Quick Inclusion-Exclusion

Toon Calders and Bart Goethals

University of Antwerp, Belgium
{toon.calders, bart.goethals}@ua.ac.be

Abstract. Many data mining algorithms make use of the well-known Inclusion-Exclusion principle. As a consequence, using this principle efficiently is crucial for the success of all these algorithms. Especially in the context of condensed representations, such as NDI, and in computing interesting measures, a quick inclusion-exclusion algorithm can be crucial for the performance. In this paper, we give an overview of several algorithms that depend on the inclusion-exclusion principle and propose an efficient algorithm to use it and evaluate its complexity. The theoretically obtained results are supported by experimental evaluation of the quick IE technique in isolation, and of an example application.

1 Introduction

The inclusion-exclusion (IE) principle is well known as it is an important method for many enumeration problems [8]. Also in many data mining applications this principle is used regularly. Moreover, as is typical in many data mining applications, when the formula is used, then it is evaluated many times. Indeed, data mining algorithms typically traverse huge pattern spaces in which hundreds to millions of potential patterns are evaluated. In this paper, we consider frequent itemsets and give an overview of several methods to efficiently evaluate the Inclusion-Exclusion formulas in order to obtain the supports of itemsets containing negated items. This leads us to the Quick Inclusion-Exclusion (QIE) algorithm, that is based on the same principles as the ADTree structure [13], and of which we show its efficiency in theory as well as in practice.

First, we shortly revisit the IE-principle and how it connects to itemsets and data mining.

Let $A_1, \ldots, A_n$ be n finite sets. The inclusion-exclusion principle is the following equality:

$$\left|\bigcup_{i=1}^{n} A_i\right| = \sum_{1\leq i\leq n} |A_i| - \sum_{1\leq i<j\leq n} |A_i \cap A_j| + \ldots - (-1)^n \left|\bigcap_{i=1}^{n} A_i\right|$$

We can connect the IE principle with frequent set mining as follows. 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, b, and d are the positive items, and $\overline{c}$ denotes the negation of c. We will often denote a generalized itemset $X \cup \overline{Y}$, where X is the set of positive items, and Y the set of items that are negated. For example, for the generalized itemset $\{a, b, \bar{c}, d\}$, $X = \{a, b, d\}$ and $Y = \{c\}$. A transaction T is said to *contain a general itemset* $G = X \cup \overline{Y}$ if $X \subseteq T$ and $T \cap Y = \emptyset$. 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 IE principle [8], we can now derive that for a given general itemset $G = X \cup \overline{Y}$ based on I,

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

Indeed; for all $y \in Y$, let A_y denote the set of transactions that contain $X \cup \{y\}$. Then, $\bigcup_{y \in Y} A_y$ denotes the set of transactions that contain X, and at least one item of Y. Hence, $|\bigcup_{y \in Y} A_y|$ equals $\mathit{support}(X) - \mathit{support}(G)$. This observation in combination with IE leads to the equation (1). The collection of formulas to compute the supports of all generalized itemsets based on *abcd* can be seen in Figure 1.

Note, if the supports of all strict subsets of I are known, from the support of one generalized itemset based on I, the support of all other generalized itemsets can be derived.

In the next section, we explain the uses of the IE principle within several frequent set mining tasks. Then, we present several algorithms that compute the supports of all generalized itemsets at once and show that the QIE algorithm is the most efficient algorithm to solve this problem. Several experiments illustrate the theoretically obtained results in Section 4 after which Section 5 ends with conclusions and future work.

2 Multiple Uses of IE

2.1 Support Estimation and bounding

Recently, several techniques have been developed to estimate the support of an itemset or the confidence of an association rule, based on the given supports of some sets [10–12, 14, 16]. The motivation for these techniques comes from the fact that the traditional support-confidence framework is well-suited to the market-basket problem, but is less appropriate for other types of transactional datasets. See, e.g., [16] for an extensive argumentation of this claim. Therefore, other interestingness measures have been developed.

The main idea is that interesting itemsets are ones that are both frequent (have required support) and have dependencies between the items. For example, consider items a and b. Assume that support measures are translated to probabilities (by dividing absolute support by number of database records). For example, $P(a)$ is the percentage of records with item a. To determine whether items a and b are independent (and hence not correlated), we need to check if the following 4 equations hold. Measures of correlation are based on the degree

$$
\begin{aligned}
support(\overline{abcd}) &= support(\emptyset) - support(a) - support(b) - support(c) - support(d) \\
&\quad + support(ab) + support(ac) + support(ad) \\
&\quad + support(bc) + support(bd) + support(cd) \\
&\quad - support(abc) - support(abd) - support(acd) - support(bcd) \\
&\quad + support(abcd) \\
support(a\overline{bcd}) &= support(a) - support(ab) - support(ac) - support(ad) \\
&\quad + support(abc) + support(abd) + support(acd) - support(abcd) \\
support(\overline{a}b\overline{cd}) &= support(b) - support(ab) - support(bc) - support(bd) \\
&\quad + support(abc) + support(abd) + support(bcd) - support(abcd) \\
support(\overline{ab}c\overline{d}) &= support(c) - support(ac) - support(bc) - support(cd) \\
&\quad + support(abc) + support(acd) + support(bcd) - support(abcd) \\
support(\overline{abc}d) &= support(d) - support(ad) - support(bd) - support(cd) \\
&\quad + support(abd) + support(acd) + support(bcd) - support(abcd) \\
support(ab\overline{cd}) &= support(ab) - support(abc) - support(abd) + support(abcd) \\
support(a\overline{b}c\overline{d}) &= support(ac) - support(abc) - support(acd) + support(abcd) \\
support(a\overline{bc}d) &= support(ad) - support(abd) - support(acd) + support(abcd) \\
support(\overline{a}bc\overline{d}) &= support(bc) - support(abc) - support(bcd) + support(abcd) \\
support(\overline{a}b\overline{c}d) &= support(bd) - support(abd) - support(bcd) + support(abcd) \\
support(\overline{ab}cd) &= support(cd) - support(acd) - support(bcd) + support(abcd) \\
support(abc\overline{d}) &= support(abc) - support(abcd) \\
support(ab\overline{c}d) &= support(abd) - support(abcd) \\
support(a\overline{b}cd) &= support(acd) - support(abcd) \\
support(\overline{a}bcd) &= support(bcd) - support(abcd) \\
support(abcd) &= support(abcd)
\end{aligned}
$$

Fig. 1. The IE formulas for the itemset *abcd*.

to which these equations are violated.

$$
\begin{aligned}
P(a,b) &= P(a)P(b) \qquad & P(a,\overline{b}) &= P(a)P(\overline{b}) \\
P(\overline{a},b) &= P(\overline{a})P(b) \qquad & P(\overline{a},\overline{b}) &= P(\overline{a})P(\overline{b})
\end{aligned}
$$

Observe that standard frequent itemset mining algorithms (e.g., Apriori) only provide information needed to check the first of these equations. However, all is not lost. Given $P(a)$, $P(b)$, and $P(a,b)$, we can derive exact values for $P(\overline{a},b)$, $P(a,\overline{b})$, and $P(\overline{a},\overline{b})$ by evaluating the Inclusion-Exclusion formulas for these terms without taking any additional counts.

The following four approaches are examples of similar measures, that also require the supports of itemsets with negations, and hence for which a quick inclusion-exclusion algorithm is useful:

- The dependency estimate of Silverstein et al., based on the χ^2 test [16].
- The dependence value of Meo, based on maximal entropy [12].
- The non-derivable itemsets (NDIs), based on tight support bounding [6].
- The support quota of Savinov [15], combining the dependency values of [12] and the tight support bounding of [6].

In this paper we will show a method to compute the IE sums in time $\mathcal{O}(n2^n)$. This exponential cost may seem unrealistically high for real applications. In reality, however, in the applications we describe, the IE sums need to be computed mostly for relatively small itemsets. For an empirical proof of the feasibility of computing IE sums, we refer to the experimental section, where it is shown that for the computation of NDIs, the exponential cost is reasonable.

We now discuss the four approaches in more detail.

Dependency Estimate In [16], a χ^2-test is used to test the (in)dependence of items in an itemset. An itemset is only considered interesting if the items are dependent at a given significance level. The test of dependence is as follows. First, a contingency table for the itemset is constructed. This contingency table contains an entry for every combination of occurence/absence of the items in the itemset. Hence, the cells in the contingency table are exactly the supports of every generalized itemset based on the set. This contingency table is then compared to the estimates for the cells under the assumption of statistical independence. For the cell holding the support of $X \cup \overline{Y}$, this independence estimate is

$$E(X \cup \overline{Y}) \;:=\; |\mathcal{D}| \cdot \prod_{x \in X} \frac{\mathit{support}(\{x\})}{|\mathcal{D}|} \cdot \prod_{y \in Y} \frac{|\mathcal{D}| - \mathit{support}(\{y\})}{|\mathcal{D}|} \quad .$$

Then, the χ^2-score is used to quantify the difference between the observed counts and the estimated counts. This degree of independence for a set I is:

$$\chi^2(I) \;:=\; \sum_{X \cup \overline{Y} \text{ based on } I} \frac{(\mathit{support}(X \cup \overline{Y}) - E(X \cup \overline{Y}))^2}{E(X \cup \overline{Y})} \quad .$$

The set is then called *dependent at significance level* α if $\chi^2(I)$ exceeds the cut-off value χ^2_α.

In [16], an algorithm to find all dependent itemsets that also satisfy a support constraint is given. In this algorithm, the contingency tables are constructed by scanning the complete database. Because scanning the transaction database for every candidate separately can be very costly, in [16], the contingency tables of all candidates at the same level are constructed in one pass over the database.

The construction of the contingency tables in the algorithm of [16], however, has two big disadvantages: first, scanning the database can be very costly, especially for large datasets. Second: the contingency tables grow exponentially with the size of the itemset. Therefore, maintaining all contingency tables in memory simultaneously results in gigantic memory requirements. Therefore, we propose the use of the inclusion-exclusion principle for the construction of the contingency tables instead. Indeed, the cells in the contingency table are exactly the supports of the generalized itemsets, and, as was shown in the introduction, for a given set, the support of all its generalized itemsets based can be computed, based solely on the supports of all its subsets.

Notice that this use of the inclusion-exclusion principle goes far beyond the algorithm of [16] alone; every algorithm using contingency tables can benefit from it, and many statistical measures use contingency tables.

Dependency Values In [12], Meo addresses the following problem with the estimate of [16]. A major drawback of the framework proposed in [16] is that in the estimation of the support of the contingency table entries, only the supports of the singleton itemsets are used. In this way, it is possible (and even often the case) that the estimated supports are inconsistent with the supports of itemsets of higher length. Meo addresses this problem by adopting a maximal entropy model to estimate the support of an itemset. Let I be an itemset for which we want to estimate the support, based on the supports of all its strict subsets. First, the notion of the entropy of a transaction database is defined. In general, entropy is defined as a measure on probability distributions. Let $\Omega = \{\omega_1, \ldots, \omega_m\}$ be a set of possible outcomes of an experiment. Let X be a probability distribution that assigns probability p_i to ω_i, for $i = 1 \ldots m$. The entropy of X is then defined as $\sum_{i=1}^{n} p_i \cdot \ln(p_i)$.

To define the entropy of a transaction database, it suffices to regard the database as a probability distribution. When we are interested in the itemset I, we can view the database as a probability structure assigning probabilities to the generalized itemsets based on I. That is, the different generalized itemsets are the "events", and their probability is their support divided by the total number of transactions in the database. From this viewpoint, the entropy of the database when restricted to the itemset I, denoted $\mathcal{E}_I(\mathcal{D})$, is defined as

$$\sum_{X \cup \overline{Y} \text{ based on } I} \frac{support(X \cup \overline{Y})}{|\mathcal{D}|} \cdot \ln\left(\frac{support(X \cup \overline{Y})}{|\mathcal{D}|}\right) .$$

Remember from Section 1, that if we know the supports of all strict subsets of I, then from the support of I, the support of all generalized itemsets based on I can be derived. The maximal entropy estimate for the support of I now is the one that maximizes the entropy $\mathcal{E}_I(\mathcal{D})$. In [12], based on the maximal entropy estimate, the notion of *Dependence Values* of an itemset is defined as the difference of this estimated support and the actual support of the itemset. For the exact details on the computation, we refer to [12], but for here it suffices that again all IE formulas need to be computed. In [12], these IE sums are calculated in isolation. This will correspond to our brute force evaluation method which we improve upon significantly in this paper. As the experiments will show, the gain of the quick inclusion-exclusion is large, which implies that the application of our quick IE-computation improves the performance of determining dependence values significantly.

2.2 Non-Derivable Itemsets

In [6], tight bounds for an itemset are given for the case in which the supports of all its subsets are known. That is, from the supports of the strict subsets of I, a lower bound l and an upper bound u are calculated, such that the support of I must be in the interval $[l, u]$. A set is considered uninteresting if its lower bound equals its upper bound, because this equality implies that the support of the itemset is completely determined by the supports of its subsets. Such a set is called a derivable itemset. In [6], an algorithm is given to find all non-derivable itemsets.

The bounds in [6] are based on the inclusion-exclusion principle. Recall the equality

$$\mathit{support}(X \cup \overline{Y}) \;=\; \sum_{X \subseteq J \subseteq I} (-1)^{|J \setminus X|} \mathit{support}(J) \;\; .$$

Since $\mathit{support}(X \cup \overline{Y})$ is always positive, we get

$$0 \;\leq\; \sum_{X \subseteq J \subseteq I} (-1)^{|J \setminus X|} \mathit{support}(J) \;\; . \tag{2}$$

In [6], this observation was used to calculate lower and upper bounds on the support of an itemset I, by isolating $\mathit{support}(I)$ in (2). For each set I, let l_I (u_I) denote the lower (upper) bound we can derive using these deduction rules. That is,

$$l_I = \max\{- \sum_{X \subseteq J \subset I} (-1)^{|J \setminus X|} \mathit{support}(J) \mid X \subseteq I, |I \setminus X| \text{ odd}\},$$

$$u_I = \min\{ \sum_{X \subseteq J \subset I} (-1)^{|J \setminus X|} \mathit{support}(J) \mid X \subseteq I, |I \setminus X| \text{ even}\}.$$

Notice that these sums only differ little from the IE-sums we are optimizing. In fact, the sums coincide when we set $\mathit{support}(I)$ equal to 0. Therefore, our quick inclusion-exclusion technique directly leads to an efficient procedure for computation of bounds on the support of an itemset.

Notice that for the bounds in [6], the supports of all subsets of I must be known. This is often the case (e.g. in levelwise algorithms), but not always. In these cases, approximate inclusion-exclusion techniques can be used to find bounds on the support of an itemset. In [7], e.g., bounds on the support of an itemset are given when the support of all subsets up to a certain size only are known. These bounds are based on the so-called Bonferoni inequalities, which are an extension of the inclusion-exclusion principle.

Support Quotas In [15], Savinov proposes the use of *support quotas* to improve the performance of mining the dependence rules of [12]. The support quota of an itemset is defined as the size of the bounding interval for its support as in [6]. Let $[l, u]$ be the bounds on the support of I. The support of I must always be in this interval. If the estimate $e(I)$ is consistent with the supports of the subsets of I, there must exist a database that is consistent with the supports of all subsets of I, and with $support(I) = e(I)$. For example, the maximum entropy estimate of [12] is in this case. Therefore, the difference between $e(I)$ and the actual support of I can maximally be $u - l$. If $u - l$ is smaller than the minimal dependence value, the difference between the estimate and the actual support will be smaller than this threshold as well and hence can be pruned. Moreover, since the interval width decreases when going from sub- to superset, all supersets of I can be pruned as well. This is a very interesting situation, as the dependence value is non-monotonic and thus not allows for pruning supersets. By using support bounding, however, an upper bound on the dependence values can be found that is monotonic. Even though Savinov's technique was introduced specifically for the dependence rules of [12], it can be extended to improve every estimate that is consistent with given supports, leading to yet another important application of quick IE.

2.3 Condensed Representation

The use of the quick inclusion-exclusion technique goes beyond advanced interesting measures for itemsets. In [11], for example, Mannila et al consider the collection of frequent sets as a condensed representation that allows to speed up support counts of arbitrary Boolean expressions over the items. In this context, the inclusion-exclusion principle can be used as a mean to estimate the support of arbitrary Boolean formulas based on the support of the frequent itemsets alone. Our quick IE method can here be used to quickly find the support of all conjunctive Boolean formulas with negations.

Also when we are only interested in the frequent itemsets, condensed representation are very useful, since the collection of all frequent itemsets can already be far too large to store. In the literature, many different condensed representations have been studied. In [5], the free sets [3], disjunction-free sets [4], generalized disjunction-free sets [9], and the non-derivable sets [6] are all shown to be based on the same support bounding technique which is based on the inclusion-exclusion formulas. For the exact details of this connection we refer to [5]. Because of this connection, improving the efficiency of the inclusion-exclusion computation results in performance gains when constructing one of these condensed representations.

3 QIE: The Algorithm

We first start with a formal problem definition.

Definition 1. *Let I be an itemset. Suppose that of every subset of I its support has en given. The* IE problem *is to compute for every subset X of I, the sum*

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

```
Require: $I \subseteq \mathcal{I}$, $support(J)$ for all $J \subseteq I$
Ensure: $support(X \cup \overline{Y})$ for all $X \dot\cup Y = I$
1: for all $X \subseteq I$ do
2:     $support(X \cup \overline{Y}) := 0$
3:     for all $J \supseteq X$ do
4:         find $support(J)$
5:         $support(X \cup \overline{Y}) += (-1)^{|J \setminus X|} support(J)$
6:     end for
7: end for
```

Fig. 2. BFIE: Brute Force IE.

Obviously, efficiently computing IE is crucial for the success of all previously discussed methods. Nevertheless, for a given itemset I, there exist $2^{|I|}$ rules, and every such rule, with $I = X \cup Y$, consist of $2^{|Y|}$ terms, resulting in a total number of terms equal to

$$\sum_{X \subseteq I} 2^{|I \setminus X|} = \sum_{i=0}^{|I|} \binom{|I|}{i} 2^i = 3^{|I|} \quad .$$

In what follows, we present several techniques to evaluate these rules efficiently. To compare their costs, we assume all itemsets are stored in a trie-like data structure [2]. Finding the support of a single itemset of size k in such a trie requires exactly k lookup operations. The cost model we use, assigns a cost of 1 to every lookup operation, and thus a cost of k for the retrieval of the support of an itemset of size k. In theory, however, this cost is in worst case as high as $\log(|\mathcal{I}|)$, but in practice, and with the use of advanced indexing techniques such as hash tables, the cost of 1 for every lookup operation is realistic. Notice that we could also hash the itemsets directly, and thus have a cost of $\mathcal{O}(1)$ per *itemset* that needs to be looked up. Nevertheless, the computation of any reasonable hash-key will be linear in the size of the set. For one computation this linearity does not matter and can be omitted from the complexity analysis. In our case, however, we need to incorporate the fact that this computation needs to be done for many itemsets.

3.1 Brute Force IE

A brute force algorithm would simply evaluate all rules separately and fetch all supports one at a time, as shown in Fig. 2.

The total cost of this algorithm is captured in the following Lemma.

Lemma 1. *For a given itemset I, with $|I| = n$, computing the supports of all generalized itemsets $X \cup \overline{Y}$, with $X \cup Y = I$, using the brute force algorithm, comes down to a cost of $2n3^{n-1}$.*

Proof. Computing the support of $X \cup \overline{Y}$ requires the retrieval of $\binom{|Y|}{0}$ sets of size $|X| = k$, $\binom{|Y|}{1}$ sets of size $|X| + 1$, ..., $\binom{|Y|}{|Y|}$ sets of size $|X \cup Y| = |I| = n$,

```
Require: Root node $n$, $X \cup \overline{Y}$
Ensure: $support(X \cup \overline{Y})$
 1: if $X \cup \overline{Y}$ is empty then
 2:     return $n.support$;
 3: end if
 4: {Let $i$ be the first item in $X \cup \overline{Y}$}
 5: if $i$ in $X$ then
 6:     $X := X \setminus \{i\}$;
 7:     return $CIE(n \rightarrow i, X \cup \overline{Y})$;
 8: else
 9:     $Y := Y \setminus \{i\}$;
10:     return $CIE(n, X \cup \overline{Y}) - CIE(n \rightarrow i, X \cup \overline{Y})$;
11: end if
```

Fig. 3. CIE: Combined IE for a single $support(X \cup \overline{Y})$.

which amounts to

$$\sum_{i=0}^{n-k} \binom{n-k}{i}(i+k) \;=\; (n+k)2^{n-k-1} \;.$$

Thus, evaluating the supports for all $X \cup \overline{Y}$ comes down to

$$\sum_{k=0}^{n} \binom{n}{k}(n+k)2^{n-k-1} \;=\; 2n3^{n-1} \;.$$

3.2 Combined IE

Instead of retrieving the support of all supersets of X separately, this can be done in a single large retrieval operation combined with the computation of the support of $X \cup \overline{Y}$. Indeed, the items in X occur in all itemsets we retrieve, while the items in Y are "optional". This observation is reflected in the recursive procedure illustrated in Figure 3.

Initially, the procedure starts in the root node, and scans over the items in $X \cup Y$. If the current item is in X, then it is found among the children of the current node and the procedure recursively continues from this node for the remaining items. If the current item is in Y, then the computation is split into two paths; on one path, the item is ignored, and the procedure recursively continues from the current node for the remaining items; on the other path, the item is found among the children of the current node and the procedure recursively continues from this node for the remaining items. In this way, the different computation paths end up in exactly the supersets of X, the supports are returned and, depending on their cardinality, added or subtracted. In Figure 4, this procedure is illustrated for the set $\overline{a}bc\overline{d}$. The arrows indicate the recursion, the encircled nodes the itemsets that are summed.

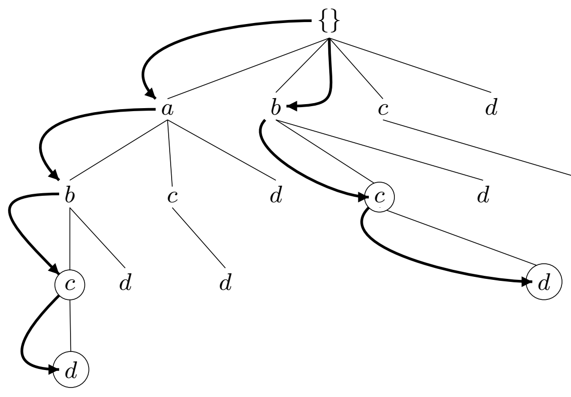

Fig. 4. Example trace of combined IE algorithm for a single $\overline{a}bc\overline{d}$.

Lemma 2. *For a given itemset I, with $|I| = n$, computing the supports of all generalized itemsets $X \cup \overline{Y}$, with $X \cup Y = I$, using the combined IE algorithm, comes down to a cost of* $2(3^n - 2^n)$.

Proof. The total cost of computing the supports of all generalized itemsets consists of the total number of visits to non-root nodes in the itemset trie.

Consider a node in the trie associated with itemset J. This node is visited in the computation of the support for all generalized itemset $X \cup \overline{Y}$, such that there is a superset of X that is in the trie below the node for J. Because the trie below J contains all sets with J as prefix, this means that there must be a superset of X such that J is a prefix of this superset. Let j be the last item in the set J. The trie below J contains the supports of all sets $J \cup J'$ with J' a subset of $\{i \in \mathcal{I} \mid i > j\}$. Hence, there exists a superset of X that is below J in the itemset trie, if and only if X is a subset of $J \cup \{i \in \mathcal{I} \mid i > j\}$. Let $m = |\{i \in \mathcal{I} \mid i < j, i \notin J\}|$ be the number of so-called "missing items" in J. The number of times J is visited during the combined IE algorithm depends on this number of missing items m, and is 2^{n-m}.

To make the total sum of visits, we still need to determine the number of nodes with m missing items. This number equals $\binom{n}{m+1}$. Indeed, consider an itemset with m missing items. This itemset is completely characterized by the list of the m missing items, and its last item. Therefore, the number of itemsets with m missing items is exactly the number of combinations of $m + 1$ out of n.

Combining these two results, the total number of visits to nodes in the trie can be obtained:

$$\sum_{m=0}^{n-1} \binom{n}{m+1} 2^{n-m} \;=\; 2(3^n - 2^n) \;.$$

```
Require: $I \subseteq \mathcal{I}$, $support(J)$ for all $J \subseteq I$
Ensure: $support(X \cup \overline{Y})$ for all $X \dot{\cup} Y = I$
1: for all $X \subseteq I$ do
2:     $A[X_b] := support(X)$;
3: end for
4: for $i := 1$ to $2^{|I|}$ do
5:     {Let X be the itemset for which $X_b = i$}
6:     for all $J \subseteq I$, such that $J \supset X$ do
7:         $A[i] := A[i] + (-1)^{|J \setminus X|} A[J_b]$;
8:     end for
9: end for
```

Fig. 5. NIE: Naive IE algorithm using direct access to the itemset supports.

3.3 Direct Access

Although the previous method already combines the retrieval of the supports of several itemsets in a single scan through the itemset trie, this still needs to be done for every possible subset of I. Fortunately, it is also possible to collect the support of all subsets of I once, store it in a specialized storage structure, and access this structure for every set X. Preferably, the specialized structure does not introduce too much overhead, and allows for fast access to the supports of the supersets of a set X. These requirements can be met with a simple linear array, and an indexing pattern based on the bit-pattern of the itemsets. From now on, we assume that the items of $\mathcal{I}$ are ordered.

To store the supports of the subsets of an itemset I, we create an array of size $2^{|I|}$, denoted by A, and the ith entry of A is denoted by $A[i]$. Then, the *bitpattern* of an itemset $X \subseteq I$, denoted by X_b, is simply the sequence $x_1 \ldots x_{|I|}$, where $x_j = 1$ if $i_j \in X$, and $x_j = 0$ otherwise. The *index* of X is the number represented by the bitpattern of X. Hence, this index can be used to directly access the entry in A storing the support of X.

The array structure, and the bitpattern access method have several interesting properties.

1. Enumerating all subsets of I comes down to a for loop from 0 to $2^{|I|}$.
2. The indices of the supersets of a set X can be enumerated by switching some of the 0-bits to 1 in the bitpattern of X.
3. The order in which the subsets are stored is also known as the *reverse pre-order*. This order has the interesting property that, given two sets X, X', such that $X \subseteq X' \subseteq I$, the index of X will be smaller than the index of X'. Therefore, we compute the support of $X \cup \overline{Y}$ in ascending order of the index of X. After that, we can simply replace the entry containing the support of X with the support of $X \cup \overline{Y}$ as we do not need its support anymore anyway.

Given this array containing the supports of all subsets of I, we automatically obtain the naive algorithm that sums for each $X \subset I$, the supports of all supersets of X. The exact algorithm is shown in Fig. 5 and illustrated in Figure 6. The arrows in the figure represent all necessary additions or subtractions.

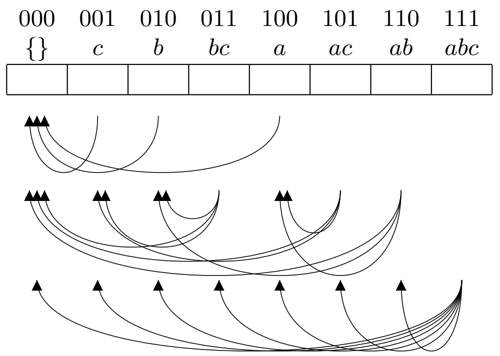

Fig. 6. Illustration of the naive IE algorithm using direct access to the itemset supports, for the generalized itemsets based on *abc*.

The first three lines store the support of each subset of I in the array. Then, in line 4, a for-loop traverses each subset X in ascending order of its index. In the nested for-loop, all supports of all supersets of X are added (subtracted) to the support of X, resulting in the support of $X \cup \overline{I \setminus X}$, stored at index X_b of the array.

Lemma 3. *For a given itemset I, with $|I| = n$, computing the supports of all generalized itemsets $X \cup \overline{Y}$, with $X \dot{\cup} Y = I$, using the naive IE algorithm with direct access to the itemset supports, comes down to a cost of 3^n.*

Proof. Retrieving the supports of all subsets of I has a cost of $2^{|I|}$. For each generalized itemset $X \cup \overline{Y}$, the IE formulas consist of $2^{|Y|} - 1$ operations, resulting in a total of exactly $3^{|I|} - 2^{|I|}$ operations over all generalized itemsets. Hence, the total cost equals $2^{|I|} + 3^{|I|} - 2^{|I|} = 3^{|I|}$.

Until now, we have mainly concentrated on optimizing the number of costly retrievals in the itemset trie by introducing an array for the itemset supports with an efficient indexing structure. In this way, the number of retrieval operations could be lowered from 3^n to 2^n. The number of additions, however, remains $3^n - 2^n$. Even though the cost of *one* addition is negligible compared to the cost of one retrieval operation, the cost of $3^n - 2^n$ additions quickly grows far beyond the cost of 2^n retrieval operations.

3.4 Quick IE

The retrieval of the supports of all subsets is not the only operation that can be shared among the different inclusion-exclusion computations. Indeed, many of the inclusion- exclusion sums share a considerable number of terms. Therefore, by sharing part of the computation of the sums, a considerable number of additions can be saved. For example, consider the sums for $ab\overline{cd}$ and $a\overline{bcd}$:

$$ab\overline{cd} = ab - abc - abd + abcd$$
$$a\overline{bcd} = a - \mathbf{ab} - ac - ad + \mathbf{abc} + \mathbf{abd} + acd - \mathbf{abcd}$$

```
Require: $I \subseteq \mathcal{I}$, $support(J)$ for all $J \subseteq I$
Ensure: $support(X \cup \overline{Y})$ for all $X \dot{\cup} Y = I$
 1: for all $X \subseteq I$ do
 2:     $A[X_b] := support(X)$;
 3: end for
 4: for $l := 2;l < 2^{|I|};l := 2l$ do
 5:     for $i := 1;i < 2^{|}I|; i+ = l$ do
 6:         for $j := 0$ to $l - 1$ do
 7:             $A[i + j] := A[i + j] - A[i + l/2 + j]$;
 8:         end for
 9:     end for
10: end for
```

Fig. 7. QIE: Quick IE algorithm.

Hence, if we first compute $support(ab\overline{cd})$, and then use

$$a\overline{bcd} \;=\; a\overline{cd} - ab\overline{cd} \;=\; a - ac - ad + acd - ab\overline{cd}$$

we save 3 additions. In general, for a generalized itemset G, and an item a not in G, the following equality holds:

$$support(\overline{a}G) = support(G) - support(aG) \ .$$

This fact can now be exploited in a systematic manner as in Fig. 7.

Again, the algorithm starts by filling an array with the supports of all subsets of I. In the end, the entry for X in the array will contain the support of $X \cup \overline{Y}$, with $X \dot{\cup} Y = I$. To get to this support, the entries of A are iteratively updated. Let $I = \{i_1, \ldots, i_n\}$. After the jth iteration, the entry for X will contain the support of the generalized set $X \cup \overline{\{i_{n-j+1}, \ldots, i_n\} \setminus X}$. In other words, in the jth iteration, the entries for all X that do not contain item i_{n-j+1} are updated by adding $\overline{i_{n-j+1}}$ to it, and updating its support accordingly.

For example, let I be the itemset $\{a, b, c\}$. Before the procedure starts, array A contains the following supports:

000	001	010	011	100	101	110	111
$\{\}$	c	b	bc	a	ac	ab	abc

In the first iteration, item c is handled. This means that in this iterations, the entries of all sets X that do not contain c are updated to contain the support of $X \cup \overline{c}$. Hence, after the first iteration, the array contains the following supports:

000	001	010	011	100	101	110	111
$\overline{c}$	c	$b\overline{c}$	bc	$a\overline{c}$	ac	$ab\overline{c}$	abc

In the second iteration, item b is handled. Thus, after the second iteration, the array contains the following supports:

000	001	010	011	100	101	110	111
$\overline{bc}$	$\overline{b}c$	$b\overline{c}$	bc	$a\overline{bc}$	$a\overline{b}c$	$ab\overline{c}$	abc

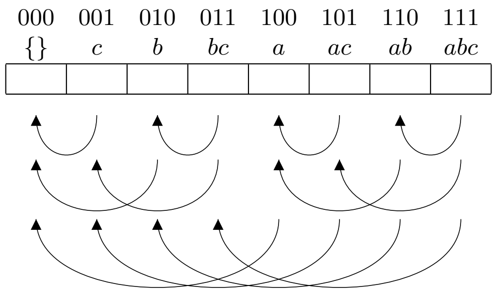

Fig. 8. Visualization of the QIE algorithm.

In the third and last iteration, item a is handled, giving the final array:

000	001	010	011	100	101	110	111
$\overline{a}bc$	$\overline{ab}c$	$\overline{a}b\overline{c}$	$\overline{a}bc$	$a\overline{bc}$	$a\overline{b}c$	$ab\overline{c}$	abc

Lemma 4. *For a given itemset I, with $|I| = n$, computing the supports of all generalized itemsets $X \cup \overline{Y}$, with $X \dot{\cup} Y = I$, using the Quick IE algorithm, comes down to a cost of $2^n + n2^{n-1}$.*

Proof. The first term comes from filling the array with all subsets of I. Then, for every item $i \in I$, we update all itemsets not containing i, i.e. exactly 2^{n-1}.

Notice that the principle used in QIE, is also used in ADTrees [13], in the more general context of relations with categorical attributes, in order to speed up the computation of contingency tables. An ADTree is a datastructure that stored the counts of some queries over the relation. If now a transaction database is considered as a relation with binary attributes, and the construction of the ADTree is slightly modified such that only counts of itemsets are stored, the computation of a contingency table in ADTree, and the computation of the supports of all general itemsets by QIE will become very similar.

3.5 Summary

The memory and time requirements of the different methods discussed in this section are summarized in the following table. n denotes the number of items in an itemset I, for which the supports of all generalized itemsets based on it are computed. For the space requirement, we only report the memory required for the computation, not the input-, or the output size. This choice is motivated by the fact that the frequent itemsets and their supports can be stored in secondary storage, and the output can either directly be filtered for patterns meeting a support threshold, or be written to a database. Hence, the space requirement is the amount of main memory needed to compute the inclusion-exclusion. Notice also that reporting the total memory requirement instead would be far less informative, as it would yield a lower bound of $\mathcal{O}(2^n)$ for all methods.

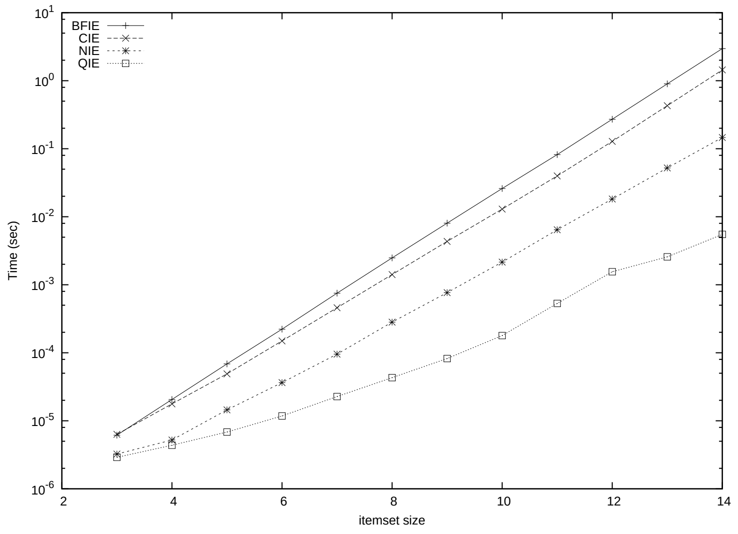

Fig. 9. Time needed to compute the generalized itemsets.

Method	Space	Time
Brute force	constant	$2n3^{n-1} = \mathcal{O}(n3^n)$
Combined, no direct access	constant	$2(3^n - 2^n) = \mathcal{O}(3^n)$
Combined, direct access	$\mathcal{O}(2^n)$	$3^n = \mathcal{O}(3^n)$
QIE	$\mathcal{O}(2^n)$	$2^n + n2^{n-1} = \mathcal{O}(n2^n)$

From this table we can conclude that, if the itemset is small, the QIE method is clearly the best. In the case, however, that the itemsets are stored in secondary storage, n is large, and memory is limited, the combined method without direct access is preferable.

4 Experiments

We implemented the well known Apriori algorithm [1] and adapted it to incorporate all proposed techniques for computing the supports of all generalized itemsets. The experiments were ran on a 1.2 GHz Pentium IV using 1GB of memory. More specifically, for every candidate itemset, we recorded the amount of time needed to compute the supports of all generalized itemsets. The results of this experiment are shown in Figure 9. In this figure, the average running time per itemset length has been given. The dataset we used for this experiment was the BMS-Webview-1 dataset donated by Kohavi et al. [17]. We experimented on several datasets as well, but only report on BMS-Webview-1, as these figures are independent of the dataset used. As expected, the algorithms behave as shown in theory. (Note the logarithmic y axes.)

For further evaluation, we also implemented the presented techniques in our NDI implementation to see what the effect would be in a real application.

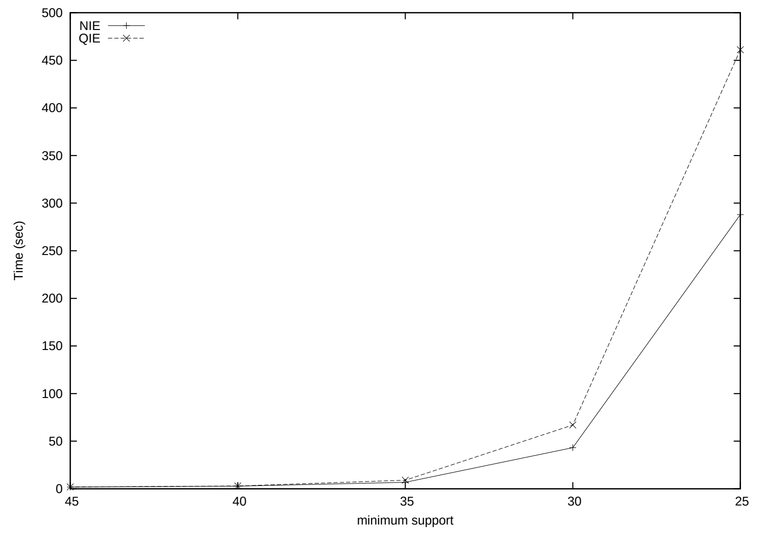

Fig. 10. Peformance improvement of the NDI algorithm for BMS-Webview-1.

We performed our experimented on the BMS-Webview-1 and BMS-Webview-2 datasets [17], and the results turned out to be very nice. The results are shown in Figure 10 for BMS-Webview-1 and in Figure 11 for BMS-Webview-2. We only show the results for the NIE and QIE algorithm as these are the two fastest.

Although Non-Derivable Itemsets are shown to be small in general [6], and the computation of the IE formulas in the candidate generation phase is only a small part of the total cost of the algorithm, we observe remarkable speedups showing the applicability for the proposed QIE algorithm. For example, in BMS-Webview-1, for the lowest threshold, a speedup of more that 200 seconds was obtained.

5 Conclusion

We presented an overview of algorithms to compute the supports of so called generalized itemsets, i.e. itemsets in which items are allowed to be negated. We explained how this can be done without going back to the database, using the principle of Inclusion-Exclusion. We showed that many data mining applications could benefit from this principle in case an efficient algorithm existed. The QIE algorithm is theoretically and experimentally shown to be extremely efficient compared to several other techniques.

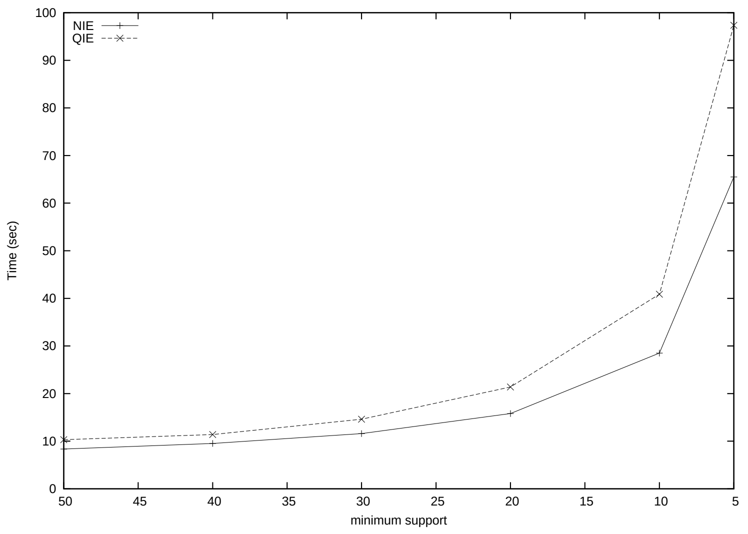

Fig. 11. Peformance improvement of the NDI algorithm for BMS-Webview-2.

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J.B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings 20th International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.
2. C. Borgelt and R. Kruse. Induction of association rules: Apriori implementation. In W. Härdle and B. Rönz, editors, *Proceedings of the 15th Conference on Computational Statistics*, pages 395–400, `http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html`, 2002. Physica-Verlag.
3. J.-F. Boulicaut, A. Bykowski, and C. Rigotti. Approximation of frequency queries by means of free-sets. In *Proc. PKDD Int. Conf. Principles of Data Mining and Knowledge Discovery*, pages 75–85, 2000.
4. A. Bykowski and C. Rigotti. A condensed representation to find frequent patterns. In *Proc. PODS Int. Conf. Principles of Database Systems*, 2001.
5. 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.
6. 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.
7. S. Jaroszewicz and D. A. Simivici. Support approximations using bonferroni-type inequalities. In *Proc. PKDD Int. Conf. Principles of Data Mining and Knowledge Discovery*, pages 212–224, 2002.
8. D.E. Knuth. *Fundamental Algorithms.* Addison-Wesley, Reading, Massachusetts, 1997.

9. M. Kryszkiewicz and M. Gajek. Why to apply generalized disjunction-free generators representation of frequent patterns? In *Proc. International Syposium on Methodologies for Intelligent Systems*, pages 382–392, 2002.
10. H. Mannila. Local and global methods in data mining: Basic techniques and open problems. In *ICALP 2002, 29th International Colloquium on Automata, Languages, and Programming*, 2002.
11. H. Mannila and H. Toivonen. Multiple uses of frequent sets and condensed representations. In *Proc. KDD Int. Conf. Knowledge Discovery in Databases*, 1996.
12. R. Meo. Theory of dependence values. *ACM Trans. on Database Systems*, 25(3):380–406, 2000.
13. A. Moore and M.S. Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8:67–91, 1998.
14. D. Pavlov, H. Mannila, and P. Smyth. Beyond independence: Probabilistic models for query approximation on binary transaction data. *IEEE Trans. on Knowledge and Data Engineering*, 15(6):1409–1421, 2003.
15. A. Savinov. Mining dependence rules by finding largest support quota. In *ACM Symposium on Applied Computing*, pages 525–529, 2004.
16. Craig Silverstein, Sergey Brin, and Rajeev Motwani. Beyond market baskets: Generalizing association rules to dependence rules. *Data Mining and Knowledge Discovery*, 2(1):39–68, 1998.
17. Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proc. KDD Int. Conf. Knowledge Discovery in Databases*, pages 401–406. ACM Press, 2001.