Mining Frequent Itemsets
A Perspective from Operations Research

Wim Pijls[*] Walter A. Kosters[†]

Econometric Institute Report EI 2008-24

October 20, 2008

Abstract

Many papers on frequent itemsets have been published. Besides some contests in this field were held. In the majority of the papers the focus is on speed. Ad hoc algorithms and datastructures were introduced. In this paper we put most of the algorithms in one framework, using classical Operations Research paradigms such as backtracking, depth-first and breadth-first search, and branch-and-bound. Moreover we present experimental results where the different algorithms are implemented under similar designs.
Keywords: Data mining, Operations Research, Frequent itemsets, Search

1 Introduction

Frequent itemset mining as a research area came into being in the nineties. The seminal paper appeared in 1994 [Agrawal and Srikant, 1994]. In the subsequent decade numerous papers were published. The basic problem in *frequent itemset mining* is, given a series of sets, to find all subsets that are contained in at least *minsup* of them; here *minsup* is a user-specified threshold. The problem of frequent itemset mining plays an important role in several data mining fields [Tan et al., 2006], such as association rules [Agrawal and Srikant, 1994], warehousing [Wu, 2006], correlations [Brin et al., 1998] and classification [Hu et al., 1999, Kosters et al., 1999, Pijls and Potharst, 2000]. The subject is also related to rough sets [Pawlak, 2000] and logical analysis of data [Boros et al., 2000]. Moreover, frequent itemsets have many application areas, amongst others customer relationship management, fraud detection, product assortment decisions [Brijs et al., 1999], episode mining [Mannila et al., 1995], functional dependency discovery [Huhtala et al., 1999], etc.
In the literature on frequent itemsets the algorithms are usually studied from a practical viewpoint. Almost any paper focuses on speed. To achieve an optimal running time, specific implementation tricks are applied. In the current paper, the theory of frequent itemsets is discussed from an algorithmic viewpoint. The algorithms in the majority of the papers

[*]Econometric Institute, Erasmus University Rotterdam, P.O.Box 1738, 3000 DR Rotterdam, The Netherlands, e-mail: pijls@few.eur.nl

[†]Leiden Institute of Advanced Computer Science, Universiteit Leiden, P.O. Box 9512, 2300 RA Leiden, The Netherlands, e-mail kosters@liacs.nl

use paradigms that are common in Operations Research, such as backtracking, branch-and-bound, depth-first search and breadth-first search. Nevertheless, the theory on frequent itemsets is usually not treated in terms of those paradigms. Complex dedicated datastructures are utilized. Consequently, the distinction between the algorithm and the datastructure under consideration is hardly ever made. Our goal is to achieve greater transparency, using the well-known concepts from Operations Research. Therefore, this paper makes a clear distinction between algorithms and datastructures. After a decade with an extensive variety of methods a theoretical survey is needed. In this paper, the well-known algorithms are put into one framework. Other surveys, however without a unifying view, can be found in [Bodon, 2006, Goethals, 2003]. Given the vast literature, mining frequent itemsets has become a topic as classical as sorting arrays or finding shortest paths in a network. It should take a prominent place in textbooks on "Algorithms and Datastructures".
For a present-day computer time and memory constraints are no significant issues anymore, with respect to many frequent itemset mining problems. Memory sizes exceeding 1 GB are standard nowadays. Running our algorithms on benchmark sets shows that even for sets with hundreds of megabytes, it takes only a few seconds to obtain the full collection of frequent item sets. Many papers introduced dedicated datastructures in order to obtain an optimal running time. Almost any paper claims a high performance in running time. We refrain from speed optimizations. Only for comparison, we present a small set of figures.

The algorithms most frequently discussed in textbooks on Datamining are APRIORI [Agrawal and Srikant, 1994, Agrawal et al., 1996], FP-GROWTH [Han et al., 2000, 2004] and ECLAT [Zaki, 2000]. From FP-GROWTH the methods of OPPORTUNEPROJECT [Liu et al., 2002] and TOPDOWN-FP [Wang et al., 2002] are derived. Next to ECLAT a variant called DECLAT [Zaki and Gouda, 2003] is known. All these algorithms are discussed here along with their derivatives. As said before, we do not strive for a minimal running time. Only a rough comparison between the best-known algorithms is made, in order to get a better understanding.

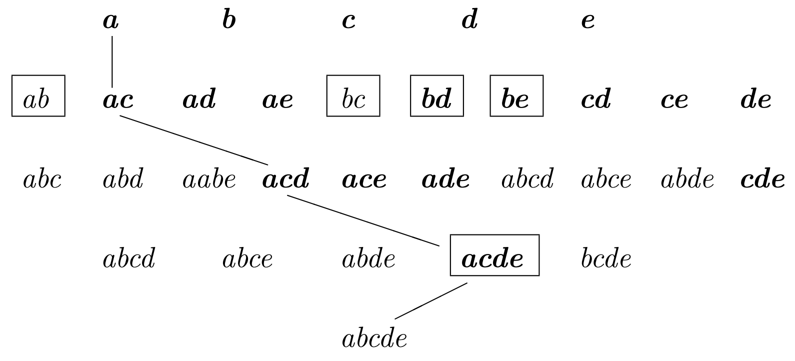
Outline This paper is organized as follows. Section 2 gives the preliminaries, including the datastructures used and some issues concerning complexity. Section 3 discusses a general *depth-first* framework for mining frequent itemsets. Multiple instances of this algorithm are presented in Section 4 and 5, utilizing *tries* and *tid-lists* (tid stands for "transaction identifier"), respectively. The depth-first instances are compared with respect to running time in Section 6. Section 7 treats the *breadth-first* algorithm, which appears to be much slower than its depth-first counterparts. In Section 8 *maximal frequent* itemsets are studied. Finally some concluding remarks are made.

2 Preliminaries

A transaction database $\mathcal{B}$ is an ordered series of *transactions* (or records), where each transaction is a set consisting of a finite number of *items*. The items will be denoted by small letters *a, b, c, d, ...*. Note, however, that i and s will always be used as variables. We say that a transaction T *supports* an itemset I, if all items of I are included in T. The *support* of an itemset I is defined as the number of transactions that support I. Similarly the support of a single item is defined, by looking at the corresponding singleton set. For a single item, we use the term *frequency* rather than *support*.
An itemset is *frequent* if its support is larger than or equal to the minimum support *minsup*,

a threshold value given beforehand. If an itemset I is frequent, it is also called a *pattern*. Consider the transaction database of Figure 1 (left), our running example in this paper. This

tid	transaction
1	*be*
2	*acde*
3	*acde*
4	*bcde*
5	*abcde*
6	*abd*



a **b** **c** **d** **e**
ab **ac** **ad** **ae** bc **bd** **be** **cd** **ce** **de**
abc abd aabe **acd** **ace** **ade** abcd abce abde **cde**
abcd abce abde **acde** bcde
abcde

Figure 1: Example database together with its lattice, with **frequent sets** and border sets (in boxes), and one example path.

database consists of 6 transactions with *transaction identifiers* (*tid's*) from 1 through 6. There are 5 items involved: a, b, c, d and e. Throughout this paper we assume that the alphabetical order of the items conforms to a non-decreasing frequency order. So the frequencies of a, b, c, d, e form a non-decreasing series. An itemset can be denoted by just concatenating its members, in some order. So, instead of the more formal $\{a, d, e\}$ we often put *ade*. Suppose we have *minsup* $= 3$. The collection of all itemsets can be represented by a lattice, see Figure 1 (right). The empty set $\emptyset$ could have been added on top. If one adds edges between sets in subsequent rows that are included in one another, one arrives at the so-called Hasse diagram of this partial order. The bold-faced sets are frequent, i.e., their support has a value ≥ 3. A full listing of the frequent sets is shown in the following table:

support 5:	*d, e*
support 4:	*a, ad, b, c, cd, cde, ce, de*
support 3:	*ac, acd, acde, ace, ade, ae, bd, be*

Table 1: The frequent itemsets of the database

A frequent itemset S is called *maximal*, if any superset of S is infrequent. Analogously, an infrequent itemset T is called *minimal* if any subset is frequent. In our example data set with *minsup* $= 3$ the sets *acde*, *bd* and *be* are maximal and *ab* and *bc* are minimal. The collection of maximal frequent and minimal infrequent sets is called the *border* of the lattice. In Figure 1 (right) the elements of the border are framed in a rectangular box. The elements of the border have the property that any subset is frequent and any superset is infrequent. Consider a path from a 1-itemset at the top to the full itemset *abcde* at the bottom, such that each itemset in this path is a subset of its successor. Such a path includes at most one itemset from the border. See Figure 1 (right) for an example.
The complements of the three maximal frequent sets are b, *ace* and *acd* respectively, defining a collection $\mathcal{D}$. In the theory on hypergraphs a *transversal* is a well-known concept. Given a collection $\mathcal{C}$ of sets, a transversal is defined as a set with at least one element in each set of $\mathcal{C}$. For the collection $\mathcal{D}$ of complements of the maximal frequent sets, the transversals of $\mathcal{D}$ are

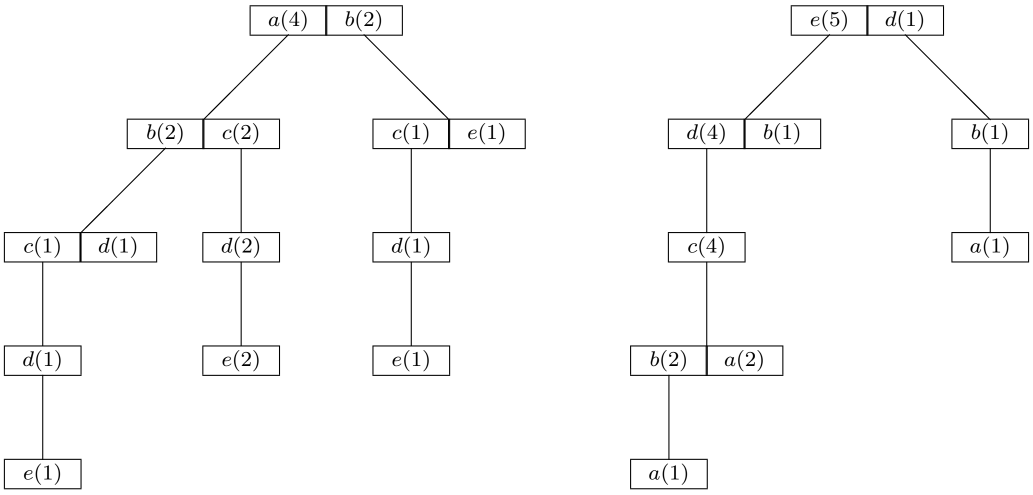


Figure 2: Two tries representing the example database.

the sets that are not contained in any frequent set. This implies that the transversals of $\mathcal{D}$ are exactly the infrequent sets. Consequently, frequent sets may also be found using algorithms from hypergraph theory, see [Gunopulos et al., 1997, Uno and Satoh, 2003].

Datastructures

An appropriate and compact datastructure for the database is the *trie* [Goodrich and Tamassia, 2005]. The trie was initially introduced to count the frequency of words in a text [Briandais, 1959, Fredkin, 1960]. Words or strings with a common prefix share a common path (starting from the root) in a trie. Such a trie can be easily constructed from an initially empty trie, by adding the strings under consideration one by one, following the path determined by their prefixes; counts indicating how often the corresponding prefix occurs, are updated during this traversal, and, if necessary, new nodes are created on the fly. As already mentioned above, transactions can be written as strings, e.g., the string *abd* denotes the transaction containing the items a, b and d. Those strings can be put into a trie. Figure 2 (left) displays the trie corresponding to the transaction database of Figure 1. The trie is made up of *cells* including an item denotation. The number between brackets indicates the *count* of the prefix under consideration. For instance, the count of prefix *abd* equals 2, meaning that 2 transactions start with *abd*. A set of adjacent cells is called a *block*. Any information in a data set necessary to find frequent item sets, is preserved in the trie.
When a transaction is written as a string, it is not necessary to place the items in alphabetical order. It is also possible to apply, e.g., the anti-alphabetical order, so *dba* instead of *abd*. Figure 2 (right) shows the trie corresponding to transactions written in anti-alphabetical order. For some algorithms this anti-alphabetical representation is preferable, as we will see in Section 4.1.
Although memories are large nowadays, the trie may not fit into the internal memory. In that

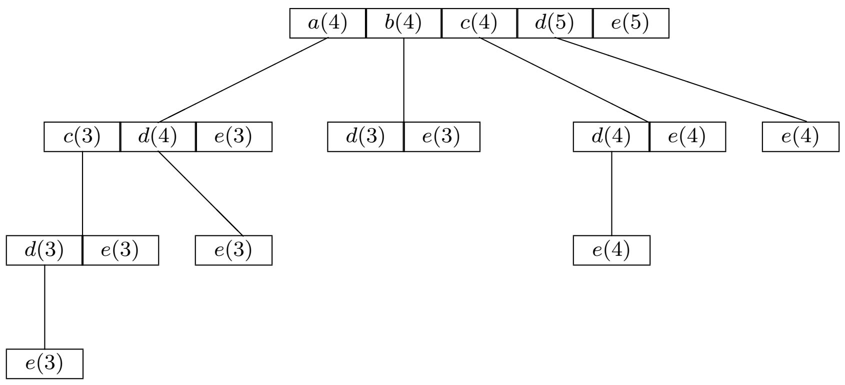


Figure 3: The trie representing the frequent itemsets of Table 1.

case the following approach is of use, as proposed in [Savarese et al., 1995]. The database is partitioned into k parts. Let n and n_i denote the number of transactions in the entire database and its i-th part, $1 \leq i \leq k$, respectively. Likewise, let *supp* and $supp_i$ denote the global and local support, the latter being the support for the i-th part. An itemset I can only be frequent globally if I is frequent locally in at least one part. More formally, $supp(I) \geq minsup$ implies $supp_i(I) \geq minsup \times n_i/n$ for at least one i. Combining the locally frequent sets gives the candidates for the globally frequent itemsets. A new pass through the database determines which of these candidates are globally frequent.

The collection of frequent itemsets consists of a huge number of strings. We also put this collection of strings into a trie. In our implementations the output of the algorithm is the trie of frequent itemsets. For the frequent sets of Table 1 this trie is shown in Figure 3. The number in each cell indicates the support of the itemset corresponding to the path from the root up to that cell.

Complexity issues

An important class of problems, introduced in [Valiant, 1979], is class $\#P$. Like the class *NP*, class $\#P$ has a subclass of $\#P$-complete problems. For many *NP*-complete decision problems, the associated counting problem has been proved to be $\#P$-complete [Papadimitriou, 1994]. Enumerating the solutions of a $\#P$-complete problem, is *NP*-hard, i.e., intractable in worst case [Garey and Johnson, 1979, Papadimitriou, 1994]. Consequently, this task requires an exhausting method such as backtracking, branch-and bound or dynamic programming.
In [Gunopulos et al., 2003] it was shown that counting the number of frequent items for a given minimal support is $\#P$-complete. This implies that we need an exhausting method to enumerate the frequent itemsets. In Sections 3 and 7 we discuss two algorithms. In Sections 4 and 5 we elaborate on particular datastructures useful for the algorithm of Section 3.

3 Finding patterns by depth-first search

As mentioned before, we need an enumeration method for finding frequent sets. The most rudimentary enumeration method is *backtracking* [Golomb and Baumert, 1965], actually an

implementation of depth-first search. In this section we discuss a general backtracking algorithm, which will act as a framework for the algorithms in the subsequent two sections, where some practical instances are discussed.
Let a transaction database $\mathcal{B}$ be given. The patterns in a database $\mathcal{B}$ can be found by backtracking using the following recursive procedure, being a first naive version:

```
(1)  naive-find-patterns(itemset $P$, itemset $I$) ::
(2)  for every $i \in I$ (in a randomly chosen order) do
(3)      $s \leftarrow$ support($i, \mathcal{B}$); // the support of $i$ in $\mathcal{B}$
(4)      if $s \geq$ minsup then
(5)          $P' \leftarrow P \cup \{i\}$ and place $P'$ in the output;
(6)          $I' \leftarrow \{ i' \mid i' \in I$ and $i \prec i' \}$;
(7)          naive-find-patterns($P', I'$);
```

the NaiveFindPatterns *algorithm*

The main call of this procedure is *naive-find-patterns*$(\emptyset, I)$, where I denotes the set of frequent single items occurring in $\mathcal{B}$. Notice that line (6) assumes an order $\prec$ for the items in I, which may differ from the one used in line (2). The algorithm NaiveFindPatterns may also be viewed as an instance of Rymon's set enumeration [Rymon, 1992]. Indeed, if $\mathcal{B}$ consists of one transaction made up by the itemset $I = \{i_1, i_2, \ldots, i_n\}$ and *minsup* $= 1$, this main call generates all $2^n - 1$ (non-empty) subsets of I. This is the straightforward method for generating all subsets of a given set.
In general, a recursive procedure applied to an input instance generates a so-called recursion tree, which displays the subsequent nested calls. When the main call of *naive-find-patterns* is applied to our example database of Section 2, the recursion tree looks like Figure 3, provided that the $\prec$ order conforms to the alphabetical order. Each block B of adjacent cells containing the items $i_1, i_2, \ldots, i_n$ corresponds to a call *naive-find-patterns*(P, I) with $I = \{i_1, i_2, \ldots, i_n\}$ and P consisting of the ancestor items of block B.
The above code allows an obvious improvement. In each next nested call a smaller part of the database is relevant to establish the support of i in $\mathcal{B}$ in line (3). We introduce the notion of a *conditional database*, also called a *projected database*. Given a database $\mathcal{B}$, we can define the conditional database for an itemset P and itemset I with $P \cap I = \emptyset$, in short *cond-db*$(\mathcal{B}, P, I)$. This object is defined as the collection $\mathcal{B}'$ of transactions that include P; furthermore, each transaction t of $\mathcal{B}'$ is restricted to the items from I, so t contains only items from I. We call P the *conditional itemset*. The following extended procedure exploits the concept of a conditional database:

```
(1)  $\textit{find-patterns}(\textit{database } \mathcal{B}, \textit{itemset } P, \textit{itemset } I)$ ::
(2)  **for** $\textit{every } i \in I$ $\textit{(in a randomly chosen order)}$ **do**
(3)      $s \leftarrow \textit{support}(i, \mathcal{B})$;
(4)      $P' \leftarrow P \cup \{i\}$ $\textit{and place}(P', s)$ $\textit{in the output}$;
(5)      $I' \leftarrow \{\ i' \mid i' \in I \textit{ and } i \prec i'\ \}$;
(6)      $\mathcal{B}' \leftarrow \textit{cond-db}(\mathcal{B}, \{i\}, I')$;
(7)      $\textit{determine support}(i', \mathcal{B}')$ $\textit{for each } i' \in I'$;
(8)      $\textit{remove infrequent items } i' \textit{ from } I' \textit{ and modify } \mathcal{B}' \textit{ accordingly}$;
(9)      **if** $I' \neq \emptyset$ **then**
(10)         $\textit{establish an order} \prec \textit{in } I'$;
(11)         $\textit{find-patterns}(\mathcal{B}', P', I')$;
```

the FindPatterns *algorithm*

The values *support*$(i, \mathcal{B})$, which are invoked in line (3), are supposed to be included in the representaton of $\mathcal{B}$. Line (7) computes these values for the new conditional database $\mathcal{B}'$. Our experiments have shown that the size of the conditional database reduces drastically in each nested call.
The main call is: *find-patterns*$(\mathcal{B}_0, \emptyset, I_0)$, where $\mathcal{B}_0$ denotes the original database and I_0 the set of frequent single items in $\mathcal{B}_0$. With this main call the pre- and postcondition of each nested call *find-patterns*$(\mathcal{B}, P, I)$ are the following.

pre: $\mathcal{B}$ is equal to *cond-db*$(\mathcal{B}_0, P, I)$ with P a frequent itemset in the original database $\mathcal{B}_0$; every item $i \in I$ is frequent in $\mathcal{B}$.

post: the output is enhanced with pairs (Q, s) for every frequent pattern Q in $\mathcal{B}_0$ such that $P \subsetneq Q$ (notice the strict inclusion).

The precondition clearly holds for the main call, if infrequent items are removed. To show that the precondition holds for any nested call, one must show that, assuming the precondition for the call in line (1), the precondition also holds for the recursive call in line (11). This is easily derived using the following property: the set of transactions in *cond-db*$(\mathcal{B}_0, P, I)$ supporting $\{i'\}$ is equal to the set of transactions in $\mathcal{B}_0$ supporting $P \cup \{i'\}$. Likewise, one must show that, assuming the postcondition for the recursive call in line (11), this condition also holds for the call in line (1).

For each recursive call an order $\prec$ has to be chosen in line (10). Making I' inherit the order of I is an obvious heuristic choice, which is also simple to implement. With this heuristic choice the order $\prec$ between the items is fixed in every nested call. The experiments to be discussed in Section 6 were conducted with the order $\prec$ always alphabetical, conforming to a non-decreasing initial frequency, as settled in Section 2. This implies that i is combined in line (5) with items i' that have an equal or higher initial frequency. Figure 3 is also based upon this order. We found that this order performs much better than the reverse order, i.e., $\prec$ corresponding to a non-increasing frequency.

4 Depth-first search using tries

When executing the backtracking algorithm of the previous section, we need a representation of the original and the conditional databases. In Section 2 the trie was presented as a convenient datastructure. This datastructure with some enhancements will also be utilized in this section. The creation of a conditional database $\mathcal{B}'$ out of a given database $\mathcal{B}$ in line (6) of the FindPatterns algorithm is a process that needs special attention. We elaborate on that process in this section.
In order to construct the conditional database *cond-db*$(\mathcal{B}, \{i\}, I)$, the occurrences of i in the trie representing $\mathcal{B}$ need to be traced. To facilitate this step, for each item i the cells including i are put into a linked list. So we have for each item a linked list, see Figure 4 for an illustration. The links are drawn as dashed lines. Each linked list belonging to one item has a head; its tail is visualized by a black bullet. These linked lists are referred to as *chains*.

4.1 Bottom-up traversal in the trie: FP-growth

In this subsection the conditional database is traversed in a bottom-up manner. Since we prefer to combine i with items i' of higher alphabetical order, it is assumed that the bottom-up order in the trie is also alphabetical. So the items i' with $i \prec i'$ are the items occurring in the ancestors of i. This is the case in the trie of Figure 4, an extension of Figure 2 (right).
To construct the conditional database for item i, we walk from every occurrence of i to the root. For this traversal each cell needs to have a pointer to its father. We now describe the

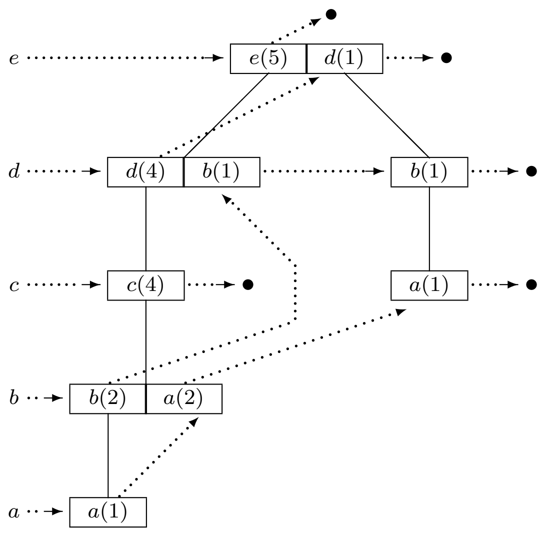


Figure 4: The right trie of Figure 2 enhanced with *chains*.

construction of the conditional database $\mathcal{B}'$ from $\mathcal{B}$ in line (6) in more detail:

1) Find in $\mathcal{B}$, using the chain for i, the cells including item i.
2) Consider from each cell found in 1) the path to the root. The count in that cell is also the count of the path.
3) Count the total frequency of each individual item using the counts of the paths in 2); remove infrequent items from the paths.

4) If desired, change the order in each path; the bottom-up order should comply with the $\prec$ order.
5) Compose a new trie from the paths.

In most practical situations step 4) is omitted, carrying over the order $\prec$ order from I to I'. The above steps are illustrated in Figure 5. In that case there are three cells including item a. For the conditional database with pattern a we consider the three paths starting from those cells. Each of the paths has a support which is given by the support of a in the bottom cell. The three paths (above the cells containing a) are shown on the left in Figure 5. Note that b is removed in step 3), since its frequency equals $2 < 3 = minsup$.

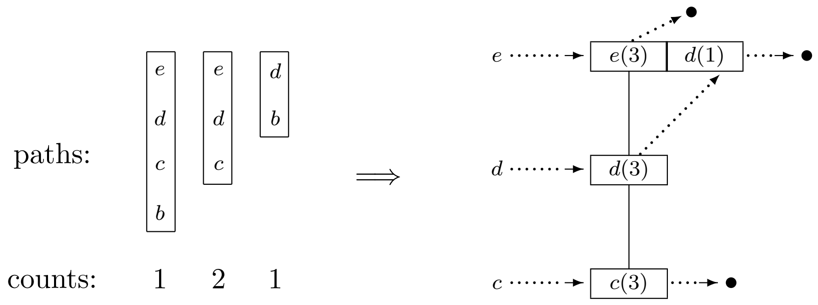


Figure 5: Constructing the conditional database and its trie for the a-chain, bottom-up.

The FINDPATTERNS algorithm which constructs the conditional database with the above bottom-up method, is known in the literature as the FP-GROWTH algorithm (FP stands for *Frequent Pattern*), published in [Han et al., 2000, 2004]. The trie with linked chains inside is called an *FP-tree* there. FP-GROWTH enters a separate phase with separate code, when the merge involves only one path with one count. Then every subset in the path is frequent. We need no separate code, since this situation is already captured by the code of Section 3; however, some speedup is achievable. In [Kosters and Pijls, 2003] we already identified FP-GROWTH as a backtracking instance.

4.2 Top-down traversal in the trie

Another method is the top-down approach contrasting the bottom-up manner of the previous subsection. Instead of visiting the ancestors of item i we visit the descendants. Two variants are distinguished. The first variant is known as TOPDOWN-FP [Wang et al., 2002] or OPPORTUNEPROJECT [Liu et al., 2002]. The second method is new and was not proposed in the literature before.
Section 3 mentioned that a more beneficial running time is achieved, if i is combined with items i' of higher alphabetical order. Therefore, when i is combined with descendents i', we need a trie where any top-down path shows the items alphabetically. Hence, this subsection, dealing with the top-down approach, is based upon Figure 2 (left).

The first variant is explained using Figure 6. In general, a conditional database is constructed as follows:

1) Using the chains, find the cells including item i.

2) Visits all descendants from the cells found in 1). This can be done using depth-first traversal starting from each cell in the chain.

3) Count the support for each item occurring in a descendant of i. Discard the infrequent items.

4) Put the cells containing frequent items, as far as these cells are descendants of i, into a linked list. So new chains are obtained.

Note that the new conditional database necessarily inherits the $\prec$ order from its generator. In Figure 6 (left) a representation of the conditional database *cond-db*$(\mathcal{B}_0, S, I)$ is displayed, where $\mathcal{B}_0$ is the example database of Section 2, $S = \{a\}$ and $I = \{c, d, e\}$. Since item b is infrequent in this subtrie, there is no related chain. Figure 6 (right) shows the conditional database with $S = \{a, c\}$ and $I = \{d, e\}$. All conditional databases are represented within one trie which is unaltered over time. Only the chains are changed in each nested call.

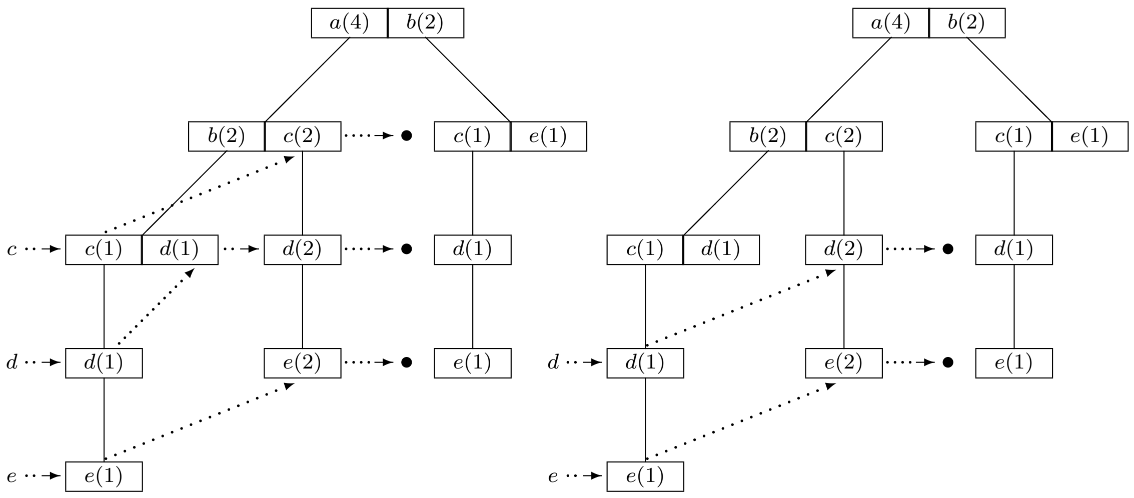


Figure 6: The conditional database with $\{a\}$ (left), resp. $\{a, c\}$ (right), as conditional set.

For this algorithm, the order in line (2) of *find-patterns* is not arbitrary. Suppose a database $\mathcal{B}$ is given as a trie with a set of chains. Let itemset I be given by $\{i_1, i_2, \ldots, i_n\}$, in accordance with the top-down order of the trie. If the conditional database $\mathcal{B}'$ for item $i = i_k$ is constructed, the chains belonging to the items $i_{k+1}, i_{k+2}, \ldots, i_n$ are changed. When the sub-call for item i_k has finished, the chains in $\mathcal{B}$ for the items $i_{k+1}, i_{k+2}, \ldots, i_n$ are out of kilter. Therefore, in order to be correct, *find-patterns* has to follow the order $\{i_n, i_{n-1}, \ldots, i_1\}$ in line (2), or put another way, the chains need to be treated in a bottom-up order.

To explain the second variant, consider Figure 7. The conditional tree is constructed analogously to the bottom-up method of the previous subsection. Suppose *cond-db*$(\mathcal{B}, \{b\}, \{c, d, e\})$ with $\mathcal{B} = \mathcal{B}_0$ is to be constructed. The subtries underneath the cells with item b are isolated, see Figure 7 (left) showing two subtries. Item c with support equal to 2 is found infrequent. Each cell including c is replaced with its child block. Due to this action we arrive at Figure

7 (middle). Merging these subtries gives Figure 7 (right). This resulting trie is provided with chains.

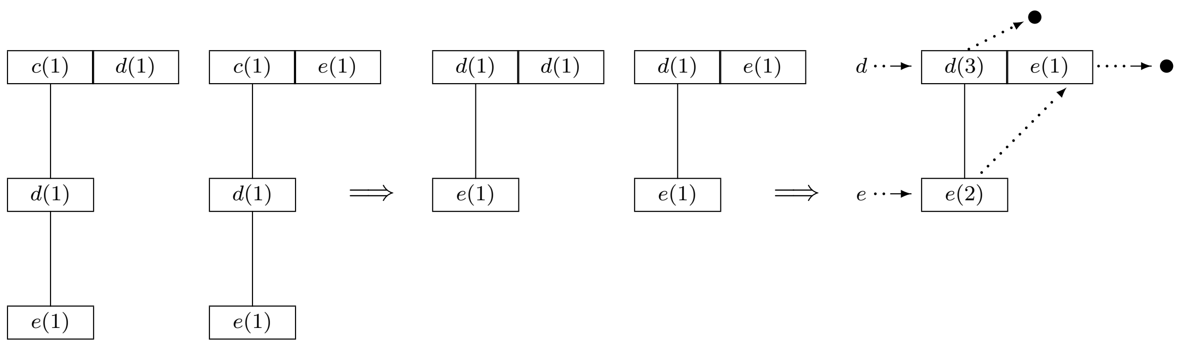


Figure 7: Constructing the conditional tree, top-down.

An advantage of this method is that, given two paths with a common subpath, the common part has to be traversed only once. This is different from FP-GROWTH, where each bottom-up path has to traversed fully, even if a subpath has been visited earlier as part of another bottom-up walk.

An extra feature of the top-down approach is that we can switch dynamically between the two variants. When the recursion depth is low, the second variant is applied, reducing the size of the trie and hence the traversal time considerably. For larger depths, time is saved when the construction of a new trie is left undone.

4.3 Other algorithms

Some other depth-first algorithms have been designed. H-MINE [Pei et al., 2001] resembles the above first top-down variant. Instead of projecting the data set into a trie, an array of transactions is built, where each transaction is an ordered list of item id's. This array is kept throughout the entire execution. The first item in a linear structure representing a transaction is called the *head* of the transaction. Similarly to the above first variant, building a new conditional database means inserting new links. Transactions with identical heads are linked. For so-called *sparse* databases (see Section 6) where the transactions have only few prefixes in common, this structure may be profitable. Moreover, this algorithm is scalable as it can easily be combined with the partitioning method discussed in Section 2.
Another algorithm related to the ones of this section is AFOPT [Liu et al., 2003]. This algorithm changes the database $\mathcal{B}$ in each iteration of the **for**-loop in *find-patterns*. The items i are treated according to their order in the root of the trie, from left to right. At the end of the iteration, item i is removed from the database $\mathcal{B}$. This amounts to removing the leftmost item in the root of the trie. See [Liu et al., 2004] for details. In [Borgelt, 2005] a similar algorithm is discussed.

5 Depth-first search using tid-lists: Eclat

In this section we discuss two other depth-first instances, viz. ECLAT [Zaki, 2000] and DECLAT [Zaki and Gouda, 2003], acronyms respectively for "Equivalence class transformation" and "difference-Eclat". The ECLAT algorithm uses a simple datastructure, reducing the trans-

formation between $\mathcal{B}$ and $\mathcal{B}'$ to a simple action. For each item i in a database $\mathcal{B}$, a so-called *tid-list* contains the id's of the transactions that include item i. A database $\mathcal{B}$ is represented by the set I along with the tid-lists $tid(i)$ for every $i \in I$.
Let the tid-list in the conditional database $\mathcal{B}' = cond\text{-}db(\mathcal{B}, \{i\}, I')$ be denoted by tid'. (One must realize that tid' depends on i.) Then $tid'(i')$ for an item i' in the conditional database $\mathcal{B}'$ is determined by the relation:

$$tid'(i') = tid(i) \cap tid(i'). \tag{1}$$

The following table shows the tid-lists of database $\mathcal{B}$ in Section 2 with conditional set $\{a\}$, so $\mathcal{B}' = cond\text{-}db(\mathcal{B}, \{a\}, \{b, c, d, e\})$:

$$\begin{array}{lcl@{\qquad}lcccl}
tid(a) & = & \{2,3,5,6\} & & & & & \\
tid(b) & = & \{1,4,5,6\} & tid'(b) & = & tid(a) \cap tid(b) & = & \{5,6\} \\
tid(c) & = & \{2,3,4,5\} & tid'(c) & = & tid(a) \cap tid(c) & = & \{2,3,5\} \\
tid(d) & = & \{2,3,4,5,6\} & tid'(d) & = & tid(a) \cap tid(d) & = & \{2,3,5,6\} \\
tid(e) & = & \{1,2,3,4,5\} & tid'(e) & = & tid(a) \cap tid(e) & = & \{2,3,5\}
\end{array}$$

When computing (1), we maintain a variable *feasible* for either list in the right-hand side. This variable is initialized to the size of the list. Whenever an element in a list turns out to fall outside the intersection, the variable *feasible* of that list decreases. As soon as at least one variable *feasible* is smaller than *minsup*, the resulting intersection is certainly infrequent and the intersecting process can be aborted.

Another method to represent a conditional database $\mathcal{B}$ is applied in dECLAT [Zaki and Gouda, 2003]. For each item $i \in I$, a list is available containing the id's of transactions in the conditional database $\mathcal{B}$ that do *not* include i. This list is called the complementary tid-list of i, denoted by $ctid(i)$. The new conditional database $\mathcal{B}'$ is constructed in line (6) using the following relation, holding for every $i' \in I'$:

$$ctid'(i') = ctid(i') \backslash ctid(i)\ , \tag{2}$$

where $ctid'$ denotes the complementary tid-list in $\mathcal{B}'$. Applying (2) with $i = a$ to our example database of Section 2 gives:

$$\begin{array}{lcl@{\qquad}lcccl}
ctid(a) & = & \{1,4\} & & & & & \\
ctid(b) & = & \{2,3\} & ctid'(b) & = & \{2,3\} \backslash \{1,4\} & = & \{2,3\} \\
ctid(c) & = & \{1,6\} & ctid'(c) & = & \{1,6\} \backslash \{1,4\} & = & \{6\} \\
ctid(d) & = & \{1\} & ctid'(d) & = & \{1\} \backslash \{1,4\} & = & \emptyset \\
ctid(e) & = & \{6\} & ctid'(e) & = & \{6\} \backslash \{1,4\} & = & \{6\}
\end{array}$$

The size of the conditional database $\mathcal{B}'$ for an item i is equal to $|\mathcal{B}'| = |\mathcal{B}| - |ctid(i)|$. In order to compute the support of item i' within $\mathcal{B}'$ in line (6) of *find-patterns*, the values $|\mathcal{B}'|$ and $|ctid(i')|$ suffice. This is due to the following relation:

$$|tid'(i')| = |\mathcal{B}'| - |ctid'(i')| \tag{3}$$

As a consequence of (3), the work needed to compute the sets $ctid(i')$ can be reduced. When computing $ctid'(i')$ using (2), we again maintain a variable *feasible* which is initialized to $|\mathcal{B}'|$. Whenever an element in $ctid'(i')$ is found, we decrease *feasible.* As soon as this variable drops under *minsup*, it is certain that $|tid(i')| < minsup$ and hence, the computation can be terminated.

In [Zaki and Gouda, 2003] dEclat was presented by means of relation (2), not regarding *ctid* as a complementary tid-set in the conditional database. Using our description a program can switch dynamically from Eclat to dEclat.

6 Evaluation

We have implemented the algorithms of the previous sections, in order to illustrate the principles, and also to give some feeling for the relative performance. The Java language has been chosen in order to obtain transparent program code. The programs are publicly available from [Pijls]. We did not strive for an optimal performance in space or time. Our goal was only to compare the algorithms under similar circumstances. The difficulties in benchmarking frequent patterns algorithms are pointed out in [Rácz et al., 2005]. The experiments were conducted on a 2.8 GHz Pentium IV machine running Windows XP.

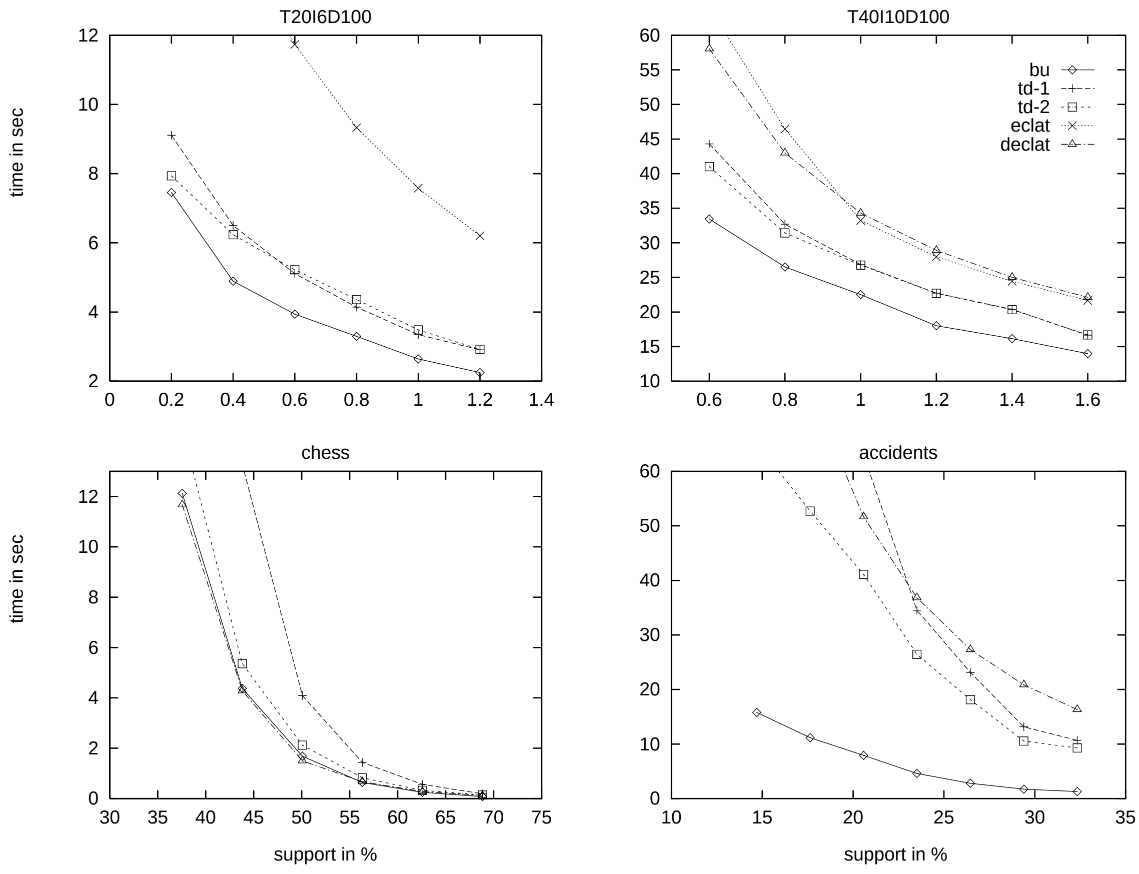


Figure 8: Experimental results for four databases.

We tested five algorithms:

- three algorithms from Section 4: Bottom-up (`bu`) and the first and second variant of Topdown (`td-1` and `td-2` respectively),
- two algorithms from Section 5: Eclat (`eclat`) and dEclat (`declat`).

The test sets are well-known among data mining researchers (see [UCI Machine Learning Repository]):

- `chess` (335 kB, 3.196 transactions),
- `accidents` (34,6 MB, 340.183 transactions),
- `T20I6D100K` (7,6 MB, $\approx$ 100.000 transactions),
- `T40I10D100K` (15,1 MB, 100.000 transactions).

The output of our programs is a trie, which is transformed to a file. The results are shown in Figure 6. One distinguishes between dense and sparse sets. One calls a data set *dense* if the relative support of the 1-items is relatively high, for instance over 10% [Pei et al., 2001]. Otherwise it is called *sparse*. The sets `T20I6D100` and `T40I10D100` have been generated by the program for synthetic data sets taken from [Quest Synthetic Data Generation Code]. They are sparse sets. The parameters in these sets are: D = number of transactions, T = average size of transactions and I = average size of maximal potentially large itemsets. The set `chess` was taken from [UCI Machine Learning Repository]. It has few but coherent records and is typically dense. The set `accidents` is a very large one. It was described in [Geurts et al., 2003, Geurts, 2003] and was made available for the purpose of `FIMI'03` [Goethals and Zaki, 2003]. Notice that some sets lack some graphs: `declat` is absent in `T20I6D100` and `eclat` is absent for `chess` and `accidents`. Those graphs fall outside the range of the picture.

For any set BOTTOM-UP is the winner, clear of its competitors, whereas TOPDOWN-2 is second in most cases. There is one surprising exception. The DECLAT algorithm turns out to be strong in dense data sets. Beside `chess` we tested DECLAT also with similar sets from [UCI Machine Learning Repository] and found similar results.
Many implementations are known for the algorithms of the Sections 4 and 5. An overview of the available software is found in [Software for associations discovery]. The two contests `FIMI'03` [Goethals and Zaki, 2003] and `FIMI'04` [Bayardo et al., 2004] were good opportunities to present implementations. Most of the programs submitted were based upon FP-GROWTH, ECLAT or APRIORI. The deviations from the instances published in the current paper mainly concern modifications to the datastructures. The winning program at `FIMI'03` was described in [Grahne and Zhu, 2003], a program based upon FP-GROWTH. The winner at `FIMI'04` was a highly optimized ECLAT based program [Uno et al., 2004]. A strong competitor was [Rácz, 2004], an FP-GROWTH-like program which uses an array-based implementation of the FP-tree.

7 Breadth-first search

Next to depth-first search discussed in Section 3 we have breadth-first search. In general, breadth-first search requires that the search space is kept in memory. As aforementioned, the output trie in Figure 3 is a convenient datastructure to store the full collection of frequent item sets. Breadth-first search builds this trie layer by layer. First, the layer at depth 1 is built, or put another way, the collection of 1-itemsets is composed. (A k-itemset is an itemset with k elements.) Next, the 2-nd layer or the collection of 2-itemsets is constructed, etc. In each iteration the following actions are performed. When n layers or the collection of n-itemsets have been built, the candidates for the $(n+1)$-th layer are constructed. From any two frequent n-itemsets with the form $S \cup \{i_1\}$ and $S \cup \{i_2\}$ respectively, where S is a frequent set of $n-1$ items and $i_1 \prec i_2$, a candidate of the form $S \cup \{i_1, i_2\}$ is composed. When no pair

of this type can be found, the execution stops. After the candidates at layer $n+1$ have been composed, their frequency is determined. To that end, for each transaction T with at least $n+1$ items, it is checked, whether T supports the new candidates. Visiting a candidate C amounts to traversing a path in the output trie from the root to depth $n+1$. The candidates are traced using backtracking. Obviously, as soon as an item outside T is encountered on a path to a candidate, the traversal through this path is aborted. To speed up visiting the candidates, each cell has a boolean variable indicating whether any candidate is among its descendants.
The algorithm is informally described as follows:

```
(1)   construct the root block of the output trie,
(2)       consisting of cells with the frequent 1-items;
(3)   continuing ← true;
(4)   while continuing = true do
(5)       construct a new layer of candidates;
(6)       if no new candidate can be composed then
(7)           continuing ← false;
(8)       else
(9)           for each transaction T
(10)              for each candidate C
(11)                  if T supports C then
(12)                      increase count of C by 1;
```

the BREADTH-FIRST *algorithm*

In the seminal paper on frequent item sets, this algorithm was presented under the name APRIORI [Agrawal and Srikant, 1994, Agrawal et al., 1996]. At that time, the database usually did not fit into the main memory and was partitioned when executing line (9). A faster version of APRIORI was presented in [Brin et al., 1997]. Another variant was proposed in [Kosters and Pijls, 2003]. Nowadays memories are large enough to contain the database in the representation of Figure 2. The paths in the trie with length $\geq n+1$ must be traversed instead of taking single transactions, as was the case in the original APRIORI. For such paths P the candidates in the output trie are visited. The above criterion to abort the walks to the candidates can be extended. If an item i in P matches an item i in a cell L of the output trie, the depth of i in P is compared with the number of remaining layers underneath L. Then it may be concluded that P cannot support any candidates descending from L.
We also implemented APRIORI with the database in a trie, see [Pijls]. Unfortunately this breadth-first algorithm performs very poorly compared with its depth-first counterparts.

8 Maximal frequent item sets

In Section 2 we defined *maximal frequent itemsets*. As the number of frequent itemsets may be very large, sometimes only the maximal frequent itemsets are asked. The maximal sets in the example database are *acde*, *bd* and *be*, as can be seen in Figure 1. Deciding if there is maximal frequent item set with at least n items for a given minimal support s is *NP*-complete [Yang, 2004]. Multiple algorithms for finding maximal sets have been designed, among others [Bayardo, 1998, Gouda and Zaki, 2005] and several algorithms presented in [Bayardo et al.,

2004, Goethals and Zaki, 2003].
When looking for maximal sets, one has to check for any itemset whether it is subsumed by others. This is the basic approach underlying almost any algorithm. Besides everyone has his own method to add optimizations.
We designed a procedure based upon the code of *find-patterns* in Section 3. The following observation is crucial for our procedure. Consider two subcalls in line (11) with pattern parameters $P'_1 = P \cup \{i_1\}$ and $P'_2 = P \cup \{i_2\}$ where $i_1 \prec i_2$. Any frequent itemset generated in the subcall with parameter P'_1 includes P'_1 and hence i_1. The subcall with parameter P'_2 will not generate any frequent itemset including i_1. Consequently, the itemsets generated in the subcall with parameter P'_1 cannot be subsumed by any set generated by the call with parameter P'_2. Suppose that the order in the execution of the **for**-loop is always in accordance with the $\prec$ order. (This order needs not to be the same for each nested call.) Then any frequent itemset can only be subsumed by an itemset that is generated before. This observation is utilized in our following procedure, an enhancement of *find-patterns* in Section 3.
We maintain a collection of maximal frequent itemsets generated. Since a trie is again a suitable datastructure, we call this collection the *max-trie*. If $I' = \emptyset$ in line (10) and hence no recursive call is executed in line (11), the itemset P' is a candidate for being maximal. It is checked, if P' is subsumed by any member of the max-trie. If not, P' is added to the max-trie. So the following piece of code is inserted after line (11):

```
(12)        else if no superset of $P'$ is in the max-trie then
(13)            add $P'$ to the max-trie;
```

For the check in line (12) we must compare P' with each string M in the max-trie. If a consistent order $\prec$ (for instance the alphabetical order) is used in all nested calls, the orders within P' and M are the same and the comparison is straightforward. The support of every prefix string in P' as well as in M is also known. Taking this support into account during the comparison process, we may sometimes decide soon that P' cannot be included in M. There is yet another optimization. If a recursive call of *find-patterns* in line (11) has generated a maximal set with size equal to $|P'| + |I'|$, the **for**-loop may be aborted. This might be viewed as an instance of the branch-and-bound paradigm.
It turns out that the above method for finding maximal sets is feasible and has a running time comparable to the methods for finding all frequent sets. Apparently, the action in lines (12) and (13) does not take much time.

From frequent itemsets one can derive association rules consisting of an antecedent and a consequent, related by a confidence. Apart from maximal sets so called *closed* frequent itemset are studied in the literature. These correspond to association rules with 100% confidence.

9 Concluding remarks

In this paper we studied the best-known methods for mining frequent itemsets. We considered algorithms for finding all frequent itemsets as well as maximal frequent sets. Our unifying framework allows a clear description of the many methods. Section 4.2 even introduced a novel algorithm. In our comparison of Section 6 the BOTTOM-UP method appeared to be the

fastest.
Mining frequent itemsets is indeed an interesting problem useful for illustrating the concepts which are common in the combinatorial optimization and Operations Research (OR) community. In the introduction we mentioned several fields related to OR, in which frequent itemsets play an important role. Therefore, the theory of frequent itemsets should be part of the toolbox of any worker in the OR area.

References

R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 487–499. Morgan Kaufmann, 1994.

R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

R. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD'98)*, pages 85–93, 1998.

R. Bayardo, B. Goethals, and M. J. Zaki, editors. *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, 2004. CEUR Workshop Proceedings [online].
`http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-126/`; (accessed April 23, 2008).

F. Bodon. A survey on frequent itemset mining. Technical report, Budapest University of Technology and Economics, 2006.

C. Borgelt. Keeping things simple: Finding frequent item sets by recursive elimination. In Goethals et al. [2005], pages 66–70, 2005.

E. Boros, P.L. Hammer, T. Ibaraki, A. Kogan, E. Mayoraz, and I. Muchnik. An Implementation of Logical Analysis of Data. *IEEE Transactions on Knowledge and Data Engineering*, 12:292–306, 2000.

R. de la Briandais. File searching using variable length keys. In *Proceedings Western Joint Computer Conference*, pages 295–298, 1959.

T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: A case study. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 1999)*, pages 254–260, 1999.

S. Brin, R. Motwani, J.D. Ullman, and S.Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the 1997 ACM SIGMOD International Conference (SIGMOD'97)*, pages 255–264, 1997.

S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: Generalizing association rules to correlations. *Data Mining and Knowledge Discovery*, 2:39–68, 1998.

E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1960.

M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* Freeman, 1979.

K. Geurts. Traffic accidents data set. [online], 2003.
`http://fimi.cs.helsinki.fi/data/accidents.pdf`; accessed April 23, 2008.

K. Geurts, G. Wets, T. Brijs, and K. Vanhoof. Profiling high frequency accident locations using association rules. In *Proceedings of the 82nd Annual Transportation Research Board, US*, 2003.

B. Goethals. Survey on frequent pattern mining. [online], 2003.
`http://www.cs.helsinki.fi/u/~goethals/publications/survey.ps`; accessed April 23, 2008.

B. Goethals and M.J. Zaki, editors. *Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, 2003. CEUR Workshop Proceedings [online].
`http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-90/`; (accessed April 23, 2008).

B. Goethals, S. Nijssen, and M.J. Zaki, editors. *Proceedings of the First International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*, 2005. ACM.

S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 1965.

M.T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java.* John Wiley and Sons, Fourth edition, 2005.

K. Gouda and M.J. Zaki. Genmax: An efficient algorithm for mining maximal frequent itemsets. *Data Mining and Knowledge Discovery*, 11:1–20, 2005.

G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Goethals and Zaki [2003], 2003.

D. Gunopulos, R. Khardon, H. Mannila, and H. Toivonen. Data mining, hypergraph transversals, and machine learning. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'97)*, pages 209–216, 1997.

D. Gunopulos, R. Khardon, H. Mannila, S. Sajula, H. Toivonen, and R.S. Sharm. Discovering all most specific sentences. *ACM Transactons on Database Systems*, 28:140–174, 2003.

J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 1–12, 2000.

J. Han, J Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8:53–87, 2004.

K. Hu, Y. Lu, L. Zhou, and C. Shi. Integrating classification and association rule mining: A concept lattice framework. In *Proceedings of the Seventh International Workshop on New Directions in Rough Sets, Data Mining and Granular Soft Computing*, pages 443–447. Springer, 1999. ISBN 3-540-66645-1.

Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42:100–111, 1999.

W. A. Kosters, E. Marchiori, and A. J. Oerlemans. Mining clusters with association rules. In *Proceedings of the Third International Symposium on Advances in Intelligent Data Analysis (IDA'99)*, pages 39–50. Springer Lecture Notes in Computer Science 1642, 1999. ISBN 3-540-66332-0.

W.A. Kosters and W. Pijls. Apriori: A depth-first implementation. In Goethals and Zaki [2003], 2003.

G. Liu, H. Lu, J.X. Yu, W. Wei, and X. Xiao. AFOPT: An efficient implementation of pattern growth approach. In Goethals and Zaki [2003], 2003.

G. Liu, H. Lu, W. Lou, Y. Xu, and J.X. Yu. Efficiently mining of frequent patterns using ascending frequency ordered prefix-tree. *Data Mining and Knowledge Discovery*, 9:249–274, 2004.

J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'02)*, pages 229–238, 2002. ISBN 1-58113-567-X.

H. Mannila, H. Toivonen, and A.I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, pages 210–215, 1995.

C. H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1994.

Z. Pawlak. *Rough Sets, Theoretical Aspects of Reasoning about Data.* Morgan Kaufman, 2000.

J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proceedings of the 2001 IEEE International Conference on Data Mining (ICDM 2001)*, pages 441–448, 2001.

W. Pijls. Homepage. [online].
`http://people.few.eur.nl/pijls/`; accessed April 23, 2008.

W. Pijls and R. Potharst. Classification and target group selection based upon frequent patterns. In *Proceedings Twelfth Belgium-Netherlands Artificial Intelligence Conference (BNAIC'00)*, pages 125–132, 2000.

Quest Synthetic Data Generation Code. IBM Almaden Research Center [online].
`www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html`; accessed April 23, 2008.

B. Rácz. nonordfp: An FP-growth variation without rebuilding the FP-tree. In Bayardo et al. [2004], 2004.

B. Rácz, F. Bodon, and L. Schmidt-Thieme. On benchmarking frequent itemset mining algorithms: From measurement to analysis. In Goethals et al. [2005], pages 36–45, 2005.

R Rymon. Search through systematic set enumeration. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 539–550, 1992.

A. Savarese, E. Omiecinsky, and S.B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB'95)*, pages 432–443, 1995.

Software for associations discovery. [online]. `http://www.kdnuggets.com/software/associations.html`; accessed April 23, 2008.

P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to data mining.* Addison Wesley, 2006.

UCI Machine Learning Repository. [online]. `http://archive.ics.uci.edu/ml/datasets.html`; accessed April 23, 2008.

T. Uno and K. Satoh. Detailed description of an algorithm for enumeration of maximal frequent sets with irredundant dualization. In Goethals and Zaki [2003], 2003.

T. Uno, T. Asai, Y. Uchida, and H. Arimura. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In Bayardo et al. [2004], 2004.

L.G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

K. Wang, L. Tang, J. Han, and J. Liu. Top-down FP-growth for association rule mining. In *Advances in Knowledge Discovery and Data Mining, Proceedings of the Sixth Pacific-Asia Conference (PAKDD 2002)*, pages 334–340. Springer Lecture Notes in Artificial Intelligence 2336, 2002. ISBN 978-3-540-43704-8.

C. Wu. Applying frequent itemset mining to identify a small itemset that satisfies a large percentage of orders in a warehouse. *Computers and Operations Research*, 33:3161–3170, 2006.

G. Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *Proceedings of the Tenth ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2004)*, pages 344–353, 2004.

M.J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, 2000.

M.J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2003)*, pages 326–335, 2003.