This parameter is called the support of the rule. The third parameter tries to capture the dependence of the antecedent and consequence. It states that the relative frequency of the memory card is three times higher in the baskets that contain a digital camera than the relative frequency in all baskets. The most profitable pricing strategies, however, are based on association rules whose support parameter is high, i.e the products of the rule are bought by many people; they are frequent product sets.

To determine the interesting rules, the frequently occurring sets of products (itemsets) have to be mined first. Many algorithms were proposed to solve the Frequent Itemset Mining (FIM) problem efficiently. The methods and algorithms developed here turned out to be useful in other data mining fields like classification, clustering, and functional dependency discovery. The association rules and the frequent itemsets were used successfully not just in marketing but in other application domains such as improving the efficiency of electronic commerce, bioinformatics, DNA and protein analysis, medical diagnosis, inductive databases, query expansion, Web log and bank account analysis, etc.

The basic FIM problem was extended in many applications, which has led to the introduction of hierarchical, negative, incremental, parallel and quantitative FIM. FIM with non-universal support threshold, temporal, cyclic and fuzzy association rules. In some application domains the itemset model was not general enough. There are efficient algorithms to mine frequent sequences, trees, graphs, and Boolean formulas. These generalizations also enjoy a broad range of applications, including webpage personalization, stock sequence and chemical compound analysis, etc.

In this work we concentrate on the basic FPM problem.

2 The importance of Apriori-based approach

Algorithm Apriori proposed by Agrawal et al. in 1994 was the first FIM algorithm that could cope with large datasets. Its modifications and improvements ruled the world of FIM till the prefix based, depth-first, recursive algorithms such as Eclat in 1997 and FP-growth in 2000 entered the scene. A highly optimized Apriori algorithm proposed by Orlando et al. called DCI that adapted a hybrid support count function overtook the gold medal in the arena of FIM implementations in 2002.
The open source competitions of FIM implementations (FIMI) brought many interesting observations. First of all, it has become clear that there exists no best algorithm that beats all the other algorithms on every database. Second, the basic Apriori algorithm outperformed many new and sophisticated algorithms and in fact, Apriori turned out to be the best at extremely large databases where FP-growth and Eclat based algorithms aborted due to the large memory requirement. Third, the improved version of DCI was among the four best implementations. These facts prove that Apriori is an important and appreciated member of the family of FIM algorithms.

Moreover, the merits of Apriori do not end with border or itemset discovery. Its adaptations in frequent sequence, tree and graph mining are also among the most important and most efficient algorithms. Techniques for decreasing the run-time and memory requirement are often applicable in searching for frequent patterns of other types. Our experiments proved, for example, that a naive adaptation of a basic Apriori to sequence mining often outperforms an adaptation of a highly optimized Eclat version. It is not clear that prefix-based, depth first recursive algorithms outperform Apriori-based solutions, if we step out of the world of itemsets.

2.1 FIM, Algorithm Apriori

The Frequent Itemset Mining problem is the following. Given a sequence of itemsets, find the frequent ones among them. An itemset is frequent if it occurs as a subset more than a given number of the input dataset.

The algorithm scans the transaction datasets several times. After the first scan the frequent 1-itemsets are found, and in general after the $\ell$th scan the frequent $\ell$-itemsets are extracted. The method does not determine the support of every possible itemset. In an attempt to narrow the domain to be searched, before every pass it generates candidate itemsets and only the support of the candidates are determined. In each iteration three major steps are repeated: candidate generation, support count of the candidates, removing infrequent candidates. An itemset becomes a candidate if every proper subset of it is frequent. Due to the bottom-up search, all frequent itemsets of size smaller than the candidate are already determined, therefore it is possible to do the subset validations.

After all the candidate $(\ell + 1)$-itemsets have been generated, a new scan of the transactions is effected and the precise supports of the candidates are determined. The candidates with low support are thrown away. The algorithm ends when no new candidates are generated.

The intuition behind candidate generation is based on the simple fact the every subset of a frequent itemset is frequent. This is immediate, because if a transaction $t$ contains an itemset $X$ then $t$ contains every subset $Y \subseteq X$.

3 Research Objectives

This thesis deals with the Apriori based solution of the basic frequent pattern mining problem. Apriori is one of the oldest, and it is without doubt the most widely-known solution. Despite the immense research efforts focused on the method and its applications, some major data structure/implementation issues and possible speed-up methods have left without much consideration. Our main goal was to fill this void.
4 Research Methodology

Those who believe that their work is of high value, often say that the main problem of frequent pattern mining is the lack of reproducibility and the impossibility of verification.

In the beginning of the FPM era a typical paper proposed some new techniques, reasoned with some intuitive, informal thoughts and showed its efficiency on some carefully generated datasets. This procedure led to indignation, because the efficiency of the implementation of the rival algorithm was often significantly below the efficiency of the implementation done by the original authors. The generality, drawbacks, limits of the proposed algorithm were rarely discussed.

Fortunately, this era quickly closed after some famous implementations were made publicly available, and at the conferences of high standards it was required that the proposed algorithms be compared with the known implementations. The level was raised further by the two FIMI competitions. Now we have ultrafast FIM implementations, nevertheless nobody exactly knows why do they perform so well, what are the limitations of the solutions, what kind of input data they prefer. They are like black-boxes, and only the authors can change the parts of the implementation, which is attributed to the highly optimized, non-object oriented codes, which are almost impossible to read by other researchers.

4.1 FIM template library

If we would like to understand the performance effect of all parts of a code, we have to make it modularized. This is not a trivial task in a highly optimized environment. In [C3] we presented some techniques, which are based on templated class hierarchy and inline functions, to make a code object-oriented without sacrificing efficiency. To achieve a perfect FPM world, object oriented codes are not enough yet. The codes have to be in a library, where any part of an implementation can be replaced by an other element of the same functionality and any technique can be switched on and off. This way each part of an algorithm can be tested separately and together with other techniques. We can measure how does a certain solution contribute to the final performance, how do different techniques assist or hold back each other.

These principles were followed in building up our FPM template library, which contains our fully pluggable Apriori, Eclat and FP-growth implementations that are competitive with (and in most of the cases outperform) the black-box implementations. For example in our Apriori algorithm different template classes are responsible for doing the support counting, the candidate generation, coding and decoding the items, caching the transaction. All techniques, like dead-end pruning, equisupport extension, etc. can be turned on and off by a template parameter. The data structure is also a template parameter. If it is a trie, then the representation of the list of edges is given by an other template class, in which even the vector representation is pluggable, therefore we chose STL vector or our lightweight, self-made vector.

The FPM template library made possible to conduct a comprehensive set of experiments with reasonable effort. In a black-box system this would have required a lot of laborious and error-prone work. The library is made publicly available and started to be used by other researchers.

The contributions of the thesis are supported by several thousand experiments. All techniques are tested on 16 benchmark databases with 7 support thresholds. The bench-
mark databases involve synthetic as well as real-world data. To really test the implementation, the databases have different characteristics: some are dense/sparse and some contain many non-closed frequent itemsets.

4.2 Hardware friendliness diagrams

The classical run-time and memory requirement evaluation is satisfactory if we want to compare different algorithms or the efficiency of a technique. Nevertheless, if we would like to understand the outcome, more sophisticated “profilers” are needed. To this end we presented hardware friendliness diagrams that serve information about the usage of processor and processor features and the usage of memory of different levels.

An example for such diagram is depicted in below.

![Diagram showing hardware friendliness](image)

The height of the wide bars centered around the ticks show the actual run-time (the total clockticks used by the program). The colors/patterns of these bars show how well the program utilized these clockticks: the top-most part shows the amount of clockticks during which three u-ops were executed, while the bottom-most part shows the time during which the program execution was stalled for some reason (i.e., no operations were executed during that clocktick).

The narrow bars centered around the ticks show the total number of u-ops that were executed. The bar is divided into two, the upper part show the bogus u-ops, those u-ops that were speculatively executed on a mispredicted branch, and thus were rolled back. The ratio of the lower-to-upper part of this bar shows the branch prediction inefficiency.

The narrow bars beside the wide ones show the front-side bus activity, the total number of clockticks during whose at least one read/write operation was pending (i.e., data transfer time including memory latency). The upper part of these bars show the time consumed by prefetch reads (when the processor speculatively transfers data from the memory into the cache for further availability), while the lower part shows actual reads or writes. The main difference is that the delivery of data during actual reads and writes presumably stalls the execution pipeline (these are the cache misses). If the ratio of prefetch (top part) to actual wait (bottom part) is high, then a huge amount of cache misses are avoided by the prefetch mechanism, thus achieving a considerable performance gain.

4.3 Safe and hardware independent techniques

Many techniques are proposed by the FIM research community in the recent years. Most of them are unsafe, i.e., they work satisfactorily for databases with certain characteristics, but ruin efficiency, if the input data does not belong to the “preferred” group.
The limitations, weaknesses and sensitivities of the techniques are not discussed by the authors. In contrast, the methods proposed by us are safe. Applying them may lead to speed-ups of several orders of magnitude, while the efficiency is not ruined if the database is not preferred by the technique (e.g. a technique based on equisupport pruning is never applied if the database contains no non-closed itemsets). Note that this kind of run-time and memory safety is not attributed to most of the well-known techniques, including the ones used in algorithms Apriori-Hybrid, DIC, DHP, Partition, Toivonen, etc. or in support lower bound and iteration merging solutions, etc.

Our techniques are also hardware independent, i.e., we believe that they result in more efficient algorithms on all types of modern processors. Our beliefs are supported by run-time results on different processors, but more importantly, we present hardware friendliness diagrams. These diagrams provide much more information than single run-times and allow the generalization to other processors. Note that this does not hold for run-time based consequences. Let us assume that we propose a method and the experiments show that a new algorithm is faster on Pentium 4 processors. By using profilers and processor counters we understand the reasons of the speed-up. The technique improves the branch prediction facility of the processor, nevertheless the technique decreases data locality, which results in more cache misses. In this case we can not conclude that the technique will also result in a faster algorithm on other architectures. For example in AMD processors the pipeline is shorter therefore improving branch prediction may not reduce run-time as much as the increased number of cache misses decrease it. Our experiments on hardware friendliness show that our techniques are not “hardware specific”, improving the efficiency of a processor feature never ruins the efficiency of an other feature. The techniques reduce the number of instructions without ruining branch prediction and data locality.

The hardware friendliness analysis is a new approach, which helps better understand data mining solutions. Since this approach has been worked out together with Balázs Rácz and Lars Smidt Thieme, it is not part of this thesis as a contribution.

5 New Results

The results are classified into four areas. The first and the second claims are about data structure and speed-up techniques of Apriori algorithm for frequent itemset mining. The third claim establishes a theoretical foundation of the general frequent pattern mining problem. The last result is about an application of frequent pattern mining.

Claim 1 : Improving the support count of Apriori

The main bottleneck of Apriori is the support count method. To solve this computation-intensive task, the authors of Apriori proposed hash-tree to store the candidates.

Claim 1.1: We introduced inhomogeneous Trie, with special block allocator and hybrid edge representation.

The drawback of hash-trees is the lack of universally good hash-function. A hash function that is good at a certain dataset performs often quite poorly at other databases [J1]. Here, we propose an Apriori-optimized version of trie [J1], [J4], [C6], [C9]. Our
inhomogeneous trie uses a special block allocator and hybrid edge representations (that allows a hybrid support count technique) to store candidate itemsets. This highly optimized solution achieves a much better data locality and fewer number of operations than the traditional implementation of a trie, and hence results improved running time.

A trie is a rooted, labeled tree. Each label is a character and each node represents a word (sequence of characters) which is the concatenation of the characters that are on the path from the root to the node. The root is defined to be at depth 0, and a node at depth \( d \) can point to nodes at depth \( d + 1 \). A pointer is also referred to as edge or link. Tries are suitable for storing and retrieving not only words, but any finite sequences over arbitrary alphabet as well. In the FIM setting a link is labeled by a frequent item, and a node represents a sequence of items. To obtain a sequence from a set, we have to define a total order on the items. For this we always use the same order that is used to order the edges. In this case the preorder depth-first search traversal corresponds to the ascending lexicographical ordering of the itemsets.

A trie that stores all subsets of a given set is quite unbalanced. Figure 1 shows the trie that stores all subsets of itemset \( \{ABCDE\} \).

![Trie Diagram](image)

Figure 1: Example: a trie that stores all subsets of itemset \( \{ABCDE\} \)

Our inhomogeneous trie is not implemented by a single recursive structure but the leaves are described and stored separately. This is inspired by the fact that during a support count the itemsets belonging to sibling leaves are often contained in the transaction considered. To reduce the number of cache misses they should be as close to each other in the memory as possible. In the naive tree implementation the leaves are drifted from each other because of the overhead of an empty list. These overheads are spared by the separate storage of leaves. Our solution is further improved by a special block allocator, that avoids the overhead of built-in memory allocation.

The list of edges belonging to an inner node can be represented in many ways. The representation used in the algorithms greatly affects run-time and memory-need. In the case of ordered-edge list representation each edge is represented by a pair, whose first element is the label, and the second is a pointer to the child. The edges are stored in a vector, which is ordered according to the labels.
If we use indexvector representation then the child pointers are stored in a vector whose length equals to the number of frequent items. A node at index $i$ is the endpoint of the edge whose label is item $i$. If there is no edge with such label then the element is NIL. Obviuously the elements at index less than the smallest label and greater than the largest label are NIL. We save memory if these elements are not stored. In offset-indexvector representation the smallest element (the offset) and a pointer vector of size $l_{\text{max}} - l_{\text{min}} + 1$ is stored, where $l_{\text{min}}$ and $l_{\text{max}}$ denote the smallest and largest label of the edge, respectively.

We also propose hybrid edge representation. Based on the memory need we dynamically choose between ordered-edge and offset-index representation. This results in a better data locality and a more compressed trie, therefore the algorithm is faster and consumes less memory.

The hybrid edge representation makes us possible to perform a hybrid routing strategy. Routing strategy at an inner node refers to the method used to select the edges to follow during the recursive traversal of the support count method. In hybrid routing look-up by index is used if the edges are represented by offset-indexvector and simultaneous traversal in the case of ordered lists. There are many other possibilities in the latter case. Nevertheless our hardware friendliness analysis show that simultaneous traversal greatly exploits the prefetch feature of the processors, which results in a good performance.

Throughout the algorithm one child-linked trie is maintained. In this trie a counter is associated with each node. This counter stores the support of the itemset that is represented by the node. In candidate generation phases new leaves are added with zero counters, in support count phases the counters are updated, and when we eliminate infrequent subsets (infrequent removal phase), leaves with counter value less than the support threshold are pruned.

Beside examining data structure issues, we also investigate algorithmic solutions in order to speed-up support count. We developed three new methods.

**Claim 1.2: We embed dead-end pruning into candidate generation.**

Frequent itemsets of size $\ell$ are only needed in (1) writing out the results and (2) generating candidates of size $\ell + 1$. The results can be written out either in candidate generation or at the infrequent candidate removal phase. In candidate generation some leaves are extended (if adding an item to its representation results in an itemset all of whose subsets are frequent) some are not. This means that there are leaves that represent candidates and there are leaves that do not. We call the second kind of leaves dead-end leaves and a subtrie is a dead-end branch if all its leaves are dead-end leaves [C5]. Dead-end branches are also generated in infrequent removal phase. If all (or all but one) children of a node are infrequent then the node becomes a leaf and is never extended again.

The nodes of a dead-end branch are not needed for candidate generation thus its nodes’ itemsets can be written out and such nodes can be pruned from the trie. This technique has many advantages. First, the trie gets smaller. Second, the support count is faster.

Dead-end branch pruning does not require any movement in the trie, if the nodes in the candidate generation phase are visited in a preoder depth first manner. This is based on Property 5.1.
Property 5.1 Let $I$ be an itemset and $<$ an ordering on the elements of $I$. $I$ strictly precedes all subsets of $I$ according to the lexicographical ordering based on ordering $<\,$.

Consequently, an itemset $I$ can be a subset of those candidates whose generators strictly precede $I$ in the preorder traversal. Therefore a node can be pruned if no new candidates are generated from any descendants of it.

Claim 1.3: We propose Patricia trees to cache the transactions.

I/O and string to integer parsing costs are reduced if the transactions are stored in the main memory instead of disk. It is useless to store the same transactions multiple times. Instead, store them once and employ counters representing the multiplicities. This way, memory is saved and run-time may be significantly decreased [C5; T1].

The advantage of this idea is the reduced number of support count method calls. If a transaction occurs $n$ times then the expensive procedure is called just once (with counter increment $n$) instead of $n$ times (with counter increment 1). Thus the number of calls to the most expensive method may be considerably reduced. Unfortunately, the data structure needs memory, and building it up (i.e., collecting the same transactions) requires processor time.

We refer to the data structure that stores the transactions together with the multiplicities as transaction cachers. The transactions are cached after the first scan, so that infrequent items can be removed from the transactions. Different data structures can be used as transaction cachers. We have three requirements:

1. inserting an itemset has to be fast,
2. the data structure has to be memory-efficient,
3. listing the transactions and the multiplicities has to be fast.

A simple solution is an ordered vector, each element stores an itemset and its multiplicity counter. Inserting a transaction becomes slow as the number of transactions becomes large. In such cases tree-based solutions, like trie or red-black tree are more efficient. Tries provide faster algorithms, but they need more memory.

Both approaches are outperformed by Patricia trees, which overcome the defect of a trie that stems from the inefficient storage of single paths. It substitutes a single path with one link with a label equal to the set of labels that are on the path. This spares many pointers but more importantly, the memory requirement caused by the overhead of a list is greatly reduced. Thus Patricia trees keep the advantage of trie-based solution without suffering from large memory need.

Claim 1.4: We extend equisupport pruning to handle databases with many non-closed itemsets more efficiently.

The last two speed-up techniques reduce run-time greatly (sometimes the gain is of several orders of magnitude), but still they do not make it possible to process dense datasets efficiently, in which many nonclosed itemsets occur. Our new equisupport pruning with a special dead-end pruning raises Apriori to the group of those FIM algorithms that are able to cope with dense databases. The technique is based on the following lemma that states that the support of certain itemsets can be calculated directly from the support of some subsets.
Property 5.2  Let $X \subseteq Y \subseteq \mathcal{J}$. If $\sup(X) = \sup(Y)$, then $\sup(Y \cup Z) = \sup(X \cup Z)$ for any $Z \subseteq \mathcal{J}$.

Prefix equisupport pruning claims that if candidate $Y$ has the same support as its prefix $X$, then it is not necessary to generate any superset $Y \cup Z$ of $Y$ as a new candidate. Based on the above property its support can be calculated directly from subset $X \cup Z$.

The technique works in the following way. After determining the supports of the children of itemset $P$, we check at the infrequent removal phase if their support are equal to $\sup(P)$. Children with such supports are not considered as generators in later phases and the extending items that belong to them are stored in a set (called equisupport set) and associated with itemset $P$. Notice, that due to the non-redundant traversal of the itemset lattice, $Y \setminus X \prec z$ for all $z \in Z$ where $\prec$ denotes the order used to define the prefix. When writing out a frequent itemset $I$, we also output the union of $I$ with itemset $E'$ for all $E' \subseteq E$, where $E$ is the union of all equisupport sets for the prefixes of $I$.

In the thesis we investigate why we cannot use any subset in equisupport pruning in trie-based Apriori algorithms. We introduce Level 2 equisupport pruning, which takes place in the candidate generation phase. This Level 2 equisupport pruning enables us to apply a special dead-end pruning that further improves efficiency of Apriori in databases that contain many non-closed itemsets [T1].

Claim 2: Improving the candidate generation of Apriori; The consequences of order-preserving assumptions

As a result of the above techniques, the run-time of support count reduces so much that the candidate generation method becomes the bottleneck of Apriori at many databases.

Claim 2.1: We introduce an intersection-based pruning in order to generate the candidates of a given prefix much faster.

To achieve further improvements, we propose an intersection-based candidate generation technique [C2], that determines all possible candidate extensions of a given itemset in one step. This technique requires a modest fraction of movements in the candidate trie, compared to the classical method.

Let us assume that we want to add new leaves to node $P \cup x$, where $P$ denotes the prefix. When checking the subsets of itemset $P \cup \{x, y\}$, we check $P \cup x$, $P \cup y$ and $Q \cup \{x, y\}$ where $Q \subseteq P$ and $|Q| + 1 = |P|$. $P \cup x$, $P \cup y$ are the generators, they have to be frequent. Therefore when checking the subsets of $P \cup \{x, y\}$ it is enough to examine if item $y$ extends nodes $Q \cup x$ for all subsets $Q$. Similarly, when checking subsets of $P \cup \{x, z\}$ we examine if item $z$ extends nodes $Q \cup x$ for all $Q \subseteq P$. Consequently node $P \cup x$ is extended by those sibling items that extend all $Q \cup x$ nodes, i.e. the extending set equals to the intersection of labels of edges that start from nodes $Q \cup x$. This is the point where we save the traversals. If nodes that represent $Q$ itemsets are stored, then checking the subsets of $P \cup \{x, z\}$ means determining the child nodes of $Q$ nodes that are reached by label $z$ and doing the intersection. Furthermore, if the edges are stored ordered and we store the index of edges used in the actual search (and at the starting
point in the next search), then in determining the items that extend the children of \( p \) the edges of all \( Q \) nodes are traversed at most once.

In intersection-based candidate generation when extending the children of \( P \), we first find nodes \( Q \), where \( Q \subseteq P \), \( |Q| + 1 = |P| \). Then we take each label \( i \) of nodes that start from \( P \) and determine if \( x \) extends all \( Q \) nodes. If not, then \( P \cup x \) can not be extended, otherwise we take the intersection of the extender labels of \( Q \cup x \) and the label of siblings \( P \cup x \). The elements of the result set are the items that extend \( P \cup x \), because they meet the complete pruning requirement.

**Claim 2.2:** We show that by omitting complete pruning during the candidate generation we obtain faster algorithms at most databases.

Frequent pattern mining is full of beliefs that turned out to be false. One such misbelief is that the efficiency of Apriori roots from its ability of performing complete pruning, i.e., a candidate is not generated, if it has an infrequent subset. On the contrary, we show that if a proper ordering is used, then a slight modification of Apriori that does not employ complete pruning, outperforms traditional Apriori at most databases.

The advantage of the pruning is to reduce the number of candidates. The number of candidates in Apriori equals to the number of frequent itemsets plus the number of infrequent candidates, i.e., the negative border of the frequent itemsets (denoted by \( NB(F) \)). If pruning is not used, then the number of infrequent candidates becomes the size of the order-based negative border of the frequent itemsets (\( NB^\prec(F) \)), where the order corresponds to the order used in converting the sets to sequences (An itemset \( I \) is an element of the order-based negative border of \( F \) if it is not in \( F \), but its prefix \( I[|I|-1] \) and the subsequent subset of \( I \) of the same size are in \( F \)). It follows, that if we want to decrease the redundant work (i.e., determining a support of the infrequent candidates), then we have to use the order that results in the smallest order-based negative border. This issue is further analyzed in Section 4.4 of the thesis, where we claim that the ascending order according to supports is expected to result in the minimal negative border.

A disadvantage of the pruning strategy is simple: we have to traverse some part of the trie to decide if all subsets are frequent or not. Obviously this needs some time.

Here [C2] we state that pruning is not necessarily an important part of Apriori. This statement is supported by the following observation, that applies in most cases:

\[
|NB^\prec_{\text{asc}}(F) \setminus NB(F)| \ll |F|.
\]

The left-hand side of the inequality gives the number of infrequent itemsets that are not candidates in the original Apriori, but are candidates in Apriori-Noprun. So the left-hand side is proportional to the extra work to be done by omitting pruning. On the other hand, \(|F|\) is proportional to the extra work done by pruning. Candidate generation with pruning checks all the maximal proper subsets of each element of \( F \), while Apriori-Noprun does not. The outcomes of the two approaches are the same for frequent itemsets, but the pruning-based solution determines the outcome with much more effort (i.e., traverses the trie many times). Another advantage of Apriori without complete pruning is that the infrequent removal and candidate generation phases can be merged without the chance of performance penalty.
Claim 2.3: We introduce order-preserving assumptions so that we can formally handle many known but unexplained observations.

The previous issue leads to the analysis of ordering used to convert itemsets to sequences, which greatly influences both run-time and memory requirement. We investigate the ordering issue and show which orderings are favored by different techniques and methods of Apriori.

We reject the independence assumption (which declares that the frequency of itemset $I_1 \cup I_2$ equals to the product of the frequencies of $I_1$ and $I_2$) and present a new, more versatile family of notions, the order-preserving assumptions, which make possible to formally prove the effectiveness of many heuristics. The basic order-preserving assumption is the following [T1]:

**Definition 5.3** The order-preserving assumption requires that $\text{sup}(X \cup Y) \leq \text{sup}(X \cup Z)$ holds whenever $\text{sup}(Y) \leq \text{sup}(Z)$, for any sets $X, Y, Z$.

Using this assumption we can formally prove many observations, that were explained sketchy ways and without sufficient detail by different authors in the literature. One of the most important consequences is the following,

**Lemma 5.4** If the order-preserving assumption holds, then

$$NB(S) = NB^{<\text{asc}}(S),$$

where $S$ denotes a set of itemsets, in which downward closure property holds, and $<\text{asc}$ denotes the ascending ordering according to the supports. $NB(S)$ and $NB^{<\text{asc}}(S)$ stands respectively for the negative border and the order-based negative border of $S$.

Claim 3: We introduce a unified framework of the frequent pattern mining.

One of the main reasons for the chaos in frequent pattern mining is the lack of a unified theoretical framework. Concepts, techniques, ideas introduced in a special subfield of FPM are often useful in other subfields of FPM and are often introduced independently by others. Our research in frequent itemset sequence mining has shown that techniques and solutions of frequent itemset mining provide a good basis for an efficient algorithm in frequent sequence mining. Didactically we would say, that the hierarchy of the type of patterns should be examined through a "sliding window". Many techniques are applicable in the next generalization level, many of them are out of question, and also new problems enter the scene. Here, we present a partial order based unified framework of frequent pattern mining, and show some examples of treating certain techniques and notions within this framework. The basis of the description is the pattern context [B2], [C4], [C5].

**Definition 5.5** The poset $PC = (P, \preceq)$ is called a pattern context if (1) it is locally finite, (2) graded and (3) there exists exactly one minimal element of $P$. The elements of $P$ are called patterns.
The first element of the pattern context is the pattern space. We say \( p' \) is a subpattern of \( p \) and \( p \) is a superpattern of \( p' \) if \( p' \leq p \). The minimal element of a pattern space is called the empty pattern and denoted by \( \emptyset \). Without loss of generality we assume that \( |\emptyset| = 0 \). A pattern of size \( \ell \) is also called an \( \ell \)-element pattern.

The most widely known pattern space is the \( PC_{\text{SET}} = (2^J, \subseteq) \), where \( J \) is a given set and \( \subseteq \) denotes the traditional subset relation for sets. Similarly, we can define pattern context that captures item sequences, sequences of itemsets, directed labeled ordered/unordered trees, directed acyclic graphs, labeled graphs or Boolean formulas.

To illustrate the importance of this unified framework we show how to describe the notion of closed patterns and a famous algorithm by Toivonen in this environment.

**A sketch of Toivonen’s algorithm**

The input data in frequent pattern mining is very large in general. This leads to a long processing time, which can be reduced by sampling. In Toivonen’s algorithm we take a part of the input database, determine the frequent patterns (denoted by \( F' \)), and then we determine their support in the original database. This solution is obviously not complete. If a pattern is not frequent in the chosen part then it is not found to be frequent. Therefore the algorithm also determines the support of the minimal, proper upper bound (MPUB) of \( F' \). Formally:

\[
\text{MPUB}(F') = \{ p : p \not\in F', \forall p' < p, p' \in F' \}.
\]

Based on the support of \( \text{MPUB}(F') \) the algorithm may declare that it found all frequent patterns. For this it uses the following lemma.

**Lemma 5.6** Let \( D \) be an input database and \( D' \) a sample of it. The frequent patterns in \( D \) and \( D' \) are denoted by \( F \) and \( F' \) respectively. The set of patterns that are frequent and are also in \( F' \cup \text{MPUB}(F') \) is denoted by \( F^* \); i.e., \( F^* = F \cap (F' \cup \text{MPUB}(F')) \). If

\[
F^* \cup \text{MPUB}(F^*) \subseteq F' \cup \text{MPUB}(F'), \tag{1}
\]

then the set of frequent patterns equals to \( F^* \).

**Proof:** We prove the statement by contradiction. Let us assume the equation (1) holds, but there exists \( f \in F \), but \( f \not\in F^* \). Because of the definition of \( F^* \), \( f \) is not element of \( (F' \cup \text{MPUB}(F')) \). Let us examine the smallest \( f' \leq f \), such that \( f' \in F' \) but \( f' \not\in F^* \). Such \( f' \) must exist, because \( f' \) may be identical to \( f \). From the minimal property of \( f' \) it follows that all proper subsets of \( f' \) are in \( F' \cup \text{MPUB}(F') \). Then all proper subsets of \( f' \) are element of \( F^* \) and \( f' \in \text{MPUB}(F^*) \). This is a contradiction, because there is an pattern \( (f') \) that is element of the left side of equation (1) but not element of the right side. \( \square \)

Toivonen’s algorithm was presented to mine frequent itemsets. Thanks to this more general description of the algorithm it is possible to apply it to other types of patterns like trees or graphs. If we define the subpattern relation, then the adaptation of the algorithm to a given pattern context is straightforward.

**The theory of closure**

**Definition 5.7** A pattern \( p \) is closed if there exists no other pattern \( p' \) with \( \text{sup}(p) = \text{sup}(p') \) and \( p \prec p' \).

Closed patterns are interesting because of two reasons. First, the support function is anti-monotonic; therefore we expect that any superpattern of \( p \) has smaller support than \( p \), and equality of support brings interesting insights in many applications. Second, frequent closed patterns turned out to be a compact representation of the frequent patterns in the case of itemsets. We state that this property does not apply to other types of patterns.
**Definition 5.8** Pattern $p'$ is a closure of pattern $p$, if $\text{supp}(p') = \text{supp}(p)$ and there exist no other $p''$ such that $p' < p''$ and $\text{supp}(p'') = \text{supp}(p)$.

**Definition 5.9** The pattern context $(P, \preceq)$ is unique with respect to closure, if the closure of any pattern is unique.

**Lemma 5.10** $PC_{\text{SET}}$ is unique with respect to closure.

**Lemma 5.11** $PC_{\text{SEQ}}$ is not unique with respect to the closure.

**Proof:** We prove this by a contradictory example. In database $\mathcal{I} = \{\langle A, B \rangle, \langle B, A \rangle\}$ the following holds: $2 = \text{supp}(\emptyset) = \text{supp}(A) = \text{supp}(B)$ and no proper superpattern of $\langle A \rangle$ and $\langle B \rangle$ has support 2. Therefore the empty pattern has two closures. $\square$

Since $PC_{\text{SEQ}}$, $PC_{\text{TREE}}$, $PC_{\text{DAG}}$, $PC_{\text{GRAPh}}$ are the generalizations of $PC_{\text{SEQ}}$, we obtain:

**Corollary 5.12** $PC_{\text{SEQ}}$, $PC_{\text{TREE}}$, $PC_{\text{DAG}}$, $PC_{\text{GRAPh}}$ are not unique pattern contexts with respect to closure.

**Claim 4:** We develop a non-redundant frequent episode mining algorithm for filtering frequently occurring false alarms in computer networks.

The security of computer networks is a prime concern today. Various devices and methods have been developed to offer different kinds of protection (firewalls, IDS’s, antiviruses, etc.). By centrally storing and processing the signals of these devices, it is possible to detect more cheats and attacks than simply by analyzing the logs independently. The most difficult and still unsolved problem in centralized systems is the vast numbers of false alarms. If a harmless pattern, which caused by safe operation is identified as an alarm, then it is a nuisance and requires human intervention to be handled properly.

In the thesis we show how we can use data mining to discover the patterns that frequently cause false alarms. Due to the new requirements (events with many attributes, invertible parametric predicates) none of the previously published algorithms can be applied to our problem directly. We present the algorithm ABAMSEP, which discovers frequent alert-ended episodes [J2], [J5], [C7], [C8], [C18]. We prove that the algorithm is correct in the sense that it finds all episodes that meet the requirements of the specification.

**6 Application of the Results**

After several years of development and research our Apriori implementation became one of the fastest and most memory-efficient implementation. It achieves this strong performance without sacrificing the object-oriented approach. The following figure shows the advantage of our Apriori over a famous program, the Apriori-Borgelt-FIMI in
terms of run-time and memory-need. Values greater than one mean that our Apriori was faster than Apriori-Borgelt-FIMI.

Our Apriori implementation is ranked to the first place by Google, it is downloaded many times by researchers, students and users from all around the world; the website of the code and the introductory paper is cited by many data mining papers [5] [4] [3] [2] [8] [6]. The implementation is used in basic research of frequent pattern mining and in applied research of other domains. Here we list a few of them.

In an excellent paper about upper and lower bound for the support of itemsets, our Apriori implementation was chosen and extended to support the theoretical foundation with experimental results [8]. Algorithm Apriori is also used in recommender systems. Our code was employed in a subspace clustering approach [1].

A genome research group of Sanger Institute (University of Cambridge) is using the sequence mining version of our Apriori [C4] in analyzing DNA sequences. In the business application domain our implementation is employed in detecting performance antipatterns in component-based enterprise systems. For more information the reader is referred to [7].

The most important application of our code may be in epilepsy research conducted by the Intelligent Control Systems Laboratory (ICSL) at Georgia Institute of Technology. Literature suggests that seizure precursors within the electroencephalogram (EEG) may identify brain tissue that is key to the generation of recurrent seizures in patients with epilepsy. Their project investigates if seizure precursors can be exploited to track the development of the diseased tissue, thereby mapping an “epileptic network” in the brain.

The objective of the research at ICSL is to develop an automated methodology to map the epileptic network based upon the detection and statistical analysis of seizure precursors. Our program was one of the data mining tools they employed to process their extremely large datasets. The volume of data made it impossible for them to work with known commercially available systems (such as SAS Enterprise miner, SPSS, etc.) in order to find interesting association rules.

Another usage of our code is in the Pipelined Data Mining Framework. The framework designed and developed by SZTAKI allows the handling and processing of very large datasets using a data stream approach. It combines an abstraction of data source, run-time modularity and configurability with keeping performance and resource management issues in hand. This enables a versatile platform for data mining and data analysis that allows an ordinary desktop or workstation-range computer to perform DM tasks and/or run queries on such large datasets that are much outside the scope of boxed DBMS.
or statistical systems. Our Apriori implementation was chosen to be embedded in the Pipeline.

We were pleased to learn that the data structures, implementation issues and the methods for speeding-up the search and reducing the memory requirement proposed by Christian Borgelt and the author of this thesis, become the basics issues regarding algorithm Apriori. Due to the efficiency of our trie-based approach and the scalability of this solution the trie become the de facto standard concerning the main data structure of Apriori: all the major and competitive systems have adopted this approach.

7 Acknowledgments

I would like to express my thanks to my supervisors: Lajos Rónyai, whose mathematical supervision and research guidance was very useful and to Tadeusz P. Dobrowiecki for helping me starting my academic career.

I am grateful to the Department of Computer Science and Information Theory at the Budapest University of Technology and Economics and the Web Search and Data Mining Group at MTA SZTAKI, the two pleasant and stimulating environments in which my PhD research was carried out. I wish to thank the head of the department, András Recski for his support.

Among my colleagues at the Informatics Laboratory of MTA SZTAKI, I am especially grateful to Balázs Rácz for a successful research period that we spent together. I am much in debt to Lars Schmidt-Thieme for his ideas and for sharing his experiences with me. Above all, I would like to thank him for the invitation to the University of Freiburg. The joint work in Freiburg gave the final and most important impulse to this thesis.

References


Chapter in a Book


Electronic Publication


Journal Publications (with peer review)


Journal Publications in Hungarian


Conference Proceedings Publications (with Peer Review)


Conference Proceedings Publications in Hungarian
