Finding Closed Frequent Item Sets by Intersecting Transactions

Christian Borgelt
European Centre for Soft Computing
c/ Gonzalo Gutiérrez Quirós s/n
E-33600 Mieres (Asturias), Spain
christian.borgelt@softcomputing.es

Xiaoyuan Yang
Telefonica Research
Via Augusta, 177
E-08021 Barcelona
yxiao@tid.es

Ruben Nogales-Cadenas
Facultad de Ciencias Físicas
Universidad Complutense
E-28040 Madrid, Spain
ruben.nogales@fdi.ucm.es

Pedro Carmona-Saez
Facultad de Ciencias Físicas
Universidad Complutense
E-28040 Madrid, Spain
pcarmona@fis.ucm.es

Alberto Pascual-Montano
Functional Bioinformatics Group
National Center for Biotechnology-CSIC
E-28040 Madrid, Spain
pascual@cnb.csic.es

ABSTRACT

Most known frequent item set mining algorithms work by enumerating candidate item sets and pruning infrequent candidates. An alternative method, which works by intersecting transactions, is much less researched. To the best of our knowledge, there are only two basic algorithms: a cumulative scheme, which is based on a repository with which new transactions are intersected, and the Carpenter algorithm, which enumerates and intersects candidate transaction sets. These approaches yield the set of so-called closed frequent item sets, since any such item set can be represented as the intersection of some subset of the given transactions. In this paper we describe a considerably improved implementation scheme of the cumulative approach, which relies on a prefix tree representation of the already found intersections. In addition, we present an improved way of implementing the Carpenter algorithm. We demonstrate that on specific data sets, which occur particularly often in the area of gene expression analysis, our implementations significantly outperform enumeration approaches to frequent item set mining.

Categories and Subject Descriptors

I.5.5 [**Pattern Recognition**]: Implementation

Keywords

frequent item set mining; closed item set; intersection

1. INTRODUCTION

It is hardly an exaggeration to say that the popular research area of *data mining* was started by the tasks of frequent item set mining and association rule induction (see Section 2.1). At least, these tasks have a strong and long-standing tradition in data mining and *knowledge discovery in databases* and triggered an abundance of publications in data mining conferences and journals. Research efforts devoted to these tasks have led to a variety of sophisticated and efficient algorithms to find frequent item sets. Among the best-known approaches are Apriori [2, 1], Eclat [22] and FP-growth [11].

Most of these approaches enumerate candidate item sets, determine their support, and prune candidates that fail to reach the user-specified minimum support. A brief abstract description of a general search scheme, which can also be interpreted as a depth-first search in the subset lattice of the item sets, is provided in Section 2.2. In this view, enumeration approaches work "top-down", since they start at the one-element item sets and work their way downward by extending found frequent item sets by new items.

An alternative algorithmic scheme, which has been much less researched, intersects (subsets of) the given transactions. To the best of our knowledge, only two main variants of this approach have been studied up to now: [14] proposed a cumulative scheme, in which new transactions are intersected with a repository of already found closed item sets, and [15] proposed the Carpenter algorithm (and [16] its closely related variant Cobbler), which works by enumerating and intersecting candidate transaction sets. These approaches can be seen as working "bottom-up", because they start with large item sets, namely the transactions, which are reduced to smaller sets by intersecting them with other transactions.

The main reason why the intersection approach is less researched is that it is often not competitive with the item set enumeration approaches, at least on standard benchmark data sets. However, standard benchmark data sets contain comparatively few items (a few hundred), and very many transactions (tens or even hundreds of thousands). Naturally, if there are few items, there are (relatively) few candidate item sets to enumerate and thus the search space of the enumeration approaches is of manageable size. In contrast to this, the more transactions there are, the more work an intersection approach has to do, especially, since it is not linear in the number of transactions like the support computation of the item set enumeration approaches.

EDBT 2011, March 22–24, 2011, Uppsala, Sweden.

However, not all data sets, to which frequent item set mining and association rule induction can profitably be applied, have this structure. As we discuss in Section 4, in gene expression analysis one often meets data sets with very many items (several thousand to tens of thousands), but fairly few transactions (several dozens to hundreds). On these data sets item set enumeration approaches are likely to struggle, because of the huge search space, while the intersection methods can be fast, because there are only few transactions to intersect. Indeed, as we demonstrate on such data, our implementations of the intersection methods can significantly outperform the item set enumeration approaches.

The rest of this paper is structured as follows: in Section 2 we review the basics of frequent item set mining, provide a brief, abstract description of the item set enumeration approach, and define the important concept of a closed (frequent) item set. We exploit an alternative characterization of closed item sets to establish the intersection approach, and review the Galois connection by which it can be formally justified. In Section 3 we discuss our improvements of the Carpenter algorithm as well as the basic cumulative / repository-based algorithm and our prefix tree implementation of it. Section 4 describes the application area of gene expression analysis, the data sets we used in our experiments, and the results we obtained on them. Finally, in Section 6, we draw conclusions from our discussion.

2. FREQUENT ITEM SET MINING

Frequent item set mining is a data analysis method that was originally developed for market basket analysis and that aims at finding regularities in the shopping behavior of the customers of supermarkets, mail-order companies and online shops. In particular, it tries to identify sets of products that are frequently bought together. Once identified, such sets of associated products may be used to optimize the organization of the offered products on the shelves of a supermarket or the pages of a mail-order catalog or web shop, or can give hints which products may conveniently be bundled or may be suggested to a new customer.

2.1 Basic Notions and Notation

Formally, the task of frequent item set mining can be described as follows: we are given a set B of *items*, called the *item base*, and a database T of *transactions*. Each item represents a product, and the item base represents the set of all products offered by a store. The term *item set* refers to any subset of the item base B. Each transaction is an item set and represents a set of products that has been bought by an actual customer. Since two or even more customers may have bought the exact same set of products, the total of all transactions must be represented as a vector, a bag or a multiset, since in a simple set each transaction could occur at most once. (Alternatively, each transaction may be enhanced by a unique *transaction identifier*, and these enhanced transactions may then be combined in a simple set.) Note that the item base B is often not given explicitly, but implicitly as the union of all transactions.

Let $T = (t_1, \ldots, t_n)$ be a transaction database over an item base B. The *cover* $K_T(I)$ of an item set $I \subseteq B$ w.r.t. this database is set of indices of transactions that contain it, that is, $K_T(I) = \{k \in \{1, \ldots, n\} \mid I \subseteq t_k\}$. The *support* $s_T(I)$ of an item set $I \subseteq B$ is the size of its cover: $s_T(I) = |K_T(I)|$, that is, the number of transactions in the database T it is contained in. Given a user-specified *minimum support* $s_{\min} \in \mathbb{N}$, an item set I is called *frequent* in T iff $s_T(I) \geq s_{\min}$. The goal of frequent item set mining is to identify all item sets $I \subseteq B$ that are frequent in a given transaction database T. Note that the task of frequent item set mining may also be defined with a *relative* minimum support (fraction or percentage of the transactions in T). This alternative definition is obviously equivalent.

2.2 Item Set Enumeration Algorithms

A standard approach to find all frequent item sets w.r.t. a given database T and support threshold $s_{\min}$, which is adopted by basically all frequent item set mining algorithms (except those of the Apriori family), is a *depth-first search* in the subset lattice of the item base B. This approach can be interpreted as a simple *divide-and-conquer* scheme. For some chosen item i, the problem to find all frequent item sets is split into two subproblems: (1) find all frequent item sets containing the item i and (2) find all frequent item sets *not* containing the item i. Each subproblem is then further divided based on another item j: find all frequent item sets containing (1.1) both items i and j, (1.2) item i, but not j, (2.1) item j, but not i, (2.2) neither item i nor j, and so on.

All subproblems that occur in this divide-and-conquer recursion can be defined by a *conditional transaction database* and a *prefix*. The prefix is a set of items that has to be added to all frequent item sets that are discovered in the conditional database. Formally, all subproblems are tuples $S = (C, P)$, where C is a conditional transaction database and $P \subseteq B$ is a prefix. The initial problem, with which the recursion is started, is $S = (T, \emptyset)$, where T is the transaction database to mine and the prefix is empty.

A subproblem $S_0 = (C_0, P_0)$ is processed as follows: Choose an item $i \in B_0$, where B_0 is the set of items occurring in C_0. This choice is, in principle, arbitrary, but usually follows some predefined order of the items. If $s_{C_0}(i) \geq s_{\min}$, then report the item set $P_0 \cup \{i\}$ as frequent with the support $s_{C_0}(i)$, and form the subproblem $S_1 = (C_1, P_1)$ with $P_1 = P_0 \cup \{i\}$. The conditional transaction database C_1 comprises all transactions in C_0 that contain the item i, but with the item i removed. This also implies that transactions that contain no other item than i are entirely removed: no empty transactions are ever kept. If C_1 contains at least $s_{\min}$ transactions, process S_1 recursively. In any case (that is, regardless of whether $s_{C_0}(i) \geq s_{\min}$ or not), form the subproblem $S_2 = (C_2, P_2)$, where $P_2 = P_0$ and the conditional transaction database C_2 comprises *all* transactions in C_0 (regardless of whether they contain the item i or not), but again with the item i removed. If C_2 contains at least $s_{\min}$ transactions, process S_2 recursively.

Eclat, FP-growth, and several other frequent item set mining algorithms (see [7, 8]) rely on this basic scheme, but differ in how they represent the conditional databases. The main approaches are horizontal and vertical representations. In a *horizontal representation*, the database is stored as a list (or array) of transactions, each of which is a list (or array) of the items contained in it. In a *vertical representation*, a database is represented by first referring with a list (or array) to the different items. For each item a list (or array) of identifiers is stored, which indicate the transactions that contain the item. However, this distinction is not pure, since there are many algorithms that use a combination of the two forms of representing a database. For example, while Eclat [22] uses a purely vertical representation and SaM (Split and Merge) [3] uses a purely horizontal representation, FP-growth [11] combines in its FP-tree structure a (compressed) horizontal representation (prefix tree of transactions) and a vertical representation (links between tree branches).

The basic recursive processing scheme described above can easily be improved with so-called *perfect extension pruning*, which relies on the following simple idea: given an item set I, an item $i \notin I$ is called a *perfect extension* of I, iff I and $I \cup \{i\}$ have the same support, that is, if i is contained in all transactions containing I. Perfect extensions have the following properties: (1) if the item i is a perfect extension of an item set I, then it is also a perfect extension of any item set $J \supseteq I$ as long as $i \notin J$ and (2) if I is a frequent item set and K is the set of all perfect extensions of I, then all sets $I \cup J$ with $J \in 2^K$ (where 2^K denotes the power set of K) are also frequent and have the same support as I.

These properties can be exploited by collecting in the recursion not only prefix items, but also, in a third element of a subproblem description, perfect extension items. Once identified, perfect extension items are no longer processed in the recursion, but are only used to generate all supersets of the prefix that have the same support. Depending on the data set, this can lead to a considerable speed-up. It should be clear that this optimization can, in principle, be applied in all frequent item set mining algorithms that work according to the described divide-and-conquer scheme.

2.3 Types of Frequent Item Sets

One of the first observations one makes when mining frequent item sets is that the output is often huge—it may even exceed the size of the transaction database to mine. As a consequence, there are several approaches that try to reduce the output, if possible without any loss of information. The most basic of these approaches is to restrict the output to so-called *closed* or *maximal* frequent item sets. A frequent item set is called *closed* if there does not exist a superset that has the same support, or formally:

$$I \subseteq B \text{ is closed} \quad \Leftrightarrow \quad s_T(I) \geq s_{\min} \wedge \\ \forall i \in B - I: \; s_T(I \cup \{i\}) < s_T(I).$$

A frequent item set is called *maximal* if there does not exist any superset that is frequent, or formally:

$$I \subseteq B \text{ is maximal} \quad \Leftrightarrow \quad s_T(I) \geq s_{\min} \wedge \\ \forall i \in B - I: \; s_T(I \cup \{i\}) < s_{\min}.$$

Restricting the output of a frequent item set mining algorithm to only the closed or even only the maximal frequent item sets can sometimes reduce it by orders of magnitude. However, little information is lost: From the set of all maximal frequent item sets the set of all frequent item sets can be reconstructed, since any frequent item set has at least one maximal superset. Therefore the union of all subsets of maximal item sets is the set of all frequent item sets.

Closed frequent item sets even preserve knowledge of the support values. The reason is that each frequent item set has a uniquely determined closed superset with the same support. Hence the support of a frequent item set that is not closed can be computed as the maximum of the support values of all closed frequent item sets that contain it (the maximum has to be used, because no superset can have a greater support—the so-called *apriori property*). As a consequence, closed frequent item sets are the most popular form of compressing the result of frequent item set mining.

Note that closed item sets are closely related to perfect extensions: an item set is closed if it does not have a perfect extension. However, using perfect extension pruning does not mean that the output is restricted to closed item sets, because in the search not all possible extension items are considered (conditional databases do not contain all items).

2.4 Characterizing Closed Item Sets

In the preceding section we characterized closed frequent item sets based on their support and the support of their supersets. With the help of the notion of the *cover* $K_T(I)$ of an item set I (as defined above) we can define

$$I \subseteq B \text{ is closed} \quad \Leftrightarrow \quad |K_T(I)| \geq s_{\min} \; \wedge \; I = \bigcap_{k \in K_T(I)} t_k.$$

That is, an item set is closed if it is equal to the intersection of all transactions that contain it. This definition is obviously equivalent to the one given in Section 2.3: if an item set is a proper subset of the intersection of the transactions it is contained in, there exists a superset (especially the intersection of the transactions itself) that has the same cover and thus the same support. If, however, an item set is equal to the intersection of the transactions containing it, adding any item will remove at least one transaction from its cover and will thus reduce the item set support.

This characterization allows us to find closed item sets by forming, for a minimum support $s_{\min}$, all intersections of k transactions, $k \in \{s_{\min}, \ldots, n\}$ and removing duplicates. Although implementing this procedure directly is prohibitively costly, it provides the basis for intersection approaches.

2.5 Galois Connection

Intersection approaches to find closed frequent item sets can nicely be justified in a formal way by analyzing the Galois connection between the set of all possible item sets 2^B (the power set of the item base B) and the set of all possible sets of transaction indices $2^{\{1,\ldots,n\}}$ (where n is the number of transactions), as it was emphasized and explored in detail in [17]. Consider the two functions

$$\begin{array}{llll} f: 2^B & \to 2^{\{1,\ldots,n\}}, & I \mapsto K_T(I) & \text{and} \\ g: 2^{\{1,\ldots,n\}} & \to 2^B, & K \mapsto \bigcap_{k \in K} t_k. & \end{array}$$

It is easy to show [17] that this function pair is a Galois connection of 2^B and $2^{\{1,\ldots,n\}}$. As a consequence, the compound function $f \circ g: 2^B \to 2^B, I \mapsto \bigcap_{k \in K_T(I)} t_k$ is a closure operator. The closed frequent item sets are then simply the frequent item sets that are closed w.r.t. this closure operator.

More interesting, however, is the fact that with f and g being a Galois connection, the other compound function, namely $g \circ f : 2^{\{1,\ldots,n\}} \to 2^{\{1,\ldots,n\}}$, $K \mapsto K_T(\bigcap_{k \in K} t_k) = \{j \mid \bigcap_{k \in K} t_k \subseteq t_j\}$ is also a closure operator. Thus, if both sets (that is, 2^B and $2^{\{1,\ldots,n\}}$) are restricted to the closed elements w.r.t. the two closure operators, we arrive at a bijective mapping. That is, if $X = \{I \subseteq B \mid (f \circ g)(I) = I\}$ and $Y = \{K \subseteq \{1,\ldots,n\} \mid (g \circ f)(K) = K\}$, then $f|_X$ is a bijection from X to Y. As a consequence, we can find the closed frequent item sets by finding the closed sets of transaction indices that have at least the size $s_{\min}$.

3. INTERSECTING TRANSACTIONS

We discuss two ways of implementing the intersection approach: enumerating transaction sets as it is done in the Carpenter algorithm [15] and a cumulative scheme [14]. For both schemes we present improved implementations.

3.1 Enumerating Transaction Sets

The Carpenter algorithm [15] implements the intersection approach by enumerating sets of transactions (or, equivalently, sets of transaction indices) and intersecting them. This is done with basically the same divide-and-conquer scheme as for the item set enumeration approaches, only that it is applied to transactions (that is, items and transactions exchange their meaning, cf. [17]). Technically, the task to enumerate all transaction index sets is split into two sub-tasks: (1) enumerate all transaction index sets that contain the index 1 and (2) enumerate all transaction index sets that do not contain the index 1. These sub-tasks are then further divided w.r.t. the transaction index 2: enumerate all transaction index sets containing (1.1) both indices 1 and 2, (1.2) index 1, but not index 2, (2.1) index 2, but not index 1, (2.2) neither index 1 nor index 2, and so on.

Formally, all subproblems occurring in the recursion can be described by triples $S = (I, K, \ell)$. $K \subseteq \{1,\ldots,n\}$ is a set of transaction indices, $I = \bigcap_{k \in K} t_k$, that is, I is the item set that results from intersecting the transactions referred to by K, and ℓ is a transaction index, namely the index of the next transaction to consider. The initial problem, with which the recursion is started, is $S = (B, \emptyset, 1)$, where B is the item base and no transactions have been intersected yet.

A subproblem $S_0 = (I_0, K_0, \ell_0)$ is processed as follows: form the intersection $I_1 = I_0 \cap t_{\ell_0}$. If $I_1 = \emptyset$, do nothing (return from recursion). If $|K_0| + 1 \geq s_{\min}$, and there is no transaction t_j with $j \in \{1,\ldots,n\} - K_0$ such that $I_1 \subseteq t_j$, report I_1 with support $s_T(I_1) = |K_0| + 1$. If $\ell_0 < n$, then form the subproblems $S_1 = (I_1, K_1, \ell_1)$ with $K_1 = K_0 \cup \{\ell_0\}$ and $\ell_1 = \ell_0 + 1$ and $S_2 = (I_2, K_2, \ell_2)$ with $I_2 = I_0$, $K_2 = K_0$ and $\ell_2 = \ell_0 + 1$ and process them recursively.

3.1.1 List-based Implementation

From an implementation point of view the core issues of the Carpenter algorithm consist in quickly finding the intersection of the current item set with the next transaction, and to be able to test quickly whether there is a transaction (other than those that have been intersected) that contains the current item set. For the former, the implementation described in [15] uses a vertical transaction database representation. That is, for each item an array is created that lists the indices of those transactions that contain the item. A collection of these arrays is used to represent the current item set I. The intersection is carried out by collecting all arrays that contain the next transaction index ℓ_0. In a programming language that supports pointer arithmetic (like C) this can be done very efficiently by keeping track of the next unprocessed transaction index for each item.

The check whether there exists a transaction that has not been used to form the intersection and which contains the current item set is more complicated. This check is split into the checks (1) whether there exists a transaction t_j with $j > \ell_0$ such that $I_1 \subseteq t_j$ and (2) whether there exists a transaction t_j with $j \in \{1,\ldots,\ell_0 - 1\} - K_0$ such that $I_1 \subseteq t_j$. The first check is actually easy, since the transactions t_j with $j > \ell_0$ are considered in the recursive processing, which can return whether such a transaction exists. The problematic part is (2), which is solved in the implementation described in [15] by maintaining a repository of already found closed frequent item sets and always solving subproblem S_1 before S_2 (including a transaction before excluding it). In this way one can simply check (2) by determining whether I_1 is already in the repository. If there is a transaction t_j with $j \in \{1,\ldots,\ell_0 - 1\} - K_0$ and $I_1 \subseteq t_j$, then I_1 is already in the repository, since $K_0 \cup \{j\}$ is considered before K_0.

In order to make the lookup in the repository efficient, it is laid out as a prefix tree, with its top level structured as a flat array of all items and its deeper levels consisting of nodes linking to their first child and their right sibling. Especially the flat structure of the top level is important, because the data sets Carpenter is designed for have many items. Since the top level is likely to be almost fully populated, a flat array avoids traversing a long sibling list. Deeper levels, however, can be expected to be sparser, and consequently we did not observe significant improvements from laying out deeper levels as flat arrays or search trees for sibling nodes.

An important optimization of the above scheme is the analog of perfect extension pruning in the item set enumeration scheme [15]: if $I_1 = I_0$, it is not necessary to solve the second subproblem, because it cannot produce any output: any item set considered in the solution of S_2 can be intersected with t_{ℓ_0} without changing the item set, so the test for a containing transaction must fail. In our own implementation, we also added the following optimization: since we know from the arrays of transaction indices per item how many of the transactions t_j with $j \geq \ell_0$ contain a given item, we can immediately exclude any item i from the intersection for which $|K_0| + |\{j \mid \ell_0 \leq j \leq n \wedge i \in t_j\}| < s_{\min}$. The reason is that in this case no item set that will be constructed in the recursion and which contains i can reach the minimum support. This optimization leads to a considerable speed-up.

3.1.2 Table-based Implementation

An implementation based on lists of transactions indices per item has the disadvantage that collecting the reduced lists for an intersection and updating the number of remaining transactions containing an item not only requires time, but also memory. In order to reduce these costs, we designed a table- or matrix-based implementation, which represents the data set by a $n \times |B|$ matrix M as follows:

$$m_{ki} = \begin{cases} 0, & \text{if item } i \notin t_k, \\ |\{j \mid k \leq j \leq n \wedge i \in t_j\}|, & \text{otherwise.} \end{cases}$$

transaction database						matrix representation					
							a	b	c	d	e
t_1	a	b	c			t_1	4	5	5	0	0
t_2	a	d	e			t_2	3	0	0	6	3
t_3	b	c	d			t_3	0	4	4	5	0
t_4	a	b	c	d		t_4	2	3	3	4	0
t_5	b	c				t_5	0	2	2	0	0
t_6	a	b	d			t_6	1	1	0	3	0
t_7	d	e				t_7	0	0	0	2	2
t_8	c	d	e			t_8	0	0	1	1	1

Table 1: Matrix representation of a transaction database for the improved Carpenter variant.

As an example, Table 1 shows a transaction database with 5 items and 8 transactions and its matrix representation.

Although such a matrix representation needs more memory than the lists of transaction indices (since the zero entries in the matrix need not be represented in the list representation), it saves memory in the recursion, since only the items in the current intersection need to be represented and no references to the current positions in the transaction index lists are needed. In addition, the intersection can be formed by indexing the matrix with ℓ_0 and item identifiers of the current set, which is faster than traversing the lists and checking whether they contain ℓ_0 as the next transaction index.

To check for the existence of transactions that have not been used for the intersections that resulted in the current item set, but contain it, the same repository technique is used that was outlined in the preceding section.

3.2 Cumulative Intersections

An alternative to the transaction set enumeration approach is a scheme that maintains a repository of all closed item sets, which is updated by intersecting it with the next transaction. To justify this approach formally, we consider the set of all closed frequent item sets for $s_{\min} = 1$, that is, the set

$$\mathcal{C}(T) = \Big\{ I \subseteq B \;\Big|\; \exists S \subseteq T : S \neq \emptyset \wedge I = \bigcap_{t \in S} t \Big\}.$$

As was already noted in [14] and exploited for the implementation accompanying that paper, the set $\mathcal{C}(T)$ satisfies the following simple recursive relation:

$$\begin{aligned} \mathcal{C}(\emptyset) &= \emptyset, \\ \mathcal{C}(T \cup \{t\}) &= \mathcal{C}(T) \cup \{t\} \cup \{I \mid \exists s \in \mathcal{C}(T) : I = s \cap t\}. \end{aligned} \tag{1}$$

As a consequence, we can start the procedure with an empty set of closed item sets and then process the transactions one by one, each time updating the set of closed item sets by adding the new transaction itself and the additional closed item sets that result from intersecting it with the already known closed item sets. In addition, the support of already known closed item sets may have to be updated.

Furthermore, we have to consider that in practice we will not work with a minimum support $s_{\min} = 1$ as it underlies $\mathcal{C}(T)$. Unfortunately, removing intersections early, because they do not reach the user-specified minimum support is difficult: in principle, enough of the transactions to be processed in the future could contain the item set under consideration in order to make it frequent. This is a fundamental problem of the intersection approach. We improve on it by an improved version of the method that was already outlined in [14], which is analogous to the item elimination scheme described in Section 3.1.1: in an initial pass through the transaction database we determine the frequency of the individual items. (This is done by virtually all frequent item set mining algorithms anyway in order to remove infrequent items and to determine a good order in which to process the items.) The obtained counters are updated with each processed transaction, so that they always represent the number of occurrences of each item in the unprocessed transactions (cf. also the matrix representation shown in Figure 1).

Based on these counters, we can apply the following pruning scheme: suppose that after having processed k of a total of n transactions the support of a closed item set I is $s_{T_k}(I) = x$ and let y be the minimum of the counter values for the items contained in I. If $x + y < s_{\min}$, then I can be discarded, because it cannot reach the minimum support. The reason is that it cannot occur more than y times in the remaining transactions, because one of its items does not occur more often.

We have to be a bit careful, though, because I may be needed in order to construct certain subsets of it, namely those that result from intersections of it with new transactions. These subsets may still be frequent, even though I is not. As a consequence, we do not simply remove the item set, but selectively remove items from it, which do not occur frequently enough in the remaining transactions (in exactly the same way in which we eliminated items in our improved implementation of the Carpenter algorithm). Although in this way we may construct non-closed item sets, we do not create problems for the final output: either the reduced item set also occurs as the intersection of enough transactions and thus is closed, or it will not reach the minimum support threshold.

```
typedef struct _node {          // a prefix tree node
  int step;                     // most recent update step
  int item;                     // assoc. item (last in set)
  int supp;                     // support of item set
  struct _node *sibling;        // successor in sibling list
  struct _node *children;       // list of child nodes
} NODE;
```

Figure 1: Structure of the prefix tree nodes.

3.3 Prefix Tree Implementation

The core problem of implementing the scheme outlined in the preceding section is to find a data structure for storing the closed item sets that allows us to quickly compute the intersections of these sets with a new transaction and to merge the result with them. To achieve this, we rely on a prefix tree, each node of which represents an item set and is structured as shown in Figure 1: the field `children` holds the head of the list of child nodes, which are linked by the field `sibling`, which thus implements a sibling list. The item set that is represented by a node consists of the `item` stored in it plus the items in the nodes on the path from the root.

```
void isect (NODE* node, NODE **ins)
{                              // intersect with transaction
  int i;                       // buffer for current item
  NODE *d;                     // to allocate new nodes
  while (node) {               // traverse the sibling list
    i = node→item;             // get the current item
    if (trans[i]) {            // if item is in intersection
      while ((d = *ins) && (d→item > i))
        ins = &d→sibling;      // find the insertion position
      if   (d                  // if an intersection node with
      && (d→item == i)){       // the item already exists
        if (d→step ≥ step) d→supp−−;
        if (d→supp < node→supp)
          d→supp = node→supp;
        d→supp++;              // update intersection support
        d→step = step; }       // and set current update step
      else {                   // if there is no corresp. node
        d = malloc(sizeof(NODE));
        d→step = step;         // create a new node and
        d→item = i;            // set item and support
        d→supp = node→supp+1;
        d→sibling = *ins; *ins = d;
        d→children = NULL;
      }                        // insert node into the tree
      if (i ≤ imin) return;    // if beyond last item, abort
      isect(node→children, &d→children); }
    else {                     // if item is not in intersection
      if (i ≤ imin) return;    // if beyond last item, abort
      isect(node→children, ins);
    }                          // intersect with subtree
    node = node→sibling;       // go to the next sibling
  } // end of while (node)
} // isect()
```

Figure 2: Code of the intersection function.

The support of this item set is stored in the field `supp`. In order to avoid duplicate representations of the same item set, all children of a node must refer to items with lower codes than the item referred to by the node itself. In addition, we require that the items referred to by the nodes in a sibling list are in descending order w.r.t. their codes.

Finally, the field `step` is used in the intersection process to indicate whether the support of a node has already been updated from an intersection or not. This is important, because multiple different closed item sets can, when intersected with a new transaction, give rise to the same item set. The step counter (which can be seen as an incremental update flag, thus eliminating the need to clear the flag) ensures that the correct support value is computed.

The algorithm works on the prefix tree as follows: at the beginning an empty tree (consisting only of a root node, which does not store any item and thus represents the empty set) is created. Then the transactions are processed one by one. Each new transaction is first simply added to the prefix tree (cf. the recursive relation (1)). Any new nodes created in this step are initialized with a support of zero.

In the next step we compute the intersections of the new transactions with all sets represented by the current prefix tree. This is achieved with a recursive procedure that is basically a selective depth-first traversal of the prefix tree and matches the items in the tree nodes with the items of the transaction. For this purpose the transaction is represented by a global flag array `trans` with one element per item, which is set if the item is contained in the transaction and cleared otherwise. In addition, the item with the lowest code in the current transaction is stored in a global variable `imin`, so that the recursion can avoid branches that cannot produce any intersection results. Finally, there is a global variable `step` indicating the current update step, which is equal to the index of the current transaction.

The code for the intersection step is shown in Figure 2. (This is a considerably simplified version that illustrates the core steps, but does not contain all optimizations we added. For a detailed version refer to the source code, see URL below.) Each call of the function `isect` traverses a sibling list, the start of which is passed as the parameter `node`.

In principle, the recursive procedure could generate all intersections and store them in a separate prefix tree, which is afterwards merged with the original tree. (Actually this was our first implementation.) However, the two steps of forming intersections and merging the result can be combined. This is achieved with the parameter `ins`, which points to the location in the prefix tree that represents the item set that resulted from intersecting the already processed part of the current transaction with the item set represented by `node`. Hence it indicates the location where new nodes may have to be inserted in order to represent new closed item sets that result from intersections of the subtree rooted at `node` with (the unprocessed part of) the current transaction.

In more detail, the procedure works as follows: whenever the item in the current sibling equals an item in the transaction (that is, whenever `trans[i]` is true) and thus the item is in the intersection, it is checked whether the sibling list starting at `*ins` contains a node with this item, because this node has to represent the extended intersection. If such a node exists, its support is updated. For this update the `step` field and variable are vital, because they allow us to determine whether the current transaction was already counted for the support in the node or not. If the value of the step field in the node equals the current step, the node has already been updated and therefore the transaction must be discounted again before taking the maximum. The maximum is taken, because we have to determine the support from the largest set of transactions containing the item set represented by the node. If a node with the intersection item does not exist, a new node is allocated and inserted into the tree at the location indicated by `ins`.

In both cases (with the current item in the transaction or not, that is, with `trans[i]` true or false) the subtree is processed recursively, unless the item of the current node is not larger than the minimum item in the current transaction. In this case no later siblings can produce any intersection, because they and their descendants refer to items with lower codes. (Recall that items are in descending order in a sibling list and from parent to child.) The only difference between the recursive calls is that in the case where the current item is in the intersection, the insertion position is advanced to the children of the current node.

As an illustration of how the prefix tree is built, Figure 3 shows how three simple transactions are processed. In all of these trees arrows pointing downwards are child pointers

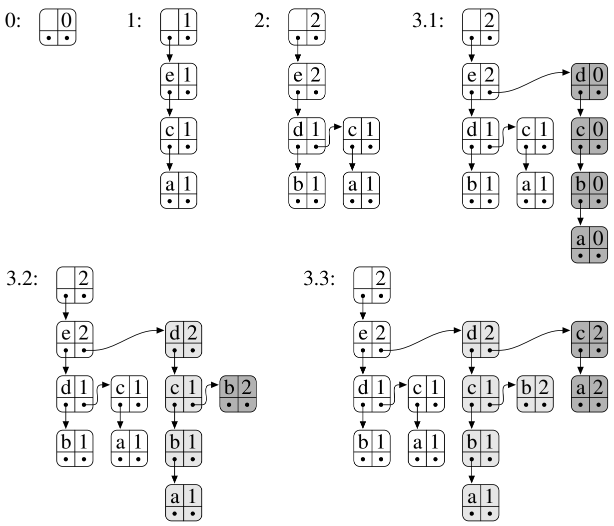

Figure 3: An example of building the prefix tree.

```
void report (NODE *node, int min)
{                             // recursively report item sets
  NODE *c;                    // to traverse the child nodes
  int max = −1;               // maximum support of a child
  for (c = node→children; c; c = child→sibling) {
    if (c→supp < min)         // traverse the child nodes,
      continue;               // but skip infrequent item sets
    if (c→supp > max)         // find maximum child support
      max = c→supp;           // for the closedness check
    report(c, min);           // recursively report item sets
  }
  if (node→supp > max) // if no child has same support
    ⟨report the item set represented by the path to node⟩
} // report()
```

Figure 4: Code of the recursive reporting.

(field `children` in the node structure) and arrows pointing to the right are sibling pointers (field `sibling` in the node structure). Initially, the prefix tree is empty (step 0, at the top left). In step 1 the transaction $\{e, c, a\}$ is added to the tree. Since it was empty before, no new intersections have to be taken care of, so no further processing is carried out. In step 2 the transaction $\{e, d, b\}$ is added. This is done in two steps: first the two nodes on the bottom left are added to represent the transaction. Then, in the recursion, it is discovered that the new transaction overlaps with the one that was already present on the item e. As a consequence, the support value of the node that represents the item set $\{e\}$ is incremented to 2. In the third step the transaction $\{d, c, b, a\}$ is processed. Since things are more complicated now, we split the processing into three steps. In step 3.1 the new transaction is added to the tree, with new nodes initialized to support 0. Steps 3.2 and 3.3 show how the prefix tree is extended by the two intersections $\{d, b\} = \{e, d, b\} \cap \{d, c, b, a\}$ and $\{c, a\} = \{e, c, a\} \cap \{d, c, b, a\}$. Note that the increment of the counters in the nodes for the full transaction $\{d, c, b, a\}$ is not shown as a separate step.

Finally, after all transactions have been processed, the closed frequent item sets have to be reported. This is done with the function shown in Figure 4. Note that *not* every node of the prefix tree generates output. In the first place, item sets that do not reach the user-specified minimum support (parameter `min`) must be discarded. In addition, however, we must also check whether a child has the same support. If this is the case, the item set represented by a node is not closed and must not be reported. In order to take care of this, the function first traverses the children and determines their maximum support. Only if the nodes own support exceeds this maximum (and thus we know that the item set represented by it is closed), it is reported.

3.4 Item and Transaction Orders

It is well known from the enumeration approaches to frequent item set mining that the order in which items are processed can have a huge impact on the processing time. Usually it is best to process them in the order of increasing frequency in the transaction database. In the intersection approach similar considerations are in place, because it can have a huge impact on the processing time how the items are coded and in which order the transactions are processed.[1]

By experimenting with our implementation, we found the following: it is usually most efficient to assign the item codes w.r.t. ascending frequency in the database (the rarest item has code 0, the next code 1 etc.) and to process the transactions in the order of increasing size (number of contained items). The order of transactions of the same size seems to have very little influence. We use a lexicographical order of the transactions based on a descending order of items in each transaction. An intuitive explanation why this scheme is fastest is that it manages to have few and small closed item sets and thus small prefix trees at the beginning, so that many transactions can be processed fast. With the reverse processing order for the transactions the prefix tree becomes fairly large already after few transactions, which slows down the processing for all later transactions.

4. GENE EXPRESSION ANALYSIS

DNA microarray technology is a powerful method for monitoring the expression level of complete genomes in a single experiment. In the last few years, this technique has been widely used in several contexts such as tumor profiling, drug discovery or temporal analysis of cell behavior [19]. Due to the widespread use of this high-throughput technique in the study of several biological systems, a large collection of gene expression data sets is available to the scientific community, some of which contain tens or hundreds of different experimental conditions and constitutes reference databases or "compendiums" of gene expression profiles. A key task to derive biological knowledge from gene expression data is to detect the presence of sets of genes that share similar expression patterns and common biological properties, such as function or regulatory mechanisms. Frequent item set mining has proved to be very efficient for the integrative analysis of such data [5, 4]. This methodology is able to integrate different types of data in the same analytic framework to uncover significant associations among gene expression profiles

[1]Note that the order of the items is independent of the order of the codes used to represent them, which we referred to in the preceding section. We may assign the codes so that the items are numbered ascendingly or descendingly w.r.t. their frequency in the transaction database or in any other way.

and also take into account multiple biological properties of the genes based on co-occurrence patterns.

In this study we have used, among others, the data provided in [12] which contains expression profiles for 6316 transcripts corresponding to 300 diverse mutations and chemical treatments in *Saccharomyces cerevisiae* (baker's yeast). This organism it is one of the most intensively studied eukaryotic model organisms in molecular and cell biology for several reasons, including the fact that it shares the complex internal cell structure of plants and animals. We believe this is a good representative example of the types of data and problems that are common in any molecular biology lab.

The original data is composed of an expression matrix where genes correspond to rows and experimental conditions to columns. By creating a transaction database from this format, genes will be considered as transactions while experimental conditions will be considered as items. This allows the extraction of relationships among experimental conditions. In this scenario we have much more transactions than items. However, the matrix may also be transposed to consider genes as items to extract relationships between them. Contrary to the previous case, the number of items in this scenario is much larger than the number of transactions. Due to the dual properties of the analysis, this use case is an excellent test base for the methodology we propose.

To construct the transaction database from this data set we converted the expression matrix into a Boolean matrix. For this purpose, and following the criteria used in other studies (see, for example, [5]), we have used two expression thresholds: genes with log expression values greater than 0.2 were considered as over-expressed and genes with log expression values lower than -0.2 were considered under-expressed. Values between these two ranks were seen as neither expressed nor inhibited.

As a second data set we used the publicly available NCBI60 cancer cell line microarray expression database as reported in [18], which was pre-processed in a similar way. Furthermore, we used the test part of the Thrombin data set that was made publicly available for the KDD Cup 2001. Even though this is not a gene expression data set (each record describes a molecule that binds or does not bind to thrombin by 139,351 binary features), it exhibits similar characteristics. In order to be able to run detailed experiments, we selected the first 64 records as transactions. Finally, we used the transposed version of the well-known BMS-Webview-1 data set, which is a common benchmark for frequent item set mining. It was derived from click streams of the online shop of a leg-care company that no longer exists and has been used in the KDD cup 2000 [13]. We used the transposed form to obtain a data set with many items and few transactions, as this is the type of data set the algorithms presented here are designed for.

5. EXPERIMENTAL RESULTS

We compared our implementation of the intersection approach to frequent item set mining, which we call IsTa (for **I**ntersecting **T**ransactions), to our implementations of the two variants of the Carpenter algorithm as well as to FP-growth (or rather FP-close, since we used the version that finds closed frequent item sets) in the implementation of [9, 10] (version of 2004), which won the FIMI'03 best implementation award, and to LCM3 [20, 21], which won (in version 2) the FIMI'04 best implementation award.[2]

We also tried the implementation of the intersection approach described in [14], but do not report the results here, because the execution times are vastly larger than those of our implementation (often exceeding a factor of 100), which is due to the fact that this implementation does not employ a prefix tree, but a simple flat structure. Furthermore, we tried to compare to the original implementation of the Carpenter algorithm by its designers (see [15]), as it is made available as part of the GEMini (Gene Expression Mining) software package[3] for Weka. However, we met several technical problems in doing so. In the first place, GEMini is available only as compiled code for Windows, as it ships partially as dynamic link libraries (DLLs). In addition, it works only with a very specific version of Weka (namely 3.5.6), while the Carpenter module did not show up in any other Weka version we tried. Finally, the Carpenter program (including Weka) would crash extremely often. For example, it was impossible to mine the baker's yeast data set for any support value less than 23 (regardless of the memory assigned to the Java virtual machine or the Windows version used). However, for those support values we could try without crashing, the GEMini implementation already needed an order of magnitude longer than our implementation. Due to these results and the comparative results reported in [6], we believe that we can also confidently claim that our implementation of the Carpenter algorithm is faster than the RERII and REPT algorithms developed in [6]. We could not compare to these algorithms directly, because we could not get hold of the implementations.

All of our experiments were carried out on a desktop computer with an Intel Core 2 Quad Q9650 processor (3GHz) and 8 GB memory running Ubuntu Linux 10.04, kernel version 2.6.32-24-generic (64 bit). The programs are written either in C or in C++ and were compiled with gcc or g++, respectively, version 4.4.3.

The results are shown in Figures 5 to 8. On the yeast data set (Figure 5) IsTa and Carpenter are slightly slower than the enumeration approaches for minimum support values greater than about 20–24 (where, however, all execution times are far less than a second). For lower support the algorithms heavily diverge in their time consumption: whereas FP-growth and LCM3 exceed 1 minute for a minimum support of 8, and grow even more heavily afterwards, IsTa manages to keep the execution time around 5 seconds. The reason for this is, of course, the fairly small number of transactions (300), combined with a huge number of items (close to 10000, at least at the lowest support values), which clearly favors the intersection approach. It should be noted that the enumeration approaches still outperform the implementation of the intersection approach described in [14], which did not finish in a reasonable time on this data set, even for larger support (and thus we terminated the run).

[2] The source code of these implementations can be downloaded from `http://fimi.cs.helsinki.fi/` and `http://research.nii.ac.jp/~uno/codes.htm`

[3] `http://nusdm.comp.nus.edu.sg/projects.htm`

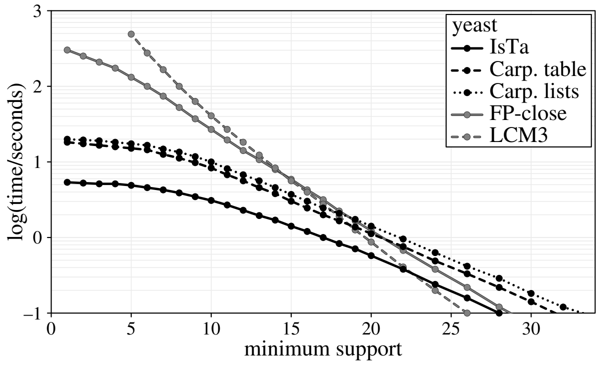

Figure 5: Results on the baker's yeast data.

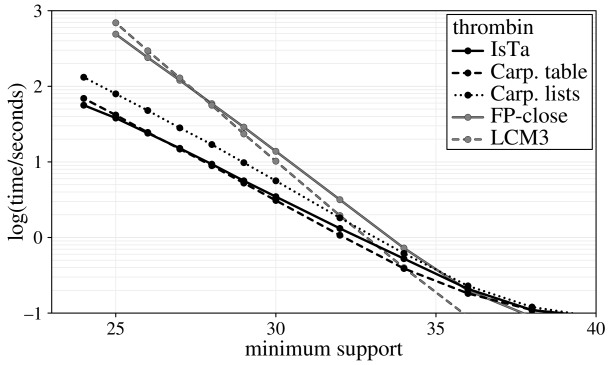

Figure 7: Results on a subset of the thrombin data.

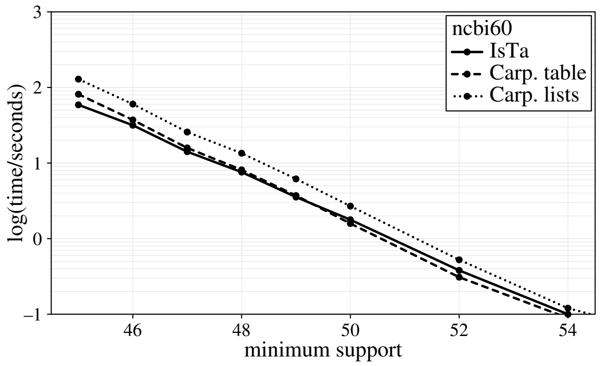

Figure 6: Results on the ncbi60 data.

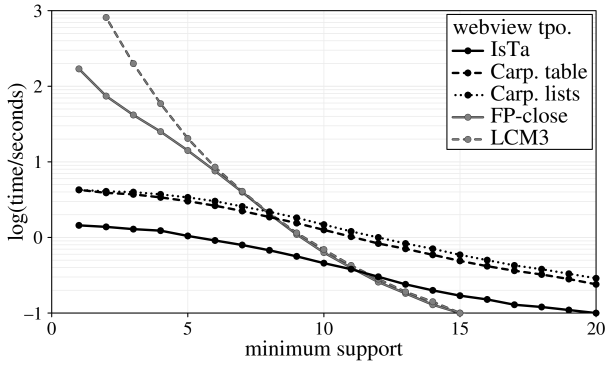

Figure 8: Results on the transposed webview data.

Of the Carpenter variants, the table-based is somewhat better than the list-based, but neither can compete with IsTa.

On the NCBI60 data set (Figure 6) the table-based variant of Carpenter and IsTa perform basically on par until IsTa can gain a small advantage at the lowest support value. The list-based Carpenter version is clearly slower by a constant factor (note the logarithmic scale on the vertical axis). There are no results for FP-growth or LCM3 on this data set, because both programs either crashed with segmentation faults or entered an infinite loop (LCM3 for higher support values).

The Thrombin subset behaves very similarly to the NCBI60 data set, with the table-based variant of Carpenter and IsTa performing basically on par until IsTa achieves a slight advantage at the lowest support value. The list-based variant is again slower by constant factor. LCM3 and FP-growth are competitive only down to a minimum support of 32 to 34, where, however, all execution times are less than a second.

Finally, on the transposed BMS-webview-1 data set the behavior is similar to the yeast data set. The table-based variant of Carpenter is slightly better than the list-based, both of which are clearly outperformed by IsTa. FP-growth and LCM3 are competitive only down to a minimum support of about 11, where all execution times are less than a second.

From the reported experimental results we can confidently infer that the intersection approach is the method of choice for data sets with few transactions and (very) many items, as they commonly occur in gene expression analysis, where a large to huge number of genes (items) is tested in a moderate number of conditions (transactions): both IsTa and Carpenter clearly outperform the item set enumeration approaches, which were represented by particularly fast implementations. While they are on par on two data sets, IsTa clearly outperforms Carpenter on the other two.

6. CONCLUSIONS

In this paper we presented an improved implementation of the cumulative intersection approach for mining closed frequent item sets, which was originally introduced by [14]. By using a prefix tree structure to represent the already found closed item sets, and by devising a recursive procedure that quickly finds all intersections with a new transaction and adds them to the prefix tree, we obtained an implementation that is orders of magnitudes faster than the implementation of [14]. In addition, we presented an improved implementation of the standard list-based scheme of the Carpenter algorithm, which significantly outperforms the version by the original authors. Finally, we suggested a table-based variant of the Carpenter algorithm, which yields even shorter execution times, even though it loses (on two data sets) against IsTa, our implementation of the cumulative intersection approach. By comparing the algorithms to the fastest representatives of the item set enumeration approaches, we showed that on a particular type of data sets, which commonly occur in gene expression analysis, our implementations significantly outperform the fastest enumeration approaches, for lower support values even by huge factors.

Software

The source code of our implementations can be found at
`http://www.borgelt.net/ista.html`
`http://www.borgelt.net/carpenter.html`

Acknowledgments

The work presented here was partially supported by the European Commission under the 7th Framework Program FP7-ICT-2007-C FET-Open, contract no. BISON-211898.

7. REFERENCES

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press / MIT Press, Cambridge, CA, USA, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. on Very Large Databases (VLDB 1994, Santiago de Chile)*, pages 487–499, San Mateo, CA, USA, 1994. Morgan Kaufmann.

[3] C. Borgelt and X. Wang. SaM: A split and merge algorithm for fuzzy frequent item set mining. In *Proc. 13th Int. Fuzzy Systems Association World Congress and 6th Conf. of the European Society for Fuzzy Logic and Technology (IFSA/EUSFLAT'09, Lisbon, Portugal)*, Lisbon, Portugal, 2009. IFSA/EUSFLAT Organization Committee.

[4] P. Carmona-Sáez, M. Chagoyen, A. Rodríguez, O. Trelles, J. M. Carazo, and A. Pascual-Montano. Integrated analysis of gene expression by association rules discovery. *BMC Bioinformatics*, 7:54, 2006.

[5] C. Ceighton and S. Hanash. Mining gene expression databases for association rules. *Bioinformatics*, 19:79–86, 2003.

[6] G. Cong, K.-I. Tan, A. Tung, and F. Pan. Mining frequent closed patterns in microarray data. In *Proc. 4th IEEE International Conference on Data Mining (ICDM 2004, Brighton, UK)*, pages 363–366, Piscataway, NJ, USA, 2004. IEEE Press.

[7] B. Goethals and M. Zaki. Advances in frequent itemset mining implementations: Introduction to FIMI'03. In *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*, Aachen, Germany, 2003. CEUR Workshop Proceedings 90.

[8] B. Goethals and M. Zaki, editors. *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2004, Brighton, UK)*, Aachen, Germany, 2004. CEUR Workshop Proceedings 126.

[9] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*, Aachen, Germany, 2003. CEUR Workshop Proceedings 90.

[10] G. Grahne and J. Zhu. Reducing the main memory consumptions of FPmax* and FPclose. In *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2004, Brighton, UK)*, Aachen, Germany, 2004. CEUR Workshop Proceedings 126.

[11] J. Han, H. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. Conf. on the Management of Data (SIGMOD'00, Dallas, TX)*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[12] T. Hughes, M. Marton, A. Jones, C. Roberts, R. Stoughton, C. Armour, H. Bennett, E. Coffey, H. Dai, Y. He, M. Kidd, A. King, M. Meyer, D. Slade, P. Lum, S. Stepaniants, D. Shoemaker, D. Gachotte, K. Chakraburtty, J. Simon, M. Bard, and S. Friend. Functional discovery via a compendium of expression profiles. *Cell*, 102:109–126, 2000.

[13] R. Kohavi, C. Bradley, B. Frasca, L. Mason, and Z. Zheng. Kdd-cup 2000 organizers' report: Peeling the onion. *SIGKDD Exploration*, 2:86–93, 2000.

[14] T. Mielikäinen. Intersecting data to closed sets with constraints. In *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*, Aachen, Germany, 2003. CEUR Workshop Proceedings 90.

[15] F. Pan, G. Cong, A. Tung, J. Yang, and M. Zaki. Carpenter: Finding closed patterns in long biological datasets. In *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2003, Washington, DC)*, pages 637–642, New York, NY, USA, 2003. ACM Press.

[16] F. Pan, A. Tung, G. Cong, and X. Xu. Cobbler: Combining column and row enumeration for closed pattern discovery. In *Proc. 16th Int. Conf. on Scientific and Statistical Database Management (SSDBM 2004, Santori Island, Greece)*, page 21ff, Piscataway, NJ, USA, 2004. IEEE Press.

[17] F. Rioult, J.-F. Boulicaut, B. Crémilleux, and J. Besson. Using transposition for pattern discovery from microarray data. In *Proc. 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD 2003, San Diego, CA)*, pages 73–79, New York, NY, USA, 2003. ACM Press.

[18] U. Scherf, D. Ross, M. Waltham, L. Smith, J. Lee, L. Tanabe, K. Kohn, W. Reinhold, T. Myers, , D. Andrews, D. Scudiero, M. Eisen, E. Sausville, Y. Pommier, D. Botstein, P. Brown, and J. Weinstein. A gene expression database for the molecular pharmacology of cancer. *Nature Genetics*, pages 236–244, 2000.

[19] R. Stoughton. Applications of DNA microarrays in biology. *Annual Review of Biochemistry*, 2004.

[20] T. Uno, T. Asai, Y. Uchida, and H. Arimura. Lcm: An efficient algorithm for enumerating frequent closed item sets. In *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*, Aachen, Germany, 2003. CEUR Workshop Proceedings 90.

[21] T. Uno, M. Kiyomi, and H. Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Proc. Workshop Frequent Item Set Mining Implementations (FIMI 2004, Brighton, UK)*, Aachen, Germany, 2004. CEUR Workshop Proceedings 126.

[22] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97, Newport Beach, CA)*, pages 283–296, Menlo Park, CA, USA, 1997. AAAI Press.