Depth-First Non-Derivable Itemset Mining

Toon Calders*
University of Antwerp, Belgium
`toon.calders@ua.ac.be`

Bart Goethals†
HIIT-BRU, University of Helsinki, Finland
`bart.goethals@cs.helsinki.fi`

Abstract

Mining frequent itemsets is one of the main problems in data mining. Much effort went into developing efficient and scalable algorithms for this problem. When the support threshold is set too low, however, or the data is highly correlated, the number of frequent itemsets can become too large, independently of the algorithm used. Therefore, it is often more interesting to mine a reduced collection of interesting itemsets, i.e., a condensed representation. Recently, in this context, the *non-derivable* itemsets were proposed as an important class of itemsets. An itemset is called derivable when its support is completely determined by the support of its subsets. As such, derivable itemsets represent redundant information and can be pruned from the collection of frequent itemsets. It was shown both theoretically and experimentally that the collection of non-derivable frequent itemsets is in general much smaller than the complete set of frequent itemsets. A breadth-first, Apriori-based algorithm, called NDI, to find all non-derivable itemsets was proposed. In this paper we present a depth-first algorithm, dfNDI, that is based on Eclat for mining the non-derivable itemsets. dfNDI is evaluated on real-life datasets, and experiments show that dfNDI outperforms NDI with an order of magnitude.

1 Introduction

Since its introduction in 1993 by Agrawal et al. [3], the frequent itemset mining problem has received a great deal of attention. Within the past decade, hundreds of research papers have been published presenting new algorithms or improvements on existing algorithms to solve this mining problem more efficiently.

The problem can be stated as follows. We are given a set of items $\mathcal{I}$, and an *itemset* $I \subseteq \mathcal{I}$ is some set of items. A *transaction* over $\mathcal{I}$ is a couple $T = (tid, I)$ where tid is the transaction identifier and I is an itemset. A transaction $T = (tid, I)$ is said to *support* an itemset $X \subseteq \mathcal{I}$, if $X \subseteq I$. A *transaction database* $\mathcal{D}$ over $\mathcal{I}$ is a set of transactions over $\mathcal{I}$. We omit $\mathcal{I}$ whenever it is clear from the context. The *cover* of an itemset X in $\mathcal{D}$ consists of the set of transaction identifiers of transactions in $\mathcal{D}$ that support X: $cover(X, \mathcal{D}) := \{tid \mid (tid, I) \in \mathcal{D}, X \subseteq I\}$. The

support of an itemset X in $\mathcal{D}$ is the number of transactions in the cover of X in $\mathcal{D}$: $support(X, \mathcal{D}) := |cover(X, \mathcal{D})|$. An itemset is called *frequent* in $\mathcal{D}$ if its support in $\mathcal{D}$ exceeds the minimal support threshold σ. $\mathcal{D}$ and σ are omitted when they are clear from the context. The goal is now to find all frequent itemsets, given a database and a minimal support threshold.

Recent studies on frequent itemset mining algorithms resulted in significant performance improvements: a first milestone was the introduction of the breadth-first Apriori-algorithm [4]. In the case that a slightly compressed form of the database fits into main memory, even more efficient, depth-first, algorithms such as Eclat [18, 23], and FP-growth [12] were developed.

However, independently of the chosen algorithm, if the minimal support threshold is set too low, or if the data is highly correlated, the number of frequent itemsets itself can be prohibitively large. No matter how efficient an algorithm is, if the number of frequent itemsets is too large, mining all of them becomes impossible.

To overcome this problem, recently several proposals have been made to construct a condensed representation [15] of the frequent itemsets, instead of mining all frequent itemsets. A condensed representation is a sub-collection of all frequent itemsets that still contains all information. The most well-known example of a condensed representation are the *closed sets* [5, 7, 16, 17, 20]. The closure $cl(I)$ of an itemset I is the largest superset of I such that $supp(cl(I)) = supp(I)$. A set I is *closed* if $cl(I) = I$. In the closed sets representation only the frequent closed sets are stored. This representation still contains all information of the frequent itemsets, because for every set I it holds that

$$supp(I) = \max\{supp(C) \mid I \subseteq C, cl(C) = C\} \ .$$

Another important class of itemsets in the context of condensed representations are the *non-derivable* itemsets [10]. An itemset is called *derivable* when its support is completely determined by the support of its subsets. As such, derivable itemsets represent redundant information and can be pruned from the collection of frequent itemsets. For an itemset, it can be checked whether or not it is derivable by computing bounds on the support. In [10], a method based on the inclusion-exclusion principle is used.

It was shown both theoretically and experimentally that the collection of non-derivable frequent itemsets is in general much more concise than the complete set of frequent itemsets. It was proven that for a given database $\mathcal{D}$, all sets of length more than $\log_2(|\mathcal{D}|) + 1$ are derivable. Hence, especially in databases with numerous items, but with few transactions, the number of non-derivable itemsets is guaranteed to be relatively small compared to the number of frequent itemsets. Many biological datasets, e.g., gene expression datasets, are typical examples of such databases. In experiments in [10], it was shown empirically that the number of non-derivable itemsets is in general orders of magnitudes smaller than the number of frequent itemsets. In most experiments, the number of non-derivable itemsets was even smaller than the number of closed sets.

In [10], a breadth-first, Apriori-based algorithm NDI to find all non-derivable itemsets was proposed. Due to the relatively small number of non-derivable itemsets, the NDI-algorithm almost always outperforms mining all frequent itemsets, independently of the algorithm [8]. When we, however, look at the time and space required by the NDI-algorithm as a function of its output-size, its performance is far below that of state-of-the-art frequent set mining algorithms. The low efficiency of NDI comes mainly from the fact that it is a breadth-first generate-and-test algorithm. All candidates of a certain level need to be processed simultaneously, and the support tests involve repeated scans over the complete database.

In contrast, in the case of mining all frequent itemsets, depth-first algorithms have been shown to perform much better and have far less costly candidate generation phases, and do not require scanning the complete database over and over again. Furthermore, item reordering techniques can be used to avoid the generation of too many candidates.

Unfortunately, depth-first algorithms essentially do not perform the so called Apriori-test, that is, test whether all of subsets of a generated candidate are known to be frequent, as most of them are simply not generated yet. Nevertheless, the supports of the subsets of a given set is exactly what is needed in order to determine whether an itemset is derivable or not. In this paper, we tackle this problem and present a depth-first algorithm, dfNDI, for mining non-derivable itemsets. The dfNDI-algorithm is based on Eclat and the diffset technique as introduced by Zaki et al. [18, 19]. As such, dfNDI combines the efficiency of depth-first itemset mining algorithms with the significantly lower number of non-derivable itemsets, resulting in an efficient algorithm for mining non-derivable itemsets.

The organization of the paper is as follows. In Section 2, the non-derivable itemsets and the level-wise NDI algorithm are revisited. In Section 3, we shortly describe the Eclat-algorithm on which our dfNDI-algorithm is based. Special attention is paid to item reordering techniques. Section 4 introduces the new algorithm dfNDI in depth, which is experimentally evaluated and compared to the level-wise NDI in Section 5.

2 Non-Derivable Itemsets

In this section we revisit the non-derivable itemsets introduced in [10]. In [10], rules were given to derive bounds on the support of an itemset I if the supports of all its strict subsets of I are known.

2.1 Deduction Rules Let a *generalized itemset* be a conjunction of items and negations of items. For example, $G = \{a, b, \overline{c}, d\}$ is a generalized itemset. A transaction (tid, I) *contains a general itemset* $G = X \cup \overline{Y}$ if $X \subseteq I$ and $I \cap Y = \emptyset$. The *support of a generalized itemset G in a database* $\mathcal{D}$ is the number of transactions of $\mathcal{D}$ that contain G.

We say that a general itemset $G = X \cup \overline{Y}$ is *based* on itemset I if $I = X \cup Y$. From the well known inclusion-exclusion principle [11], we know that for a given general itemset $G = X \cup \overline{Y}$ based on I,

$$supp(G) = \sum_{X \subseteq J \subseteq I} (-1)^{|J \setminus X|} supp(J) \ .$$

Hence, $supp(I)$ equals

$$(2.1) \quad \sum_{X \subseteq J \subset I} (-1)^{|I \setminus J| + 1} supp(J) + (-1)^{|Y|} supp(G)$$

Notice that depending on the sign of $(-1)^{|Y|} supp(G)$, the term

$$\delta_X(I) := \sum_{X \subseteq J \subset I} (-1)^{|I \setminus J|} supp(J)$$

is a lower ($|Y|$ even) or an upper ($|Y|$ odd) approximation for the support of I. In [10], this observation was used to compute lower and upper bounds on the support of an itemset I. For each set I, let l_I (u_I) denote the lower (upper) bound we can derive using the deduction rules. That is,

$$
\begin{aligned}
l_I &= \max\{\delta_X(I) \mid X \subseteq I, I \setminus X \text{ odd}\} \ , \\
u_I &= \min\{\delta_X(I) \mid X \subseteq I, I \setminus X \text{ even}\} \ .
\end{aligned}
$$

Since we need the supports of all strict subsets of I to compute the bounds, l_I and u_I clearly depend on the underlying database.

EXAMPLE 1. *Consider the following database $\mathcal{D}$:*

$$\mathcal{D} = $$

TID	Items		
1	a, b, c, d	6	a, b, e
2	a, b, c	7	a, c, e
3	a, b, d, e	8	a, d, e
4	c, e	9	b, c, e
5	b, d, e	10	b, d, e

Some deduction rules for abc are the following:

$$supp(abc) \geq supp(ab) + supp(ac) - supp(a) = 1$$
$$supp(abc) \leq supp(ab) + supp(ac) + supp(bc)$$
$$-supp(a) - supp(b) - supp(c)$$
$$+supp(\{\}) = 2$$

Hence, based on the supports of the subsets of abc, we can deduce that $supp(abc)$ is in $[1,2]$.

In this paper, we will use Equation (2.1) to compute the support of I, based on the supports of its subsets, and the support of the generalized itemset G.

EXAMPLE 2. *Some equalities for itemset abc:*
$$supp(abc) = supp(ab) - supp(ab\bar{c})$$
$$= (supp(ab) + supp(ac) + supp(bc)$$
$$-supp(a) - supp(b) - supp(c)$$
$$+supp(\{\})) - supp(\overline{abc}) \qquad \square$$

Notice incidentally that the complexity of the term $\delta_X(I)$ depends on the cardinality of $Y = I \setminus X$. The larger Y is, the more complex it will be to compute the support of I based on $X \cup \overline{Y}$. For example, for $abcd$,

$$\delta_{abc}(abcd) = abc$$
$$\delta_a(abcd) = abc + abd + acd - ab - ac - ad + a$$

In general, the number of terms in $\delta_X(I)$ is $2^{|I \setminus X|} - 1$. Therefore, whenever in the algorithm we have the choice between two generalized itemsets, we will always choose the set with the least negations, as this set will result in a more efficient calculation of the support of I.

2.2 Condensed Representation Based on the deduction rules, it is possible to generate a summary of the set of frequent itemsets. Indeed, if $l_I = u_I$, then $supp(I, \mathcal{D}) = l_I = u_I$, and hence, we do not need to store I in the representation. Such a set I, will be called a *Derivable Itemset* (DI), all other itemsets are called *Non-Derivable Itemsets* (NDIs). Based on this principle, in [10], the following condensed representation was introduced:

$$\text{NDI}(\mathcal{D}, \sigma) := \{I \mid supp(I, \mathcal{D}) \geq \sigma, l_I \neq u_I\}.$$

In the experiments presented in Section 5, it is shown that the collection of non-derivable itemsets is much more concise than the complete collection of frequent itemsets, and often even more concise than other concise representations. For a discussion on the relation between NDI and the other condensed representation we refer to [9].

2.3 The NDI Algorithm In [10], in a slightly different form, the following theorem was proven:

Algorithm 1 Eclat

Input: $\mathcal{D}, \sigma, I \subseteq \mathcal{I}$
Output: $\mathcal{F}[I](\mathcal{D}, \sigma)$
1: $\mathcal{F}[I] := \{\}$
2: **for all** $i \in \mathcal{I}$ occurring in $\mathcal{D}$ **do**
3:     $\mathcal{F}[I] := \mathcal{F}[I] \cup \{I \cup \{i\}\}$
4:     // Create $\mathcal{D}^i$
5:     $\mathcal{D}^i := \{\}$
6:     **for all** $j \in \mathcal{I}$ occurring in $\mathcal{D}$ such that $j > i$ **do**
7:         $C := cover(\{i\}) \cap cover(\{j\})$
8:         **if** $|C| \geq \sigma$ **then**
9:             $\mathcal{D}^i := \mathcal{D}^i \cup \{(j, C)\}$
10:         **end if**
11:     **end for**
12:     // Depth-first recursion
13:     Compute $\mathcal{F}[I \cup \{i\}](\mathcal{D}^i, \sigma)$
14:     $\mathcal{F}[I] := \mathcal{F}[I] \cup \mathcal{F}[I \cup \{i\}]$
15: **end for**

THEOREM 2.1. [10] **(Monotonicity)** *Let $I \subseteq J$ be itemsets. If $supp(I) = \delta_X(I)$, then, for all X' such that $X \subseteq X' \subseteq X \cup (J \setminus I)$, $supp(J) = \delta_{X'}(J)$.*
    Hence, if I is a derivable, then J is derivable as well.

Based on this theorem, a level-wise Apriori-like algorithm was given in [10]. In fact, the NDI-algorithm corresponds largely to constrained mining algorithms with non-derivability as an anti-monotone constraint. In the candidate generation phase of Apriori, additional to the monotonicity check, the lower and upper bounds on the candidate itemsets are computed. Such a check is possible, since in Apriori a set I can only be a candidate after all its strict subsets have been counted. The candidate itemsets that have an upper bound below the minimal support threshold are pruned, because they cannot be frequent. The itemsets having lower bound equal to the upper bound are pruned since they are derivable. Because of Theorem 2.1, we know that the supersets of a derivable itemset will be derivable as well, and hence, a derivable itemset can be pruned in the same way as an infrequent itemset. Furthermore, from Theorem 2.1, we can derive that if for a set I, $supp(I) = l_I$, or $supp(I) = u_I$, then all strict supersets of I are derivable. These properties lead straightforwardly to the level-wise algorithm NDI given in [10].

3 The Eclat Algorithm

In this section we describe the Eclat-algorithm, since our dfNDI-algorithm is based on it. Eclat was the first successful algorithm proposed to generate all frequent itemsets in a depth-first manner [18, 23]. Later, several other depth-first algorithms have been proposed [1, 2, 13].

    Given a transaction database $\mathcal{D}$ and a minimal support

threshold σ, denote the set of all frequent itemsets with the same prefix $I \subseteq \mathcal{I}$ by $\mathcal{F}[I](\mathcal{D}, \sigma)$. Eclat recursively generates for every item $i \in \mathcal{I}$ the set $\mathcal{F}[\{i\}](\mathcal{D}, \sigma)$. (Note that $\mathcal{F}[\{\}](\mathcal{D}, \sigma) = \bigcup_{i \in \mathcal{I}} \mathcal{F}[\{i\}](\mathcal{D}, \sigma)$ contains all frequent itemsets.)

For the sake of simplicity and presentation, we assume that all items that occur in the transaction database are frequent. In practice, all frequent items can be computed during an initial scan over the database, after which all infrequent items will be ignored.

In order to load the database into main memory, Eclat transforms this database into its *vertical format*. I.e., instead of explicitly listing all transactions, each item is stored together with its cover (also called *tidlist*). In this way, the support of an itemset X can be easily computed by simply intersecting the covers of any two subsets $Y, Z \subseteq X$, such that $Y \cup Z = X$.

The Eclat algorithm is given in Algorithm 1.

Note that a candidate itemset is represented by each set $I \cup \{i, j\}$ of which the support is computed at line 7 of the algorithm. Since the algorithm doesn't fully exploit the monotonicity property, but generates a candidate itemset based on only two of its subsets, the number of candidate itemsets that are generated is much larger as compared to a breadth-first approach such as apriori. As a comparison, Eclat essentially generates candidate itemsets using only the join step from Apriori, since the itemsets necessary for the prune step are not available.

EXAMPLE 3. *Let the minimal support σ be 2. We continue working on $\mathcal{D}$ as given in Example 1. As illustrated in Figure 1, Eclat starts with transforming the database into its vertical format. Then, recursively, the conditional databases are formed. The arcs indicate the recursion. The tree is traversed depth-first, from left to right. For example, the database $\mathcal{D}^{ad}$ is formed using $\mathcal{D}^a$, by intersection the tid-list of d with the tid-lists of all items that come after d in $\mathcal{D}^a$. The tid-lists with the item between brackets (e.g., the tid-list for d in $\mathcal{D}^c$) indicate lists that are computed, but that are not in the conditional database because the item is infrequent. At any time, only the parents of the conditional database being constructed are kept in memory. For example, when $\mathcal{D}^{ad}$ is constructed, the databases $\mathcal{D}^a$ and $\mathcal{D}$ are in memory; once a conditional database is no longer needed, it is removed from memory.*

A technique that is regularly used, is to reorder the items in support ascending order. In Eclat, such reordering can be performed at every recursion step between line 11 and line 12 in the algorithm.

The effect of such a reordering is threefold:

(1) The number of candidate itemsets that is generated is reduced. The generation step of Eclat is comparable to
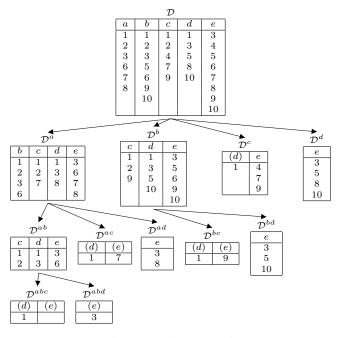

Figure 1: Eclat traversal.

the join-step of Apriori: a set $a_1 \ldots a_k$ is generated if both $a_1 \ldots a_{k-1}$ and $a_1 \ldots a_{k-2} a_k$ are frequent. By reordering, the generating sets tend to have lower support which results in less candidates.

(2) A second effect of the fact that the generating sets tend to have lower support is that their tid-lists are smaller.

(3) At a certain depth d, the covers of at most all k-itemsets with the same $k-1$-prefix are stored in main memory, with $k \leq d$. Because of the item reordering, this number is kept small.

Experimental evaluation in earlier work has shown that reordering the items results in significant performance gains.

EXAMPLE 4. *In Figure 2, we illustrate the effect of item reordering on the dataset given in Example 1, with the same support threshold $\sigma = 2$. As compared to Example 3, the tree is more balanced, the conditional databases are smaller, and there are 3 less candidates generated.*

3.1 Diffsets Recently, Zaki proposed a new approach to efficiently compute the support of an itemset using the vertical database layout [19]. Instead of storing the cover of a k-itemset I, the difference between the cover of I and the cover of the $k-1$-prefix of I is stored, denoted by the *diffset* of I. To compute the support of I, we simply need to subtract the size of the diffset from the support of its $k-1$-prefix. This support can be provided as a parameter within the recursive function calls of the algorithm. The diffset of

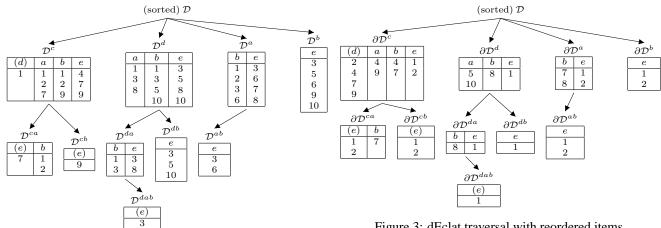Figure 2: Eclat traversal with reordered items.

Figure 3: dEclat traversal with reordered items.

an itemset $I \cup \{i, j\}$, given the two diffsets of its subsets $I \cup \{i\}$ and $I \cup \{j\}$, with $i < j$, is computed as follows:

$$diffset(I \cup \{i, j\}) := diffset(I \cup \{j\}) \setminus diffset(I \cup \{i\}).$$

This technique has experimentally shown to result in significant performance improvements of the algorithm, now designated as *dEclat* [19]. The original database is still stored in the original vertical database layout.

Notice incidentally that with item reordering,

$$supp(I \cup \{i\}) \leq supp(I \cup \{j\}) .$$

Hence, $diffset(I \cup \{i\})$ is larger than $diffset(I \cup \{j\})$. Thus, to form $diffset(I \cup \{i, j\})$, the largest diffset is subtracted from the smallest, resulting in smaller diffsets. This argument, together with the three effects of reordering pointed out before, makes that reordering is a very effective optimization.

EXAMPLE 5. *In Figure 3, we illustrate the dEclat algorithm with item reordering on the dataset given in Example 1, with the same support threshold $\sigma = 2$. The diffset of for example dab (entry b in the conditional database $\mathcal{D}^{da}$), is formed by subtracting the diffset of da from the diffset for db. Notice that the items are ordered ascending w.r.t. their support, and* not *w.r.t. the size of their diffset. The support of dab is computed as the support of da (3) minus the size of its diffset (1), which gives a support of 2.*

4 The dfNDI Algorithm

In this section we describe a depth-first algorithm for mining all frequent non-derivable itemsets. The dfNDI algorithm combines ideas behind the Eclat algorithm, the diffsets and the deduction of supports into one hybrid algorithm.

The construction of the dfNDI-algorithm is based on the following principles:

1. Just like Eclat, tidlists (and diffsets) will be used to compute the support of itemsets. Recursively, conditional databases will be formed. Hence, computing the support of an itemset will, unlike in the breadth-first version, *not* require a scan over the complete database.

2. There is one problem, however: to compute $\delta_X(I)$, the supports of all sets J such that $X \subseteq J \subset I$ must be known. Since Eclat is a depth-first algorithm, many of these supports are not available. This problem, however, can be solved by changing the order in which the search space is traversed. By changing the order, we can keep a depth-first approach, and still have the property that all subsets of a set I are handled before I itself.

3. Because we need the support of already found sets to compute bounds of their supersets, we will maintain the found frequent non-derivable itemsets in a specialized structure that allows fast lookup. Since the number of non-derivable itemsets is in general much lower than the number of frequent itemsets, the memory requirements for the specialized storage is not too bad; also, it is comparable to the amount of memory used in the ChARM algorithm to store all found frequent closed itemsets [22].

4. The deduction rules allow to extend the diffsets to tidlists of arbitrary generalized itemsets. That is, if we want to determine the support of a set I, we can use the cover of any of the generalized itemsets based on I. Then, based on the support of the generalized itemset $X \cup \overline{Y}$, and on $\delta_X(I)$, the support of I itself can be computed. This flexibility allows us to choose a generalized itemset that has minimal cover size.

In the rest of the section we concentrate on these four points. Then, we combine them and give the pseudo-code of dfNDI, which is illustrated with an example.

4.1 Order of Search Space Traversal We show next how we can guarantee in the Eclat-algorithm that for all sets I the support of its subsets is computed before I itself is handled. In Eclat, the conditional databases are computed in the following order (see Example 3):

$$
\begin{aligned}
\mathcal{D} \;\; \to \mathcal{D}^a \;\; &\to \mathcal{D}^{ab} \;\; \to \mathcal{D}^{abc} \;\; \to \mathcal{D}^{abcd} \\
&\qquad\qquad\quad\;\; \to \mathcal{D}^{abd} \\
&\to \mathcal{D}^{ac} \;\; \to \mathcal{D}^{acd} \\
&\to \mathcal{D}^{ad} \\
\to \mathcal{D}^b \;\; &\to \mathcal{D}^{bc} \;\; \to \mathcal{D}^{bcd} \\
&\to \mathcal{D}^{bd} \\
\to \mathcal{D}^c \;\; &\to \mathcal{D}^{cd} \\
\to \mathcal{D}^d
\end{aligned}
$$

Since the support of a set $a_1 \ldots a_n$ is computed as the cardinality of the tidlist of a_n in the conditional database $\mathcal{D}^{a_1 \cdots a_{n-1}}$, the supports of the itemsets are compute in the following order:

$$
\{a, b, c, d, e\}, \{ab, ac, ad, ae\}, \{abc, abd, abe\},
$$
$$
\{abcd, abce\}, \{abcde\}, \{abde\}, \{acd, ace\},
$$
$$
\{acde\}, \{ade\}, \{bc, bd, be\}, \{bcd, bce\}, \{bcde\},
$$
$$
\{bde\}, \{cd, ce\}, \{cde\}, \{de\}
$$

Hence, for example, when the support of $abcd$ is computed, the supports of bc, bd, cd, acd, and bcd are not counted yet. Therefore, when the search space is explored in this way, all rules for $abcd$ that use one of these five sets cannot be computed.

An important observation now is that for every set I, all subsets either occur *on the recursion path* to I, or *after* I. For example, the support of $abcd$ is computed in $\mathcal{D}^{abc}$. The supports of the subsets of $abcd$ are either computed on the recursive path: a, b, c, d in $\mathcal{D}$, ab, ac, ad in $\mathcal{D}^a$, abc, abd in $\mathcal{D}^{ab}$, or after $abcd$: acd in $\mathcal{D}^{ac}$, which is constructed after the recursion below $\mathcal{D}^{ab}$, bc, bd in $\mathcal{D}^b$, which comes after the recursion below $\mathcal{D}^a$ has ended, and cd in $\mathcal{D}^c$, which comes after the recursion below $\mathcal{D}^b$ has ended. We can view the recursive structure between the conditional databases in Eclat as a tree, in which the children are ordered lexicographically, as is illustrated in Figure 4. This tree is by Eclat traversed depth-first, and from left-to-right; that is: in pre-order. Let the node associated with an itemset I be the node of the conditional database in which the support of I is computed. The observation can now be restated as follows: the nodes of the subsets of I are either on the path from $\mathcal{D}$ to the node of I, or are in a branch that comes after I, *never* in a branch that comes *before* the branch of I.

Hence, we can change the order as follows: the same tree is traversed, still depth-first, but, from right to left. We will call this order the *reverse pre-order*. The numbers in the nodes of the tree in Figure 4 indicate the reverse pre-order. In this way, the path from $\mathcal{D}$ to the node of a set I remains
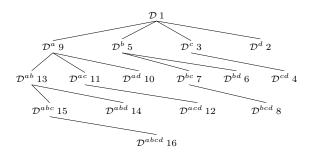

Figure 4: Recursive structure between the conditional databases

the same, but all branches that come after the branch of I, are now handled *before* the branch of I. As such, in reverse pre-order, all subsets of I are handled before I itself.
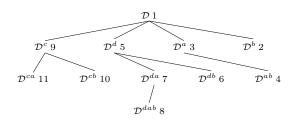
Besides enabling a depth-first algorithm for mining non-derivable itemsets, the reverse pre-order has other applications as well: a similar technique can be used when mining downwards closed collections of itemsets in a depth-first manner while preserving full pruning capabilities. It must be noted however that this ability comes the cost to store all found itemsets in the collection. Hence, the usefulness of this technique can be compromised when the downward closed collection becomes too large. For such large collections it is probably better to sacrifice part of the pruning in order to loose the store of generated itemsets. For small collections though, the reverse pre-order is very useful. Many condensed representations, such as the non-derivable itemsets, are typical examples of relatively small downward closed collections of itemsets.

Notice also that the reverse pre-order has no repercussions on performance or memory usage; the same covers are computed, and the same covers are in memory simultaneously; only the *order* in which they are generated differs. However, now we are guaranteed that *all* subsets of I are handled before I itself.

Another very important remark is that this guarantee remains even when at every step in the recursion the items in the conditional databases are reordered according to their support. Hence, we can still apply the important item reordering techniques.

EXAMPLE 6. *Consider the tree in Example 4 that was constructed with item reordering. The numbers in the nodes indicate the reverse pre-order.*

The reason that the same relation between a set and its subsets still applies can be seen as follows: the itemset $abde$ is counted in the conditional database $\mathcal{D}^{dab}$ as the cardinality of the tidlist of e. Hence, the order in which the items of $abde$ were selected is: $dabe$. That is: (a) in the database $\mathcal{D}$, d came before a, b and e; (b) in $\mathcal{D}^d$, a came before b and e; and (c) in $\mathcal{D}^{da}$, b came before e. Since the

Input: $\mathcal{D}, \sigma$
Output: $\mathcal{D}^l$
1: **for all** k occurring in $\mathcal{D}$ after l **do**
2:    // k is either j or $\bar{j}$
3:    $C[k] := cover(\{l\}) \cap cover(\{k\})$
4:    $C[\bar{k}] := cover(\{i\}) \setminus cover(\{k\})$
5:    **if** $\{i, j\}$ is σ-frequent **then**
6:      **if** $|C[j]| \leq |C[\bar{j}]|$ **then**
7:        $\mathcal{D}^{\{i\}} := \mathcal{D}^{\{i\}} \cup \{(j, C[j])\}$
8:      **else**
9:        $\mathcal{D}^{\{i\}} := \mathcal{D}^{\{i\}} \cup \{(\bar{j}, C[\bar{j}])\}$
10:      **end if**
11:    **end if**
12: **end for**

Figure 5: Recursive construction of $\mathcal{D}^l$ in dfNDI

items in a conditional database are processed backwards in the recursion, therefore, (a) a, b, d, e are in $\mathcal{D}$, and ab, ae, be, abe are handled before $\mathcal{D}^d$ is constructed, (b) ad, bd and de are in $\mathcal{D}^d$, and bde handled before $\mathcal{D}^{da}$ is constructed, and (c) ade and abd are in $\mathcal{D}^{da}$.

In general, let I be an itemset and the elements of I are chosen in the order $i_1 \ldots i_n$. Let J be a strict subset of I. If J corresponds to a prefix of $i_1 \ldots i_n$, then J is computed on the recursive path to $\mathcal{D}^{i_1 \cdots i_{n-1}}$. Otherwise, let i_j be the first item of $i_1 \ldots i_n$ that is not in J. Then, in the conditional database $\mathcal{D}^{i_1 \cdots i_{j-1}}$, all items in $J \setminus \{i_1, \ldots, i_{j-1}\}$ come strictly after i_j, and hence, the node for J is on the right of the node for I and is thus visited before I in the reverse pre-order.

4.2 Storing the Found NDIs The frequent non-derivable itemsets are stored together with their supports in a structure that allows for fast look-up of the supports. The itemsets are stored in such a way that fast lookup of the supports is possible. In our implementations, an itemset trie was used for this purpose.

In the extreme case that the number of non-derivable itemsets becomes too large to be maintained in main memory, a condensed representation [15] can be used as well. For example, only the closed itemsets [16] could be stored. Finally, remark that it is not uncommon that an algorithm needs to maintain itemsets in main memory. Most closed itemset mining algorithms need to store the found frequent closed sets in main memory (e.g. Charm [21]).

4.3 Generalizing the Diffsets In Eclat and dEclat, the conditional database $\mathcal{D}^{I \cup \{i\}}$ is generated from the database $\mathcal{D}^I$ if this database contains item i. All items $j > i$ that are in $\mathcal{D}^I$ will be in the conditional database $\mathcal{D}^{I \cup \{i\}}$. In Eclat, the tidlist of j in $\mathcal{D}^{I \cup \{i\}}$ is computed by intersecting the tidlists of i and j in $\mathcal{D}^I$. In dEclat, not the tidlists are maintained in the conditional databases, but the diffsets. The diffset of j in $\mathcal{D}^{I \cup \{i\}}$ is computed by subtracting the diffset of i in $\mathcal{D}^I$ from the diffset of j in $\mathcal{D}^I$. Hence, with the generalized itemsets notation, in Eclat the conditional database for $\mathcal{D}^{I \cup \{i\}}$ contains for all items $j > i$, $(j, cover(I \cup \{i, j\}))$. dEclat maintains for all items $j > i$, $(j, cover(I \cup \{i, \bar{j}\}))$ in the conditional database $\mathcal{D}^{I \cup \{i\}}$.

In dfNDI, we will use a similar procedure. First of all, the diffsets are extended to covers of arbitrary generalized itemsets. That is, not only covers of the type $cover(I \cup \{i, j\})$ and $cover(I \cup \{i, \bar{j}\})$ will be considered, but also $cover(X \cup \overline{Y})$ for any generalized itemset $X \cup \overline{Y}$ based on $I \cup \{i, j\}$. Secondly, the choice for which type of cover is not static in dfNDI. In contrast, in Eclat *always* $cover(I \cup \{i, j\})$, and in dEclat, *always* $cover(I \cup \{i, \bar{j}\})$ is used. In dfNDI, this choice will be postponed to run-time. At run-time both covers are computed, and the one with minimal size is chosen. In this way it is guaranteed that the size of the covers at least *halves* from $\mathcal{D}$ to $\mathcal{D}^i$. The calculation of both covers can be done with minimal overhead in the same iteration. When iterating over the cover of $\{i\}$, the set of all tid's that are not in $cover(\{i\}) \cap cover(\{j\})$ is exactly $cover(\{i\}) \setminus cover(\{j\})$.

Let $\mathcal{D}$ be a database that contains tidlists of both items and of negations of items. Let l be the list associated with item i. That is, l is either i or $\bar{i}$. The procedure used in dfNDI to construct the conditional database $\mathcal{D}^l$ from $\mathcal{D}$ is given in Figure 5. The procedure is applied recursively. Suppose that we have recursively constructed database $\mathcal{D}^G$, with $G = X \cup \overline{Y}$ a generalized itemset. For every item i (resp. negated item $\bar{i}$) in $\mathcal{D}^G$, the cardinality of the tidlist is in fact the support of the generalized itemset $X \cup \{i\} \cup \overline{Y}$ (resp. $X \cup \overline{Y} \cup \{i\}$) The support test in line 5 is then performed using the deduction rules as explained in Section 2. For example, suppose $l = i$ and $k = \bar{j}$. Let $J = X \cup Y \cup \{i, j\}$. The value $\delta_{X \cup \{i\}}(J)$ is computed (using the stored supports) and from this value and the size of the cover of $X \cup \{i, j\} \cup \overline{Y}$, the support of J can be found:

$$supp(J) = \delta_{X \cup \{i\}}(J) + (-1)^{|Y|} supp(X \cup \{i, j\} \cup \overline{Y})$$

Notice also that if $C[j]$ (resp. $C[\bar{j}]$) is empty, $supp(G \cup \{i, j\}) = 0$ (resp. $supp(G \cup \{i, \bar{j}\}) = 0$). From Theorem 2.1, we can derive that in that situation, every superset of $X \cup Y \cup \{i, j\})$ is a derivable itemset.

EXAMPLE 7. *Consider the following database $\mathcal{D}$, and the conditional databases $\mathcal{D}^a$ and $\mathcal{D}^{a\bar{b}}$:*

$$\mathcal{D}$$

a	b	c	d
1	1	2	1
2	2	4	2
3	3	5	3
5	7	6	9
8		7	

$\rightarrow$

$$\mathcal{D}^a$$

$\bar{b}$	c	$\bar{d}$
5	2	5
8	5	8

$\rightarrow$

$$\mathcal{D}^{a\bar{b}}$$

c	d
5	

$\mathcal{D}^a$ contains $\bar{b}$ because $cover(a) \setminus cover(b)$ is smaller than $cover(a) \cap cover(b)$. $\mathcal{D}^{a\bar{b}}$ contains d since $cover(\bar{b}) \setminus cover(\bar{d})$ is smaller than $cover(\bar{b}) \cap cover(\bar{d})$ in $\mathcal{D}^a$.

The support of abc is counted via the cover of c in $\mathcal{D}^{a\bar{b}}$. Since this cover contains one element, $supp(a\bar{b}c) = 1$, and hence,

$$supp(abc) = supp(ac) - supp(a\bar{b}c) = 2 - 1 = 1.$$

The support of ab is found in the trie that maintains the found frequent non-derivable itemsets. Notice that the cover of d in $\mathcal{D}^{a\bar{b}}$ is empty. Therefore, $supp(a\bar{b}d) = 0$, and thus, every superset of abd is derivable.

4.4 The Full dfNDI Algorithm The combination of the techniques discussed in this section give the dfNDI-algorithm described in Algorithm 2. In the recursion, the generalized itemset G that corresponds to the items or negated items that were chosen up to that point, must be passed on. Hence, the set of all frequent NDIs is given by $\text{dfNDI}(\mathcal{D}, \sigma, \{\})$. We assume that $\mathcal{D}$ is given in vertical format, and that all items in $\mathcal{D}$ are frequent.

As already mentioned before, reordering the items in the different conditional databases does not fundamentally change the algorithm. With reordering, we are still guaranteed that all subsets of an itemset I are handled before I itself. Therefore, in the experiments, we will use item reordering techniques to speed up the computation. There are different interesting orderings possible:

1. Ascending w.r.t. support: the advantages of this ordering are that the number of generated candidates is reduced, and that the tree is more balanced. Since for every candidate, lower and upper bounds on the support are computed, generating less candidates can be very interesting.

2. Ascending w.r.t. size of the cover: with this ordering, the size of the covers will in general be smaller, since the covers of the first items in the order are used more often than the covers of the items later in the ordering.

Depending on the application, either the first or the last ordering will be better. Which one is the best probably

---

Algorithm 2 dfNDI

Input: $\mathcal{D}, \sigma, G = X \cup \overline{Y}$
Output: $\text{dfNDI}(\mathcal{D}, \sigma, G)$

1: $\text{dfNDI} := \{\}$
2: **for all** l that occur in $\mathcal{D}$, ordered descending **do**
3:    // $l = i$ or $l = \bar{i}$, for an item i
4:    $\text{dfNDI} := \text{dfNDI} \cup \{(X \cup \overline{Y} \cup \{i\})\}$
5:    // Create $\mathcal{D}^l$
6:    $\mathcal{D}^l := \{\}$
7:    **for all** k occurring in $\mathcal{D}$ after l **do**
8:      // $k = j$ or $k = \bar{j}$, for an item j
9:      // Let $J = X \cup \overline{Y} \cup \{i, j\}$
10:     Compute bounds $[l, u]$ on support of J;
11:     **if** $l \neq u$ and $u \geq \sigma$ **then**
12:       // J in an NDI
13:       Store J in the separate trie
14:       // Compute the support of J
15:       $C[k] := cover(\{l\}) \cap cover(\{k\})$
16:       $C[\bar{k}] := cover(\{l\}) \setminus cover(\{k\})$
17:       // $|C[j]|$ is $supp(X \cup \overline{Y} \cup \{l, j\})$
18:       Compute $supp(J)$ based on the support of its subsets and on $|C[j]|$
19:       **if** $supp(J) \geq \sigma$ and $supp(J) \neq l$ and $supp(J) \neq u$ **then**
20:         **if** $|C[j]| \leq |C[\bar{j}]|$ **then**
21:           $\mathcal{D}^l := \mathcal{D}^l \cup \{(j, C[j])\}$
22:         **else**
23:           $\mathcal{D}^l := \mathcal{D}^l \cup \{(\bar{j}, C[\bar{j}])\}$
24:         **end if**
25:       **end if**
26:     **end if**
27:   **end for**
28:   // Depth-first recursion
29:   Compute $\text{dfNDI}(\mathcal{D}^l, \sigma, G \cup \{l\})$
30:   $\text{dfNDI} := \text{dfNDI} \cup \text{dfNDI}(\mathcal{D}^l, \sigma, G \cup \{l\})$
31: **end for**

depends on the selectivity of non-derivability versus the selectivity of frequency. If most sets are pruned due to non-derivability, the second ordering will be more interesting than the first. If frequency is the main pruner, the first order is more interesting.

There are many different variants possible of the dfNDI algorithm. Depending on the situation, one variant may be better than the other.
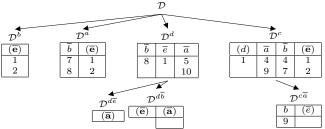
1. Ordering the items in the conditional databases, as described above.

2. Since computing the bounds can be very costly, we want to avoid this work as much as possible. Hence, in situations where frequency is highly selective, it is better to switch the support test and the non-derivability test. That is, lines 10 and 11 in Algorithm 2 are moved to after the if-test on line 19. Evidently, the test $supp(J) \neq l, u$ is removed from the if-test on line 19 to after the calculation of the bounds. Hence, only bounds are computed on itemsets that are frequent.

3. Another possibility to reduce the amount of work on the calculation of the bounds is not to compute all bounds, but only a limited number. A similar approach already turned out to be quite useful in the context of the breadth-first NDI-algorithm [10]. There, the depth of a rule based on $X \cup \overline{Y}$ was defined as $|Y|$. The lower the depth of a rule, the less complex. Instead of computing all rules, only rules up to a limited depth can be used. In practice it turned out that most of the time using rules up to depth 3 does not affect the precision of the rules. Put elsewhere, most work is done by the rules of limited depth.

EXAMPLE 8. *We illustrate the dfNDI algorithm on the database of Example 3. Every conditional database is ordered ascending w.r.t. the size of the cover. The tid-lists with the item between brackets indicate lists that are computed, but that are not in the conditional database, because the itemset I associated with the item is either (a) infrequent (e.g. d in $\mathcal{D}^c$)), or (b) $supp(I) = l_I$ or $supp(I) = u_I$. In case (b), I is a frequent NDI, but all supersets of I are derivable. The items in case (b) are indicated in bold.*

We start with the database $\mathcal{D}$. Because of the reverse pre-order that is used, first the database $\mathcal{D}^b$ is constructed. Only item e comes after b in $\mathcal{D}$. The lower and upper bounds on be are computed. $l_{be} = supp(b) + supp(e) - supp(\emptyset) = 5$, $u_{be} = supp(b) = 7$. Hence, be is not derivable. Both $cover(b) \cap cover(e)$ and $cover(b) \setminus cover(e)$ are constructed. $|cover(b) \cap cover(e)| = 5$, and hence, $supp(be) = 5$. Since $supp(be) = l_{be}$, all supersets of be must be derivable.

Next $\mathcal{D}^a$ is constructed. For ab the following lower and upper bound is computed: $l_{ab} = 3$, $u_{ab} = 6$. The two covers $cover(\{a\}) \cap cover(\{b\})$ and $cover(\{a\}) \setminus cover(\{b\})$ are

computed. From the size of these covers, the support of ab is computed: $supp(ab) = 4$. Because the support differs from the lower and the upper bound, b will be in $\mathcal{D}^a$. Since $cover(\{a\}) \setminus cover(\{b\})$ was the smallest of the two covers, $(\overline{b}, cover(\{a\}) \setminus cover(\{b\}))$ is added. Then the item e is handled. The bounds for ae are: $l_{ae} = 4$, $u_{ae} = 6$. Thus, ae is non-derivable. The two covers for ae are computed and the support is derived: $supp(ae) = 4$. Because $supp(ae) = l_{ae}$, item e will not be added to $\mathcal{D}^a$, because all supersets of ae must be derivable. This procedure continuous until all items are handled. In the construction of $\mathcal{D}^{\overline{d}\overline{e}}$, it turns out that ade is derivable, since $\delta_d(ade) = 2$, and $\delta_{\emptyset}(ade) = 2$, and hence $l_{ade} = u_{ade} = 2$.

The collection of frequent non-derivable itemsets in this example is:

$$\{\emptyset, a, b, c, d, e, ab, ac, ad, ae, bc, bd, be, ce, de, abc, abd\}$$

5 Experiments

The experiments were performed on a 1.5GHz Pentium IV PC with 1GB of main memory. To empirically evaluate the proposed dfNDI-algorithm, we performed several tests on the datasets summarized in the following table. For each dataset, the table shows the number of transactions, the number of items, and the average transaction length.

Dataset	# trans.	# items	Avg. length
BMS-POS	515 597	1 656	6,53
Chess	3 196	76	37
Pumsb	49 046	2 112	74
Mushroom	8 124	120	23

These datasets are all well-known benchmarks for frequent itemset mining. The *BMS-POS* dataset contains clickstream data from a small dot-com company that no longer exists. These two datasets were donated to the research community by Blue Martini Software. The *Pumsb*-dataset is based on census data, the *Mushroom* dataset contains characteristics from different species of mushrooms. The *Chess* dataset contains different game configurations. The Pumsb dataset is available in the UCI KDD Repository [14], and the Mushroom and Chess datasets can be found in the UCI Machine Learning Repository [6].

Obviously, as the number of non-derivable frequent itemsets is significantly smaller than the total number of frequent itemsets, it would be unfair to compare the per-

formance of the proposed algorithm with any other normal frequent itemset mining algorithm. Indeed, as soon as the threshold is small enough or the data is highly correlated, it is well known that traditional techniques fail, simply due to the massive amount of frequent sets that is produced, rather than any inherent algorithmic characteristic of the algorithms. Therefore, we merely compare the performance of dfNDI with NDI. Also note that a thorough comparison of the sizes of different condensed representations has already been presented in [10] and will not be repeated here.

In our experiments, we compared the NDI algorithm with three different versions of the dfNDI algorithm. The first version, 'dfNDI', is the regular one as described in this paper. The second version, 'dfNDI - negations', does not allow to use the covers of generalized itemsets, and hence, stores the full cover of every candidate itemset and always computes the covers by intersecting the covers of two subsets. The third version, 'dfndi + support order' is the dfNDI algorithm in which items are dynamically reordered in every recursion step accoring to their support, while the regular dfNDI algorithm orders the items according to the cover of the generalized itemset it represents. All experiments were performed for varying minimum support thresholds. The result can be seen in Figures 6 and 7.

First, we compared the time performance of these four algorithms. As expected, dfNDI performs much better compared to NDI. Using the smallest cover of a generalized itemset based on the current itemset also has a significant effect, which is especially visible in the pumsb dataset. This is of course mainly due to the faster set intersection and set difference operations on the smaller covers. Ordering the items according to the size of the covers also proves to be better as compared to ordering according to the support of the items, although the difference is never very big.

In the second experiment, we compare the memory usage of the four algorithms. As can be seen, the NDI algorithm needs least amount of memory. Of course, this is expected as the dfNDI also stores parts of the database in main memory. Fortunately, the optimization of storing only the smallest cover of each generalized itemset indeed shows to result in reduced memory usage. Again, the ordering based on the cover seems to result in a little bit better memory usage.

6 Conclusion

In this paper, we presented a new itemset search space traversal strategy, which allows depth-first itemset mining algorithms to exploit the frequency knowledge of all subsets of a given candidate itemset. Second, we presented a depth-first non-derivable itemset mining algorithm, which uses that traversal as it needs for every generated candidate itemset, all of its subsets. From the viewpoint of condensed representations of frequent sets, the non-derivable itemset

collection has already been shown to be superior to all other representations in almost all cases. Third, we generalized the diffsets technique and store the cover of an itemset containing several negated items. These covers are typically much smaller than the usual covers of regular itemsets, and hence, allow faster set intersection and set difference operations. The resulting algorithm, dfNDI, thus inherits all positive characteristics from several different techniques allowing fast discovery of a reasonable amount of frequent itemsets that are not derivable from their subsets. These claims are supported by several experiments on real life datasets.

References

[1] R. Agarwal, C. Aggarwal, and V. Prasad. Depth first generation of long patterns. In R. Ramakrishnan, S. Stolfo, R. Bayardo, Jr., and I. Parsa, editors, *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 108–118. ACM Press, 2000.

[2] R. Agarwal, C. Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, March 2001.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22(2) of *SIGMOD Record*, pages 207–216. ACM Press, 1993.

[4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. VLDB Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile, 1994.

[5] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *SIGKDD Explorations*, 2(2):66–75, 2000.

[6] C. Blake and C. Merz. *The UCI Repository of machine learning databases [http://www.ics.uci.edu/~mlearn/MLRepository.html]*. Irvine, CA: University of California, Department of Information and Computer Science, 1998.

[7] J.-F. Boulicaut and A. Bykowski. Frequent closures as a concise representation for binary data mining. In *Proc. PaKDD Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 62–73, 2000.

[8] T. Calders. Deducing bounds on the support of itemsets. In *Database Technologies for Data Mining*, chapter ?, pages ?–? Springer, 2003.

[9] T. Calders and B. Goethals. Minimal k-free representations of frequent sets. In *Proc. PKDD Int. Conf. Principles of Data Mining and Knowledge Discovery*, pages 71–82, 2002.

[10] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In *Proc. PKDD Int. Conf. Principles of Data Mining and Knowledge Discovery*, pages 74–85. Springer, 2002.

[11] J. Galambos and I. Simonelli. *Bonferroni-type Inequalities with Applications*. Springer, 1996.
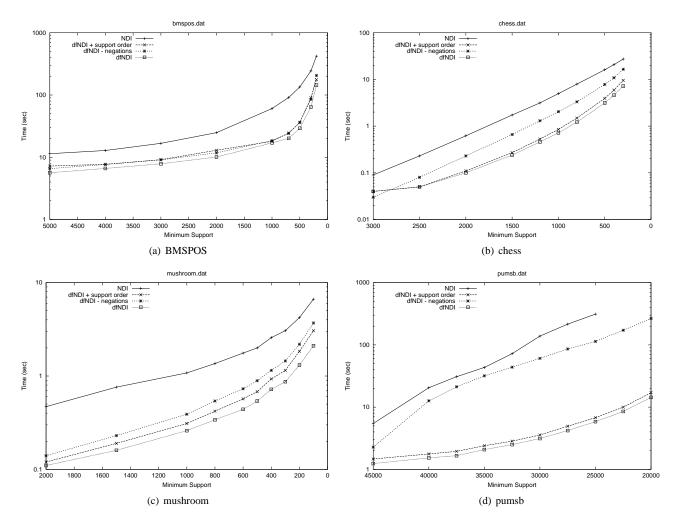
(a) BMSPOS

(b) chess

(c) mushroom

(d) pumsb

Figure 6: Performance comparison.

[12] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD Int. Conf. Management of Data*, pages 1–12, Dallas, TX, 2000.

[13] 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*, 2003. To appear.

[14] S. Hettich and S. D. Bay. *The UCI KDD Archive. [http://kdd.ics.uci.edu]*. Irvine, CA: University of California, Department of Information and Computer Science, 1999.

[15] H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In *Proc. KDD Int. Conf. Knowledge Discovery in Databases*, 1996.

[16] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. ICDT Int. Conf. Database Theory*, pages 398–416, 1999.

[17] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, Dallas, TX, 2000.

[18] M. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, May/June 2000.

[19] M. Zaki and K. Gouda. Fast vertical mining using diffsets. In L. Getoor, T. E. Senator, P. Domingos, and C. Faloutsos, editors, *Proceedings of the Eight ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 2003.

[20] M. Zaki and C. Hsiao. ChARM: An efficient algorithm for closed association rule mining. In *Technical Report 99-10, Computer Science, Rensselaer Polytechnic Institute*, 1999.

[21] M. Zaki and C. Hsiao. ChARM: An efficient algorithm for closed association rule mining. In *Proc. SIAM Int. Conf. on Data Mining*, 2002.

[22] M. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In R. Grossman, J. Han, V. Kumar, H. Mannila, and R. Motwani, editors, *Proceedings of the Second SIAM International Conference on Data Mining*, 2002.

[23] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In D. Heck-
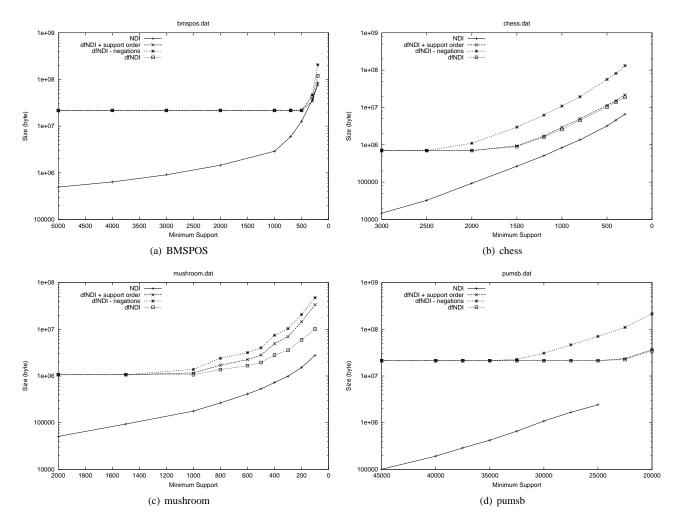
Figure 7: Memory usage comparison.

erman, H. Mannila, and D. Pregibon, editors, *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 283–286. AAAI Press, 1997.