OSDM 2005

Proceedings of the First International Workshop on Open Source Data Mining

Frequent Pattern Mining Implementations

held in conjunction with
the Eleventh ACM SIGKDD International Conference on
Knowledge Discovery and Data Mining

Edited by:
Bart Goethals
Siegfried Nijssen
Mohammed J. Zaki

OSDM'05, August 21, 2005, Chicago, Illinois, USA

Foreword

Over the past decade tremendous progress has been made in data mining methods like clustering, classification, frequent pattern mining, and so on. Unfortunately, however, the advanced implementations are often not made publicly available, and thus the results cannot be independently verified. We believe that this hampers the rapid advances in the field. With this workshop we intend to promote open source data mining (OSDM) by creating a first meeting place to discuss open source data mining methods.

The first steps towards an open source data mining workshop were set in previous years by the Frequent Itemset Mining Implementations workshops (FIMI), which enjoyed a large popularity. The OSDM workshop is held in the same spirit as these earlier workshops, and, in its first edition, the workshop therefore has a special focus on implementations of frequent pattern mining algorithms. It is our hope that in subsequent years the workshop will also focus on open source implementations for other data mining problems like clustering, classification, outlier detection, and so on.

Frequent pattern mining is a core field of research in data mining encompassing the discovery of patterns such as itemsets, sequences, trees, graphs, and many other structures. Varied approaches to these problems appear in numerous papers across all data mining conferences. Generally speaking, the problem involves the identification of items, products, symptoms, characteristics, and so forth, that often occur together in a given dataset. As a fundamental operation in data mining, algorithms for FPM can be used as a building block for other, more sophisticated data mining processes. During the last decade, a huge number of algorithms have been developed in order to efficiently solve all kinds of FPM problems. A representative set of such algorithms can now be found in these proceedings, including papers about frequent itemset mining, frequent sequence mining and frequent graph mining.

All submissions to this workshop were necessarily accompanied by source code. This source code can also be found on the homepage of the OSDM 2005 workshop:

`http://osdm.ua.ac.be/`

All papers were independently reviewed by the members of the program committee. We wish to thank all members of this committee for their effort.

Bart Goethals
Siegfried Nijssen
Mohammed J. Zaki

Table of Contents

Workshop Organization

Program Co-Chairs:
Bart Goethals, *University of Antwerp, Belgium*
Siegfried Nijssen, *Universiteit Leiden, The Netherlands*
Mohammed J. Zaki, *Rensellaer Polytechnic Institute, NY, USA*

Program Committee:
Charu Aggarwal, *IBM Watson, USA*
Christian Borgelt, *Otto-von-Guericke-University of Magdeburg, Germany*
Mohammad El-Hajj, *University of Alberta, Canada*
Lawrence B. Holder, *University of Texas at Arlington, USA*
Akihiro Inokuchi, *Osaka University and IBM, Japan*
George Karypis, *University of Minnesota, USA*
Michihiro Kuramochi, *University of Minnesota, USA*
Sergei Kuznetsov, *All-Russian Institute for Scientific and Technical Information, Russia*
Sergei Obiedkov, *Russian State University for Humanities*, *Russia*
Jian Pei, *Simon Fraser University, Canada*
Hannu Toivonen, *University of Helsinki, Finland*
Takeaki Uno, *National Institute of Informatics, Japan*

Invited Talk

Finding the Real Patterns

Geoff Webb
Faculty of Information Technology
Monash University
Victoria 3800, Australia

ABSTRACT

Pattern discovery typically explores a massive space of potential patterns to identify those that satisfy some user-specified set of criteria. This process entails a huge risk (in many cases a near certainty) that many patterns will be false discoveries. These are patterns that satisfy the specified criteria with respect to the sample data but do not satisfy those criteria with respect to the population from which those data are drawn. This talk discusses the problem of false discoveries, and presents techniques for avoiding them.

An Implementation of the FP-growth Algorithm

Christian Borgelt

Department of Knowledge Processing and Language Engineering
School of Computer Science, Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany

borgelt@iws.cs.uni-magdeburg.de

ABSTRACT

The FP-growth algorithm is currently one of the fastest approaches to frequent item set mining. In this paper I describe a C implementation of this algorithm, which contains two variants of the core operation of computing a projection of an FP-tree (the fundamental data structure of the FP-growth algorithm). In addition, projected FP-trees are (optionally) pruned by removing items that have become infrequent due to the projection (an approach that has been called FP-Bonsai). I report experimental results comparing this implementation of the FP-growth algorithm with three other frequent item set mining algorithms I implemented (Apriori, Eclat, and Relim).

1. INTRODUCTION

One of the currently fastest and most popular algorithms for frequent item set mining is the FP-growth algorithm [8]. It is based on a prefix tree representation of the given database of transactions (called an FP-tree), which can save considerable amounts of memory for storing the transactions. The basic idea of the FP-growth algorithm can be described as a *recursive elimination* scheme: in a preprocessing step delete all items from the transactions that are not frequent individually, i.e., do not appear in a user-specified minimum number of transactions. Then select all transactions that contain the least frequent item (least frequent among those that are frequent) and delete this item from them. Recurse to process the obtained reduced (also known as *projected*) database, remembering that the item sets found in the recursion share the deleted item as a prefix. On return, remove the processed item also from the database of all transactions and start over, i.e., process the second frequent item etc. In these processing steps the prefix tree, which is enhanced by links between the branches, is exploited to quickly find the transactions containing a given item and also to remove this item from the transactions after it has been processed.

In this paper I describe an efficient C implementation of the FP-growth algorithm. In Section 2 I briefly review how the transaction database is preprocessed in a way that is common to basically all frequent item set mining algorithms. Section 3 explains how the initial FP-tree is built from the (preprocessed) transaction database, yielding the starting point of the algorithm. The main step is described in Section 4, namely how an FP-tree is projected in order to obtain an FP-tree of the (sub-)database containing the transactions with a specific item (though with this item removed). The projection step is the most costly in the algorithm and thus it is important to find an efficient way of executing it. Section 5 considers how a projected FP-tree may be further pruned using a technique that has been called FP-Bonsai [4]. Such pruning can sometimes shrink the FP-tree considerably and thus lead to much faster projections. Finally, in Section 6 I report experiments with my implementation, comparing it with my implementations [5, 6] of the Apriori [1, 2] and Eclat [10] algorithms.

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

2. PREPROCESSING

Similar to several other algorithms for frequent item set mining, like, for example, Apriori or Eclat, FP-growth preprocesses the transaction database as follows: in an initial scan the frequencies of the items (support of single element item sets) are determined. All infrequent items—that is, all items that appear in fewer transactions than a user-specified minimum number—are discarded from the transactions, since, obviously, they can never be part of a frequent item set.

In addition, the items in each transaction are sorted, so that they are in *descending* order w.r.t. their frequency in the database. Although the algorithm does not depend on this specific order, experiments showed that it leads to much shorter execution times than a random order. An *ascending* order leads to a particularly slow operation in my experiments, performing even worse than a random order. (In this respect FP-growth behaves in exactly the opposite way as Apriori, which in my implementation usually runs fastest if items are sorted ascendingly, but in the same way as Eclat, which also profits from items being sorted descendingly.)

This preprocessing is demonstrated in Table 1, which shows an example transaction database on the left. The frequencies of the items in this database, sorted descendingly, are shown in the middle of this table. If we are given a user specified minimal support of 3 transactions, items f and g can be discarded. After doing so and sorting the items in each transaction descendingly w.r.t. their frequencies we obtain the reduced database shown in Table 1 on the right.

a d f	d 8	d a
a c d e	b 7	d c a e
b d	c 5	d b
b c d	a 4	d b c
b c	e 3	b c
a b d	f 2	d b a
b d e	g 1	d b e
b c e g		b c e
c d f		d c
a b d		d b a

Table 1: Transaction database (left), item frequencies (middle), and reduced transaction database with items in transactions sorted descendingly w.r.t. their frequency (right).

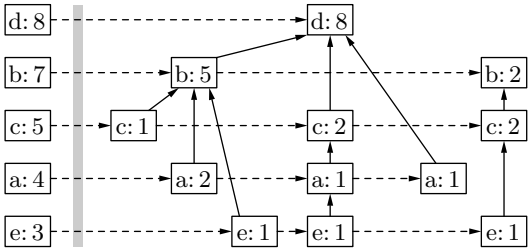


Figure 1: FP-tree for the (reduced) transaction database shown in Table 1.

3. BUILDING THE INITIAL FP-TREE

After all individually infrequent items have been deleted from the transaction database, it is turned into an FP-tree. An FP-tree is basically a prefix tree for the transactions. That is, each path represents a set of transactions that share the same prefix, each node corresponds to one item. In addition, all nodes referring to the same item are linked together in a list, so that all transactions containing a specific item can easily be found and counted by traversing this list. The list can be accessed through a head element, which also states the total number of occurrences of the item in the database. As an example, Figure 1 shows the FP-tree for the (reduced) database shown in Table 1 on the right. The head elements of the item lists are shown to the left of the vertical grey bar, the prefix tree to the right of it.

In my implementation the initial FP-tree is built from a main memory representation of the (preprocessed) transaction database as a simple list of integer arrays. This list is sorted lexicographically (thus respecting the order of the items in the transactions, which reflects their frequency). The sorted list can easily be turned into an FP-tree with a straightforward recursive procedure: at recursion depth k, the k-th item in each transaction is used to split the database into sections, one for each item. For each section a node of the FP-tree is created and labeled with the item corresponding to the section. Each section is then processed recursively, split into subsections, a new layer of nodes (one per subsection) is created etc. Note that in doing so one has to take care that transactions that are only as long as the current recursion depth are handled appropriately, that is, are removed from the section before going into recursion.

Of course, this is not the only way in which the initial FP-tree can be built. At first sight it may seem to be more natural to build it by inserting transaction after transaction into an initially empty FP-tree, creating the necessary nodes for each new transaction. Indeed, such an approach even has the advantage that the transaction database need not be loaded in a simple form (for instance, as a list of integer arrays) into main memory. Since only one transaction is processed at a time, only the FP-tree representation and one new transaction is in main memory. This usually saves space, because an FP-tree is often a much more compact representation of a transaction database.

Nevertheless I decided against such a representation for the following reasons: in order to build a prefix tree by sequentially adding transactions, one needs pointers from parent nodes to child nodes, so that one can descend in the tree according to the items present in the transaction. However, this is highly disadvantageous. As we will see later on, the further processing of an FP-tree, especially the main operation of projecting it, does not need such parent-to-child pointers in my implementation, but rather child-to-parent pointers. Since each node in an FP-tree (with the exception of the roots) has exactly one parent, this, in principle, makes it possible to work with nodes of constant size. If, however, we have to accomodate an array of child pointers per node, the nodes either have variable size or are unnecessarily large (because we have pointers that are not needed), rendering the memory management much less efficient.

It has to be conceded, though, that instead of using an array of child pointers, one may also link all children into a list. This, however, has the severe disadvantage that when inserting transactions into the FP-tree, one such list has to be searched (linearly!) for each item of the transaction in order to find the child to go to—a possibly fairly costly operation.

In contrast to this, first loading the transaction database as a simple list of integer arrays, sorting it, and building the FP-tree with a recursive function (as outlined above), makes it possible to do without parent-to-child pointers entirely. Since the FP-tree is built top down, the parent is already known when the children are created. Thus it can be passed down in the recursion, where the parent pointers of the children are set directly. As a consequence, the nodes of the FP-tree can be kept very small. In my implementation, an FP-tree node contains only fields for (1) an item identifier, (2) a counter, (3) a pointer to the parent node, (4) a pointer to the successor node (referring to the same item) and (5) an auxiliary pointer that is used when projecting the FP-tree (see below). That is, an FP-tree node needs only 20 bytes (on a 32 bit machine).

However, if we used the standard memory management, allocating a block of memory for each node, there would be an additional overhead of 4 to 12 bytes (depending on the memory system implementation) for each node for bookkeeping purposes (for instance, for storing the size of the memory block). In addition, allocating and deallocating a large number of such small memory blocks is usually not very efficient. Therefore I use a specialized memory management in my implementation, which makes it possible to efficiently handle large numbers of equally sized small mem-

ory objects. The idea is to allocate larger arrays (with several thousand elements) of these objects and to organize the elements into a "free" list (i.e., a list of available memory blocks of equal size). With such a system allocating and deallocating FP-tree nodes gets very efficient: the former retrieves (and removes) the first element of the free list, the latter adds the node to deallocate at the beginning of the free list. As experiments showed, introducing this specialized memory management led to a considerable speed-up.

4. PROJECTING AN FP-TREE

The core operation of the FP-growth algorithm is to compute an FP-tree of a projected database, that is, a database of the transactions containing a specific item, with this item removed. This projected database is processed recursively, remembering that the frequent item sets found in the recursion share the removed item as a prefix.

My implementation of the FP-growth algorithm contains two different projection methods, both of which proceed by copying certain nodes of the FP-tree that are identified by the deepest level of the FP-tree, thus producing a kind of "shadow" of it. The copied nodes are then linked and detached from the original FP-tree, yielding an FP-tree of the projected database. Afterwards the deepest level of the original FP-tree, which corresponds to the item on which the projection was based, is removed, and the next higher level is processed in the same way. The two projections methods differ mainly in the order in which they traverse and copy the nodes of the FP-tree (branchwise vs. levelwise).

The first method is illustrated in Figure 2 for the example FP-tree shown in Figure 1. The red arrows show the flow of the processing and the blue "shadow" FP-tree is the created projection. In an outer loop, the lowest level of the FP-tree, that is, the list of nodes corresponding to the projection item, is traversed. For each node of this list, the parent pointers are followed to traverse all ancestors up to the root. Each encountered ancestor is copied and linked from its original (this is what the auxiliary pointer in each node, which was mentioned above, is needed for). During the copying, the parent pointers of the copies are set, the copies are also organized into level lists, and a sum of the counter values in each node is computed in head elements for these lists (these head elements are omitted in Figure 2).

Note that the counters in the copied nodes are determined only from the counters in the nodes on the deepest level, which are propagated upwards, so that each node receives the sum of its children. Note also that due to this we cannot stop following the chain of ancestors at a node that has already been copied, even though it is clear that in this case all ancestors higher up in the FP-tree must already have been copied. The reason is that one has to update the number of transactions in the copies, adding the counter value from the current branch to all copies of the ancestors on the path to the root. This is what the second projection method tries to improve upon.

In a second traversal of the same branches, carried out in exactly the same manner, the copies are detached from their originals (the auxiliary pointers are set to null), which yields the independent projected FP-tree shown in Figure 3. This

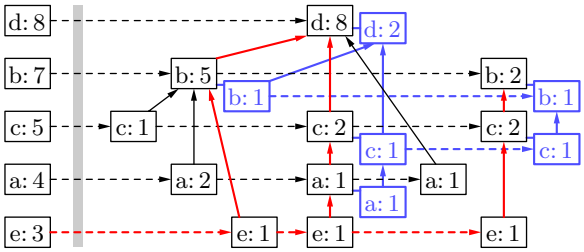


Figure 2: Computing a projection of the database w.r.t. the item e by traversing the lowest level and following all paths to the root.

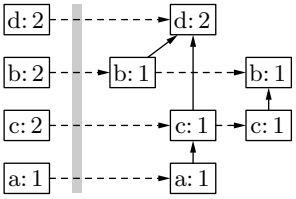


Figure 3: Resulting projected FP-tree after it has been detached from the original FP-tree.

FP-tree is then processed recursively with the prefix e. Note, however, that in this FP-tree all items are infrequent (and thus all item sets containing item e and one other item are infrequent). Hence in this example, no recursive processing would take place. This is, of course, due to the chosen example database and the support threshold.

The second projection method also traverses, in an outer loop, the deepest level of the FP-tree. However, it does not follow the chain of parent pointers up the root, but only copies the parent of each node, not its higher ancestors. In doing so, it also copies the parent pointers of the original FP-nodes, thus making it possible to find the ancestors in later steps. These later steps consist in traversing the levels of the (partially constructed) "shadow" FP-tree (not the levels of the original one!) from bottom to top. On each level the parents of the copied nodes (which are nodes in the original tree) are determined and copied, and the parent pointers of the copies are set. That is, instead of branch by branch, the FP-tree is rather constructed level by level (even though in each step nodes on several levels may be created). The advantage of this method over the one described above is that for branches that share a path close to the root, this common path has to be traversed only once with this method (as the counters for all branches are summed before they are passed to the next higher level). However, the experiments reported below show that the first method is superior in practice. As it seems, the additional effort needed for temporarily setting another parent etc. more than outweighs the advantage of the better combination of the counter values.

5. PRUNING A PROJECTED FP-TREE

After we obtained an FP-tree of a projected database, we may carry out an additional pruning step in order to simplify the tree, thus speeding up projections. I got this idea from [4], which introduces pruning techniques in a slightly different context than pure frequent item set mining (suffice it to say that there are additional constraints). One of these techniques, however, can nevertheless be used here,

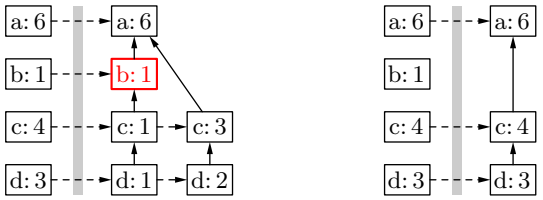


Figure 4: α-pruning of a (projected) FP-tree.

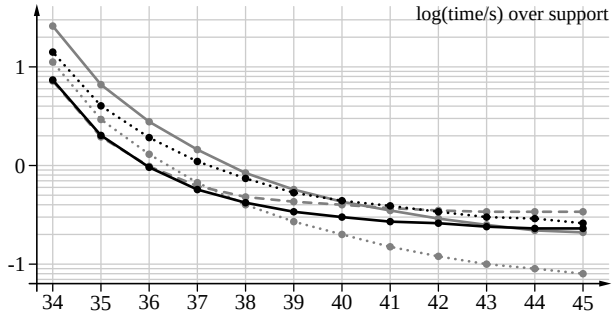


Figure 5: Results on BMS-Webview-1

namely the so-called α-pruning. The idea of this pruning is illustrated with a very simple example in Figure 4. Suppose that the FP-tree shown on the left resulted from a projection and that the minimum support is either 2 or 3. Then item b is infrequent and is not needed in projections. However, it gives rise to a branching in the tree. Hence, by removing it, the tree can be simplified and actually turned into a simple list, as it is shown on the right in Figure 4.

This pruning is achieved by traversing the levels of the FP-tree from top to bottom. The processing starts at the level following the first level that has a non-vanishing support less than the minimum support. (Items having vanishing support can be ignored, because they have no nodes in the FP-tree.) This level and the following ones are traversed and for each node the first ancestor with an item having sufficient support is determined. The parent pointer is then updated to this ancestor, bypassing the nodes corresponding to infrequent items. If by such an operation neighboring nodes receive the same parent, they are merged. They are also merged, if their parents were different originally, but have been merged in a preceding step. As an illustration consider the example Figure 4: after item b is removed, the two nodes for item c can be merged. This has to be recognized in order to merge the two nodes for item d also.

6. EXPERIMENTAL RESULTS

I ran experiments on the same five data sets that I already used in [5, 6], namely BMS-Webview-1 [9], T10I4D100K [11], census, chess, and mushroom [3]. However, I used a different machine and an updated operating system, namely a Pentium 4C 2.6GHz system with 1 GB of main memory running S.u.S.E. Linux 9.3 and gcc version 3.3.5). The results were compared to experiments with my implementations of Apriori, Eclat, and Relim. All experiments were rerun to ensure that the results are comparable.

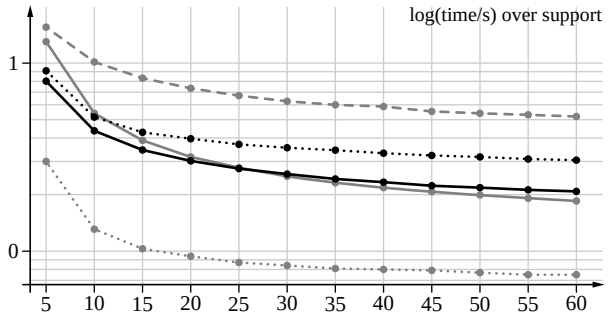


Figure 6: Results on T10I4D100K

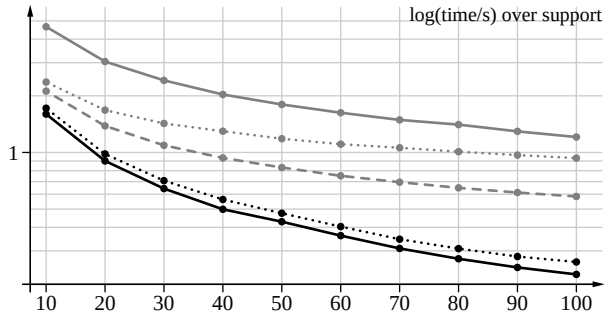


Figure 7: Results on census

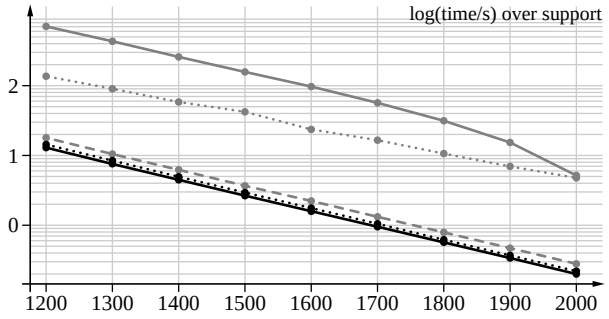


Figure 8: Results on chess

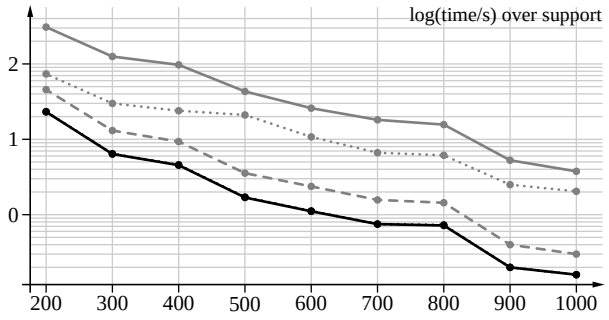


Figure 9: Results on mushroom

Figures 5 to 9 show, each for one of the five data sets, the decimal logarithm of the execution time over different (absolute) minimum support values. The solid black line refers to the implementation of the FP-growth algorithm described here, the dotted black line to the version that uses the alternative projection method. The grey lines represent the corresponding results for Apriori (solid line), Eclat (dashed line), and Relim (dotted line).[1]

Among these implementations, all of which are highly optimized, FP-growth clearly performs best. With the exception of the artificial dataset T10I4D100K, on which it is bet by a considerable margin by Relim, and for higher support values on BMS-Webview-1, where Relim also performs slightly better (presumably, because it does not need to construct a prefix tree), FP-growth is the clear winner. Only on chess, Eclat can come sufficiently close to be called competitive.

The second projection methods for FP-trees (dotted black line) generally fares worse, although there is not much difference between the two methods on chess and mushroom. This is a somewhat surprising result, because there are good reasons to believe that the second projection method may be able to yield better results than the first. I plan to examine this issue in more detail in the future.

7. CONCLUSIONS

In this paper I described an implementation of the FP-growth algorithm, which contains two methods for efficiently projecting an FP-tree—the core operation of the FP-growth algorithm. As the experimental results show, this implementation clearly outperforms Apriori and Eclat, even in highly optimized versions. However, the performance of the two projection methods, especially, why the second is sometimes much slower than the first, needs further investigation.

8. PROGRAM

The implementation of the FP-growth algorithm described in this paper (Windows™ and Linux™ executables as well as the source code, distributed under the LGPL) can be downloaded free of charge at

http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html

At this URL my implementations of Apriori, Eclat, and Relim are also available as well as a graphical user interface (written in Java) for finding association rules with Apriori.

9. REFERENCES

[1] R. Agrawal, T. Imielienski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proc. Conf. on Management of Data*, 207–216. ACM Press, New York, NY, USA 1993

[2] A. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast Discovery of Association Rules. In: [7], 307–328

[3] C.L. Blake and C.J. Merz. *UCI Repository of Machine Learning Databases*. Dept. of Information and Computer Science, University of California at Irvine, CA, USA 1998
http://www.ics.uci.edu/~mlearn/MLRepository.html

[4] F. Bonchi and B. Goethals. FP-Bonsai: the Art of Growing and Pruning Small FP-trees. *Proc. 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04, Sydney, Australia)*, 155–160. Springer-Verlag, Heidelberg, Germany 2004

[5] C. Borgelt. Efficient Implementations of Apriori and Eclat. *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003.
http://www.ceur-ws.org/Vol-90/

[6] C. Borgelt. Recursion Pruning for the Apriori Algorithm. *Proc. 2nd IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Brighton, United Kingdom)*. CEUR Workshop Proceedings 126, Aachen, Germany 2004.
http://www.ceur-ws.org/Vol-126/

[7] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*. AAAI Press / MIT Press, Cambridge, CA, USA 1996

[8] 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)*. ACM Press, New York, NY, USA 2000

[9] R. Kohavi, C.E. Bradley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 Organizers' Report: Peeling the Onion. SIGKDD Exploration 2(2):86–93. 2000.

[10] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97)*, 283–296. AAAI Press, Menlo Park, CA, USA 1997

[11] Synthetic Data Generation Code for Associations and Sequential Patterns. Intelligent Information Systems, IBM Almaden Research Center
http://www.almaden.ibm.com/software/quest/Resources/index.shtml

[1] Relim is described in a sibling paper that has also been submitted to this workshop.

Implementing Leap Traversals of the Itemset Lattice

Mohammad El-Hajj Osmar R. Zaïane
Department of Computing Science, University of Alberta
Edmonton, AB, Canada
{mohammad, zaiane}@cs.ualberta.ca

ABSTRACT

The Leap-Traversal approach consists of traversing the itemset lattice by deciding on carefully selected nodes and avoiding systematic enumeration of candidates. We propose two ways to implement this approach. The first one uses a simple header-less frequent pattern tree and the second one partitions the transaction space using COFI-trees. In this paper we discuss how to avoid nodes in the lattice that would not participate in the answer set and hence drastically reduce the number of candidates to test out. We also study the performance of HFP-Leap and COFI-Leap in comparison with other algorithms.

1. INTRODUCTION

Discovering frequent patterns is a fundamental problem in data mining. The problem is by no means solved and remains a major challenge, in particular for extremely large databases. The idea behind these algorithms is the identification of a relatively small set of candidate patterns, and counting those candidates to keep only the frequent ones. The fundamental difference between the algorithms lies in the strategy to traverse the search space and to prune irrelevant parts. For frequent itemsets, the search space is a lattice connecting all combinations of items between the empty set and the set of all items. Regardless of the pruning techniques, the sole purpose of an algorithm is to reduce the set of enumerated candidates to be counted. The strategies adopted for traversing the lattice are always systematic, either depth-first or breadth-first, traversing the space of itemsets either top-down or bottom-up. Among these four strategies, there is never a clear winner, since each one either favors long or short patterns, thus heavily relying on the transactional database at hand. Our primary motivation here is to find a new traversal method that neither favors nor penalizes a given type of dataset, and at the same time allows the application of lattice pruning for the minimization of candidate generation. Moreover, while discovering frequent patterns can shed light on the content and trends in a transactional database, the discovered patterns can outnumber the transactions themselves, making the analysis of the discovered patterns impractical and even useless. New attempts toward solving such problems are made by finding the set of maximal frequent patterns [4, 1, 10, 6, 11], where a frequent itemset is said to be maximal if there is no other frequent itemset that subsumes it. While we can derive the set of all frequent itemsets directly from the maximal patterns, their support cannot normally be obtained without counting with an additional database scan. Both our Leap-Traversal implementations we present herein traverse the itemset lattice in search of frequent maximals first. Our approaches collect enough information in the process to be able to generate all frequent patterns with their exact support from this set of maximals without having to perform an additional data scan. The data structure used to perform this is presented herein.

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

1.1 Problem Statement

The problem of mining frequent itemsets stems from the problem of mining association rules over market basket analysis as introduced in [2]. The problem consists of finding sets of items (i.e. itemsets) that are sufficiently frequent in a transactional atabase. The data could be retail sales in the form of customer transactions, text documents [9], or even medical images [17]. These frequent itemsets have been shown to be useful for other applications such as recommender systems, diagnosis, decision support, telecommunication, and even supervised classification. They are used in inductive databases [14], query expansion [15], document clustering [5], etc. Formally, as defined in [3], the problem is stated as follows: Let $I = \{i_1, i_2, ...i_m\}$ be a set of literals, called items and m is considered the dimensionality of the problem. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. A transaction T is said to contain X, a set of items in I, if $X \subseteq T$. An itemset X is said to be *frequent* if its *support* s (i.e. ratio of transactions in $\mathcal{D}$ that contain X) is greater than or equal to a given minimum support threshold σ. A frequent itemset $\mathcal{M}$ is considered maximal if there is no other frequent set that is a superset of $\mathcal{M}$.

1.2 Contributions in this paper

In this paper we study the new traversal approach, called leap-traversal, and integrate it in two implementations: one that mines a variation of FP-tree and the other one partitioning using COFI-trees. While these approaches are not particularly competitive with small datasets, we show superior performance of HFP-Leap and COFI-Leap over other

approaches with very large datasets (real and synthetic).

The rest of this paper is organized as follows: Section 2 describes the existing traversal approaches and explains the new leap-traversal method. In the same section, we discuss our pattern intersection strategies using and pruning a tree of intersection possibilities. Since we adopt some data-structures from the literature, we briefly describe in Section 3 the FP-tree, our modified data-structure: the Headerless FP-tree and COFI-trees. The complete algorithms are also presented in Section 3. Section 4 presents the related work in this discipline, while Section 5 depicts the performance evaluation of our new approaches comparing them to the commonly used methods in terms of candidate generation. We also compare them with existing state-of-the-art algorithms to determine results in term of speed, scalability, and memory usage on dense and sparse data.

2. TRAVERSAL APPROACHES

Existing algorithms use either breadth-first-search or depth-first-search strategies to find candidates that will be used to determine the frequent patterns. Breadth-first-search traverses the lattice level-by-level: where it uses frequent patterns at level k to generate candidates at level k+1 before omitting the non-frequent ones and keeping the frequent ones to be used for the level k+2, and so on. This approach usually uses many database scans, and it is not favored while mining databases that are made of long frequent patterns. When traversing the same lattice as in Figure 1 using a breadth-first strategy, frequent 1-itemsets are first generated, then used to generate longer candidates to be tested from size two to above. In our token example, this approach would test 18 candidates to finally discover the 13 frequent ones. Five were unnecessarily tested. On the contrary depth-first-search tries to detect the long patterns at the beginning and only back-tracks to generate the frequent patterns from the long ones that have already been declared as frequent. For longer patterns, depth-first-search indeed outperforms the breadth-first method. But in cases of sparse databases where long candidates do not occur frequently, the depth-first-search is shown to have poor performance. Using the depth-first approach with the same lattice as in Figure 1, 23 candidates are tested, 10 of them unnecessarily.

It is true that many algorithms have been published for enumerating and counting frequent patterns, and yet all algorithms still use one of the two traversal strategies (depth-first vs. breadth-first) in their search. They differ only in their pruning techniques and structures used. No work has been done to find new traversal strategies, such as greedy ones, or best first, etc. We need a new greedy approach that jumps in the lattice searching for the most promising nodes and based on these nodes it would generate the set of all frequent patterns.

2.1 Leap Traversal Approach: Candidate Generation vs. Maximal generation

Most frequent itemset algorithms follow the candidate generation first approach, where candidate items are generated first and only the candidate with support higher than the predefined threshold are declared as frequent while others are omitted. One of the main objectives of the existing algorithms is to reduce the number of candidate patterns. In this work, we propose a new approach to traverse the search space for frequent patterns that is based on finding two things: the set of maximal patterns, and a data-structure that encodes the support of all frequent patterns that can be generated from the set of maximal frequent patterns. Since maximal patterns alone do not suffice to generate the subsequent patterns, the data structure we use keeps enough information about frequencies to counter this deficiency. The basic idea behind the leap traversal approach is that we try to identify the frequent pattern border in the lattice by marking some particular patterns (called later path bases). Simply put, the marked nodes are those representing complete sub-transactions of frequent items. How these are identified and marked will be discussed later. If those marked patterns are frequent, they belong to the border (i.e. they are potential maximal) otherwise their subsets could be frequent, and thus we jump in the lattice to patterns derived from the intersection of infrequent marked patterns in the anticipation of identifying the frequent pattern border. The intersection comes from the following intuition: if a marked node is not a maximal, a subset of it should be maximal. However, rather than testing all its descendants, to reduce the search space we look at descendant's of two non-frequent marked nodes at a time, hence the pattern intersection. The process is repeated until all currently intersected marked patterns are frequent and hence the border is found. Before we explain the Leap-Traversal approach in detail, let us define the Frequent-Path-Bases (FPB). Simply put, these are some particular patterns in the itemset lattice that we mark and use for our traversal. An FPB if frequent could be a maximal. If infrequent, one of its subsets could be frequent and maximal. A frequent-path-base for an item A, called A-Frequent-Path-Base, is a set of frequent items that has the following properties:

1. At maximum one A-FPB can be generated from one transaction.
2. All frequent items in an A-frequent path base have support greater than or equal to the support of A;
3. Each A-FPB represents items that physically occur in the database with item A.
4. Each A-FPB has its branch-support, which represents the number of occurrences for this A-FPB in the database exactly alone (i.e. not as subset of other FPBs). In other words, the branch support of a pattern is the number of transactions that consist of this pattern, not the transactions that include this pattern along with other frequent items. The branch support is always less or equal to the support of a pattern.

As an example for the leap-traversal, assuming we have an oracle to generate for us the Frequent-Pattern-Bases from the token database in Figure 1, the same figure illustrates the process. With the initial FPBs ABC, ABCD, ACDE and DE (given by our hypothetical oracle) we end-up testing only 10 candidates to discover 13 frequent patterns, two were tested unnecessarily (ABCD and ACDE) but 5 patterns were directly identified as frequent without even testing them: AB, AC, AD, BC, CD. From the initially marked nodes, ABC and DE are found frequent, but ABCD and ACDE are not. The intersection of those two nodes yields

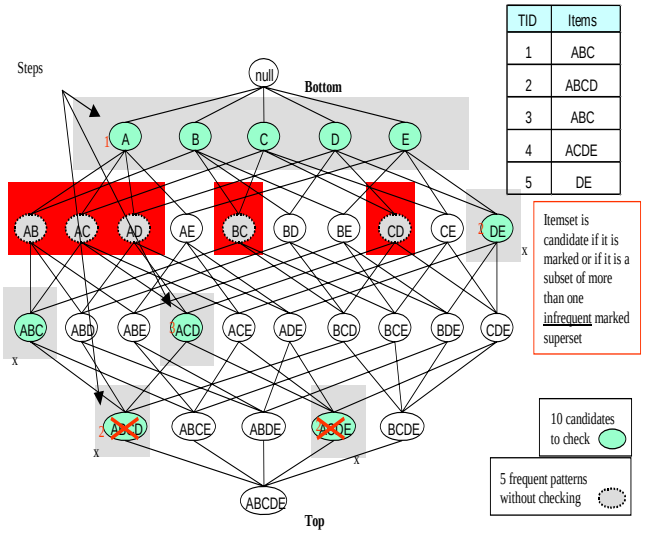


Figure 1: Leap-Traversal

ACD. This newly marked node is found frequent and thus maximal. From the maximals ACD, DE and ABC, we generate all the subsequent patterns, some even without testing (AB, AC, AD, BC and CD). The supports of these patterns are calculated from their superset FPBs. For example, AC has the support of 4 since ABC occurs (alone) twice, ABCD and ACDE occur each alone once.

Frequent pattern bases that have support greater than the predefined support (i.e. frequent patterns) are put aside as they are already known to be frequent and all their subsets are also known to be frequent. Only infrequent ones participate in the leap traversal approach, which consists of intersecting non-frequent FPBs to find a common subset of items shared among the two intersected patterns. The support of this new pattern is found as follows without revisiting the database: the support of Y where $Y = FPB1 \cap FPB2$, is the summation of the branch support of all FPBs that are superset of Y. For example if we have only two frequent path bases ABCD: 1, and ABEF: 1, by intersecting both FPBs we get AB that occurs only once in ABCD and once in ABEF which means it occurs twice in total. By doing so, we do not have to traverse any candidate pattern of size 3 as we were able to jump directly to the first frequent pattern of size 2, which can be declared de-facto as a maximal pattern, hence the name leap-traversal. Consequently, all its subsets are also frequent, which are A and B with support of 2 as they occur only once in each of the frequent path bases.

The Leap-Traversal approach starts by building a lexicographic tree of intersections among the Frequent-Pattern-Bases FPBs. It is a tree of possible intersections between existing FPBs ordered in a lexicographic manner. Assume we have 6 FPBs A, B, C, D, E, and F then Figure 2 depicts the lexicographic tree of intersections between these pattern bases. The size of this tree is relatively big as it has a depth equal to the number of FPBs, which is 6 in our case. It is also unbalanced to the left since intersection is commutative. The number of nodes for this tree equals to 52, as The number of nodes in a lexicographic tree equals to $\sum_{i=1}^{n} \binom{n}{i}$ where n is the number of Frequent Pattern Bases. It is obvious that the more FPBs we have, the larger the tree becomes. Thus, pruning this tree plays an important role for having an efficient algorithm.

Four pruning techniques can be applied to the lexicographic tree of intersections. These pruning strategies can be explained by the following theorems:

Theorem 1: $\forall X, Y \in FPBs$ ordered lexicographically, if $X \cap Y$ is frequent then there is no need to intersect any other elements that have $X \cap Y$., i.e, all children of $X \cap Y$ can be pruned.
Proof: $\forall A, X, Y \in FPBs$, $A \cap X \cap Y \subset X \cap Y$. If $X \cap Y$ is frequent then $A \cap X \cap Y$ is also frequent (apriori property) as a subset of a frequent pattern is also frequent.

Theorem 2: $\forall X, Y, Z, W \in FPBs$ ordered lexicographically, if $X \cap Y = X \cap W$ and $Y \ll W$ (i.e Y is left of W in the lexicographic tree) then there is no need to explore any children of $X \cap Y$. Since Z is left of W (or equal to W) in the lexicographical order, all children of $X \cap Y$ will also be children of $X \cap Z$ or $X \cap W$.
Proof: To prove this theorem we need to show that any children of $X \cap Y$ are repeated under another pattern $X \cap Z$ that always exists. Since $X \cap Y = X \cap W$, then $X \cap Y \cap Z = X \cap Z \cap W$ (intersection is commutative) and $X \cap Z \cap W$ always exists in the lexicographic tree of intersections because of the order. Then, we can prune $X \cap Y$.

Theorem 3: $\forall X, Y, Z \in FPBs$ ordered lexicographically, if $X \cap Y \subset X \cap Z$ then we can ignore the subtree $X \cap Y \cap Z$.
Proof: Assume we have $X, Y, Z \in FPBs$, Since $X \cap Y \subset X \cap Z$ then $X \cap Y \cap Z = X \cap Y$. This means we do not get any additional information by intersecting Z with $X \cap Y$. Thus, the subtree under $X \cap Y$ suffices.

Theorem 4: $\forall X, Y, Z \in FPBs$, if $X \cap Y \supset X \cap Z$ then we can ignore the subtree of $X \cap Z$ as long $X \cap Z$ is not frequent.
Proof: following the proof of Theorem 3 we can conclude that $X \cap Z$ is included in $X \cap Y$.

Lemma 1: At each level of the lexicographic tree of intersections, consider each item as a root of a new subtree:
(A) Intersect the siblings for each node with the root
(B) If a set exists and it is not frequent then we can prune that node.
Proof: Assume we have $X, Y, Z \in FPBs$, if X is a parent node then if $X \cap Y \cap Z$ exists and is not frequent then any superset for this intersected node is also not frequent (apriori property) that is why any intersection of X with any other item is also not frequent.

2.2 Heuristics used for building and traversing the lexicographic tree

Heuristic 1: The lexicographic tree of intersections of FPBs needs to be ordered. Four ways of ordering could be used which are: order by support, support branch, pattern length, and random. Ordering by support yields the best results, as intersecting two patterns with high support in general would generate a pattern with higher support than inter-

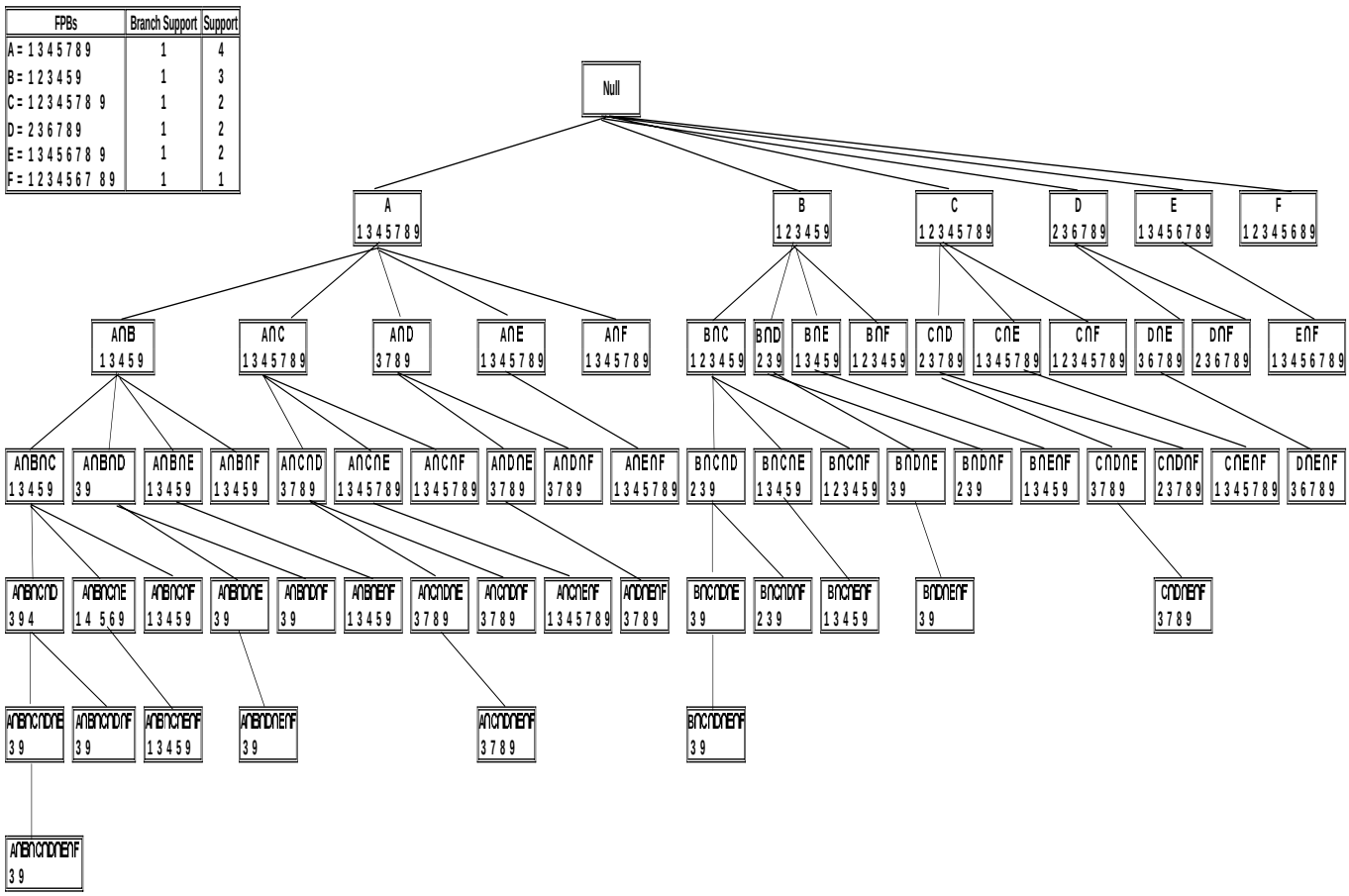


Figure 2: Lexicographic tree of intersections

secting two patterns with lower support. Ordering the tree by assigning the high support nodes at the left increases the probability of finding early frequent patterns in the left and by using Theorem 1, a larger subtree can be pruned.

Heuristic 2: The second heuristic deals with the traversal of the lexicographic tree. The breadth-traversal of the tree is better than the depth-traversal. This observation can be justified by the fact that the goal of the lattice Leap-Traversal approach is to find the maximal patterns, which means finding longer patterns early is the goal of this approach. Thus, by using the breadth-first approach on the intersection tree, we detect and test the longer patterns early before applying too many intersections that usually lead to smaller patterns.

3. TREE STRUCTURES USED

In both algorithms we use the leap-traversal approach. Algorithm 4 that performs the actual leap-traversal to find maximal patterns is called from both HFP-Leap [16] and COFI-Leap. We will first present the idea behind HFP-Leap then show the use of COFI-trees to perform the same type of jumps in the lattice.

The Leap-Traversal approach we discuss consists of two main stages: the construction of a Frequent Pattern tree (HFP-tree); and the actual mining for this data structure by building the tree of Intersected patterns.

3.1 Construction of the Frequent Pattern Tree

The goal of this stage is to build the compact data structure called Frequent Pattern Tree, which is a prefix tree representing sub-transactions pertaining to a given minimum support threshold. This data structure compressing the transactional data was contributed by Han et al. in [12]. The tree structure we use, called HFP-tree is a variation of the original FP-tree. However, we will start introducing the original FP-tree before discussing the differences with our data structure. The construction of the FP-tree is done in two phases, where each phase requires a full I/O scan of the database. A first initial scan of the database identifies the frequent 1-itemsets. The goal is to generate an ordered list of frequent items that would be used when building the tree in the second phase.

After the enumeration of the items appearing in the transactions, infrequent items with a support less than the support threshold are weeded out and the remaining frequent items are sorted by their frequency. This list is organized in a table, called header table, where the items and their respective support are stored along with pointers to the first occurrence of the item in the frequent pattern tree. The actual frequent pattern tree is built in the second phase. This phase requires a second complete I/O scan of the database. For each transaction read, only the set of frequent items present in the header table is collected and sorted in descending order according to their frequency. These sorted

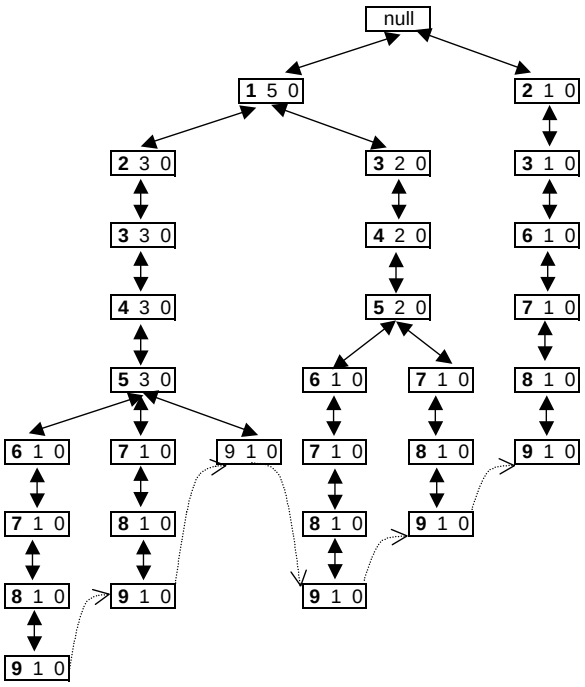


Figure 3: Headerless FP-tree: An Example.

transaction items are used in constructing the FP-Tree.

Each ordered sub-transaction is compared to the prefix tree starting from the root. If there is a match between the prefix of the sub-transaction and any path in the tree starting from the root, the support in the matched nodes is simply incremented, otherwise new nodes are added for the items in the suffix of the transaction to continue a new path, each new node having a support of one. During the process of adding any new item-node to the FP-Tree, a link is maintained between this item-node in the tree and its entry in the header table. The header table holds one pointer per item that points to the first occurrences of this item in the FP-Tree structure.

Our tree structure is the same as the FP-tree except for the following differences. We call this tree Headerless-Frequent-Pattern-Tree or HFP-tree.

1. We do not maintain a header table, as a header table is used to facilitate the generation of the conditional trees in the FP-growth model. It is not needed in our leap traversal approach;

2. We do not need to maintain the links between the same itemset across the different tree branches (horizontal links);

3. The links between nodes are bi-directional to allow top-down and bottom-up traversals of the tree;

4. All leaf nodes are linked together as the leaf nodes are the start of any pattern base and linking them helps the discovery of frequent pattern bases;

5. In addition to *support*, each node in the HFP-tree has a second variable called *participation*. *Participation* plays a similar role in the mining process as the *participation* counter in the COFI-tree [7].

Basically, the support represents the support of a node, while participation represents, at a given time in the mining process, the number of times the node has participated in already counted patterns. Based on the difference between the two variables, *participation* and *support*, the special patterns called *frequent-path-bases* are generated. These are simply the paths from a given node x, with participation smaller than the support, up to the root, (i.e. nodes that did not fully participate yet in frequent patterns). Figure 3 presents the Headerless FP-tree for the same dataset used in Figure 2.

Algorithm 1 shows the main steps in our approach. After building the Headerless FP-tree with 2 scans of the database, we mark some specific nodes in the pattern lattice using *FindFrequentPatternBases*. Using the FPBs, the leap-traversal in *FindMaximals* discovers the maximal patterns at the frequent pattern border in the lattice.

Algorithm 1 HFP-Leap: Leap-Traversal with Headerless FP-tree

Input: D (transactional database); σ (Support threshold).
Output: Maximal patterns with their respective supports.

Scan D to find the set of frequent 1-temsets $F1$
Scan D to build the Headerless FP-tree HFP
$FPB \leftarrow$ FindFrequentPatternBases(HFP)
$Maximals \leftarrow$ FindMaximals(FPB, σ)
Output $Maximals$

Algorithm 3 shows how patterns in the lattice are marked. The linked list of leaf nodes in the HFP-tree is traversed to find upward the unique paths representing sub-transactions. If frequent maximals exist, they have to be among these complete sub-transactions. The participation counter helps reusing nodes exactly as needed to determine the frequent path bases.

3.2 Construction of the COFI trees

A COFI-tree [8] is a projection of each frequent item in the original FP-tree [12] (not the Headerless FP-tree). Each COFI-tree, for a given frequent item, presents the co-occurrence of this item with other frequent items that have more support than it. In other words, if we have 4 frequent items A, B, C, D where A has the smallest support, and D has the highest, then the COFI-tree for A presents co-occurrence of item A with respect to B, C and D, the COFI-tree for B presents item B with C and D. COFI-tree for C presents item C with D. Finally, the COFI-tree for D is a root node tree. Each node in the COFI-tree has two main variables, support and participation. Participation indicates the number of patterns the node has participated in at a given time during the mining step. Based on the difference between these two variables, participation and support, frequent-path-bases are generated. The COFI-tree has also a header

Algorithm 2 COFI-Leap: Leap-Traversal with COFI-tree

Input: D (transactional database); σ (Support threshold).
Output: Maximal patterns with their respective supports.

Scan D to find the set of frequent 1-itemsets $F1$
Scan D to build the FP-tree $FP-TREE$
$A \leftarrow$ frequent item with least support
for $\forall A$ **do**
  Generate COFI tree for A, $A-COFI$
  $FPB \leftarrow$ FindFrequentPatternBases($A-COFI$)
  $Maximals \leftarrow$ FindMaximals(FPB, σ)
  Output $Maximals$
  Cleat $A-COFI$
  $A \leftarrow$ Next item with larger support if still exists
end for

table that contains all locally frequent items with respect to the root item of the COFI-tree. Each entry in this table holds the local support, and a link to connect its item with its first occurrences in the COFI-tree. A link list is also maintained between nodes that hold the same item to facilitate the mining procedure. Frequent pattern bases are generated from each COFI tree alone using the same approach used by the Headerless FP-tree. One of the advantages of using COFI-trees over the Headerless FP-tree is that we can skip building some COFI-trees during the mining process. This is due to the fact that before we build any COFI-tree we check all its local frequent items, if all its items are a subset of an already discovered maximal pattern, then there is no need to build and mine this COFI-tree as all its sub-patterns are subsets of already discovered maximal patterns.

Algorithm 2 shows the main steps in the COFI-Leap approach. After building the FP-tree with 2 scans of the database, we create independently COFI trees, in each of the COFI-tree, we mark some specific nodes in the pattern lattice using *FindFrequentPatternBases*. Using the FPBs, the leap-traversal in *FindMaximals* discovers the maximal patterns at the frequent pattern border in the lattice.

3.3 Actual Mining of Frequent-Path-Bases: The Leap-Traversal approach

Algorithm 4 is the actual leap traversal to find maximals using FP-trees generated all at one time using the Headerless FP-tree or in chunks using COFI-tree approach. It starts by listing some candidate maximals stored in *PotentialMaximals* which is initialized with the frequent pattern bases that are frequent. All the non-frequent FPBs are used for the jumps of the lattice leap traversal. These FPBs are stored in the list *List* and intermediary lists *NList* and *NList2* will store the nodes in the lattice that intersection of FPBs would point to, in other words, the nodes that may lead to maximals. The nodes in the lists have two attributes: *flag* and *startpoint*. For a node n, *flag* indicates that a subtree in the intersection tree should not be considered starting from the node n. For example, if node $(A \cap B)$ has a flag C, then the subtree under the node $(A \cap B \cap C)$ should not be considered. For a given node n, *startpoint* indicates which subtrees

Algorithm 3 FindFrequentPatternBases: Marking nodes in the lattice

Input: HFP (Headerless FP-Tree) OR $A-COFI$.
Output: FPB (Frequent pattern bases with counts)

$ListNodesFlagged \leftarrow \emptyset$
Follow the linked list of leaf nodes in HFP
for each leaf node N **do**
  Add N to $ListNodesFlagged$
end for
while $ListNodesFlagged \neq \emptyset$ **do**
  $N \leftarrow$ Pop($ListNodesFlagged$) {from top of the list}
  $fpb \leftarrow$ Path from N to $root$
  fpb.branchSupport $\leftarrow N$.support - N.participation
  for each node P in fpb **do**
    P.participation $\leftarrow$ P.participation + fpb.branchSupport
    if P.participation $<$ P.support AND $\forall c$ child of P, c.participation = c.support **then**
      add P in $ListNodesFlagged$
    end if
  end for
  add fpb in FPB
end while
RETURN FPB

in the intersection tree, descendants of n, should be considered. For example, if a node $(A \cap B)$ has the startpoint D, then only the descendants $(A \cap B \cap D)$ and so on are considered, but $(A \cap B \cap C)$ is omitted. Note that $ABCD$ are ordered lexicographically. At each level in the intersection tree, when *NList2* is updated with new nodes, the theorems are used to prune the intersection tree. In other words, the theorems help avoid useless intersections (i.e. useless maximal candidates). The same process is repeated for all levels of the intersection tree until there is no other intersections to do (i.e. *NList2* is empty). At the end, the set potential maximals is cleaned by removing subsets of any sets in *PotentialMaximals*.

It is obvious in the Leap-traversal approach that superset checking and intersections plays an important role. We found that the best way to work with this is by using the bit-vector approach where each frequent item is represented by one bit in a vector. In this approach, intersection is nothing but applying the AND operation between two vectors, and subset checking is nothing but applying the AND operation followed by equality checking between two vectors. If $A \cap B = A$ then A is a subset of B.

4. RELATED WORK

There is a plethora of algorithms proposed in the literature to address the issue of discovering frequent itemsets. The most important, and at the basis of many other approaches, is *apriori* [3]. The property that is at the heart of *apriori* and forms the foundation of most algorithms simply states that for an itemset to be frequent all its subsets have to be frequent. This anti-monotone property reduces the candidate itemset space drastically. However, the generation of candidate sets, especially when very long frequent patterns exist, is still very expensive. Moreover, *apriori* is heavily I/O bound. Another approach that avoids generating and

Algorithm 4 FindMaximals: The actual leap-traversal

Input: FPB (Frequent Pattern Bases); σ (Support threshold).
Output: $Maximals$ (Frequent Maximal patterns)

{which FPBs are maximals?}
$List \leftarrow FPB$
$PotentialMaximals \leftarrow \emptyset$
for each i in $List$ **do**
  Find support of i {using branch supports}
  if support(i) $> \sigma$ **then**
    Add i to $PotentialMaximals$
    Remove i from $List$
  end if
end for

Sort $List$ based on support
$NList \leftarrow List$
$NList2 \leftarrow \emptyset$
$\forall i \in NList$ initialize i.flag $\leftarrow NULL$ AND i.startpoint $\leftarrow$ index of i in $NList$
while $NList \neq \emptyset$ **do**
  {Intersections of FPBs to select nodes to jump to}
  for each i in $NList$ **do**
    $g \leftarrow$ Intersect(i, j) {where $j \in List$ AND $i \ll j$ (in lexicographic order) AND not j.flag}
    g.startpoint $\leftarrow j$
    Add g to $NList2$
  end for

  {Pruning starts here}
  for each i in $NList2$ **do**
    Find support of i {using branch supports}
    if support(i) $> \sigma$ **then**
      Add i to $PotentialMaximals$
      Remove all duplicates or subsets of i in $NList2$; Remove i from $NList2$ {Theorem 1}
    else
      if duplicates of i exist in $NList2$ then remove them except the most right one then remove i from $NList2$ {Theorem 2}
      Remove all non frequent subsets of i from $NList2$ {Theorem 4}
      if $\exists j \in NList2$ AND $j \supseteq i$ **then**
        i.flag $\leftarrow j$ {Theorem 3}
      end if
      for all j in $List$ **do**
        if $j \gg i$.startpoint (in lexicographic order) **then**
          $n \leftarrow$ Intersect(i, j)
          Find support of n {using branch supports}
          if support(n) $< \sigma$ **then**
            Remove i from $NList2$ {Lemma 1}
          end if
        end if
      end for
    end if
  end for
  $NList \leftarrow NList2$
  $NList2 \leftarrow \emptyset$
end while
Remove any x from $PotentialMaximals$ if ($\exists M \in PotentialMaximals$ AND $x \subset M$)
$Maximals \leftarrow PotentialMaximals$
RETURN $Maximals$

testing itemsets FP-Growth [12]. FP-Growth generates, after only two I/O scans, a compact prefix tree representing all sub-transactions with only frequent items. A clever and elegant recursive method mines the tree by creating projections called conditional trees and discovers patterns of all lengths without directly generating all candidates the way *apriori* does. However, the recursive method to mine the FP-tree requires significant memory, and large databases quickly blow out the memory stack.

MaxMiner [4], DepthProject [1], GenMax [10], & MAFIA [6] are state-of-the-art algorithms that specialize in finding frequent maximal patterns. MaxMiner is an apriori-like algorithm that might need to scan the database k times to find a pattern of length k. This algorithm performs a breadth-first traversal of the search space. At the same time it performs intelligent pruning techniques to eliminate irrelevant paths of the search tree. A look-ahead strategy achieves this, where there is no need to further process a node if it, with all its extensions, is determined to be frequent. To improve the effectiveness of the superset frequency pruning, MaxMiner uses a reorder strategy. DepthProject performs a depth-first search of the lexicographic tree of the itemsets with some superset pruning. It also uses a look-ahead pruning with item reordering. The result of the mining process of DepthProject is a superset of maximal patterns and requires a post-pruning to remove non-maximal patterns. MAFIA, which is one of the fastest maximal algorithms, uses many pruning techniques such as the look-ahead used by the MaxMiner, checking if a new set is subsumed in another existing maximal set, and other clever heuristics. GenMax is a vertical approach that uses a novel strategy, progressive focusing,to find supersets. In addition, it counts supports faster using diffsets [18].

5. PERFORMANCE EVALUATIONS

To evaluate our leap-traversal approach, we conducted a set of different experiments using both approaches HFP-Leap and COFI-Leap. First, we measured their effectiveness compared to other algorithms when mining relatively small datasets. We also compared our algorithms with other state-of-the-art algorithms solely to discover the maximal patterns, in terms of speed, memory usage and scalability.

For mining Maximal Frequent Itemsets (MFIs), Depth-Project [1] was shown to achieve more than one order of magnitude speedup over MaxMiner [4]. MAFIA [6] was shown to outperform DepthProject by a factor of 3 to 5. Gouda and Zaki presented GENMAX that has been described in their work [10] as the current best method to mine the set of exact MFIs. They also claim that MAFIA is the best method for mining the superset of all MFIs.

The contenders we tested against are MAFIA [6], FPMAX [11] and GENMAX [10]. MAFIA was shown to outperform MaxMiner [4] and Depth-Project [1] for mining maximal itemsets. FPMAX is an extension of the FP-Growth [12] approach. We used an enhanced code of FPMAX that won the FIMI-2003 award for best frequent mining implementation. The implementations of these algorithms were all provided to us by their original authors or downloaded from the FIMI repository. We used the latest version of MAFIA that does not need a post-pruning step and generates directly the set of ex-

act MFIs. All our experiments were conducted on an IBM P4 2.6GHz with 1GB memory running Linux 2.4.20-20.9 Red Hat Linux release 9. Timing for all algorithms includes the pre-processing cost such as horizontal to vertical conversions for both GenMax and MAFIA. The time reported also includes the program output time. We tested these algorithms using both real and synthetic datasets. All experiments were forced to stop if their execution time reached our wall time of 5000 seconds. We made sure that all algorithms reported the same exact set of frequent itemsets on each dataset (i.e. no false positives and no false negatives).

5.1 Mining real databases

The first set of experiments we conducted mined real datasets such as Plant-Protein and retail. Plant-Protein data is a very dense dataset with about 3000 transactions using more than 7000 items (subsequence of amino-acids). The transactions represent plant proteins extracted from SWISS-PROT. In these experiments we found that FPMAX is almost always the winner in terms of speed. On the other hand, we also found that these algorithms (except Leap approach algorithms) use extremely large amount of memory, in spite of the fact that the tested database where in general small in terms of number of transactions. This observation raised the following question, "How do these algorithms behave once they start mining extremely large databases?" To test this idea we expiremented these algorithms on synthetic databases ranging from 5,000 transactions up to 50 million transactions, with dimensions ranging from 5000 items to 100,000 items. All experiments on the Plant-Protein and retail database are depicted in Figures 4, and 6 for the time comparison ones, and in Figures 5, and 7 for the memory usage comparison ones.

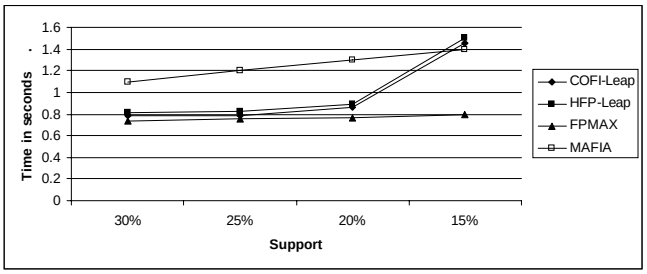


Figure 4: Mining Plant-Protein database

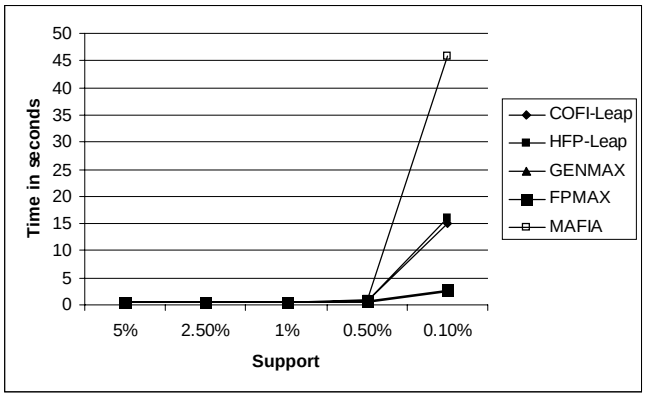


Figure 6: Mining retail database

5.2 Mining Relatively small synthetic databases

In this set of experiments, we generated synthetic datasets using [13]. We report results here only for MAFIA, FPMAX, COFI-Leap, and HFP-Leap since GenMax, inexplicably, did not generate any frequent patterns in most cases. In this set of experiments we were focusing on studying the effect of changing the support while testing three parameters: transaction length, database size, and dimension of the database. We created databases with transaction length averaging 12 and 24 items per transaction. Our datasets also have as a dimension size two of different values: 5000 and 10000 items. The database size varied from 10,000 to 250,000 transactions per database. Notice that these datasets are considered relatively sparse. COFI-Leap, HFP-Leap, and FPMAX outperformed MAFIA in some cases by two orders of magnitude. Among the three best algorithms we could not declare one of them as a de-facto winner. These experiments are depicted in Figures 8, and 9. All algorithms, except MAFIA, overlap in the figures.

5.3 Mining Extremely large synthetic databases

To Distinguish the subtle differences between Leap approach and FPMAX, we conducted our experiments on extremely large datasets. In these series of experiments we used three synthetic datasets made of 5M, 25M, and 50M transactions, with a dimension equals to 100K items, and average transaction length equals to 24. All experiments were conducted using a support of 0.05%. In mining 5M transactions the three algorithms show similar performance as COFI-Leap finishes in almost less than 300 seconds, HFP-Leap finishes its work as the second in 320 second while FPMAX finishes in 375 seconds. At 25M transactions the difference starts to increase. The final test mining a transactional database with 50M transactions, HFP-Leap discovers all patterns in 1980 second. COFI-Leap won HFP-Leap by almost 100 seconds, while FPMAX finishes in 2985 seconds. The results, averaged on many runs, are depicted in Figure 10.

From these experiments we see that the difference between FPMAX and Leap-based algorithms while mining synthetic datasets becomes clearer once we deal with extremely large datasets. Leap approachs save at least one third of the execution time compared to FPMAX. This is due to the reduction in candidate checking and to the lower memory requirements by the leap-based approachs.

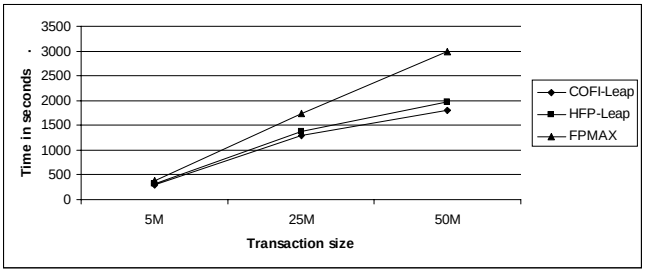


Figure 10: Mining extremely large database

5.4 Memory Usage

We also tested the memory usage by FPMAX, MAFIA HFP-Leap, and COFI-Leap while mining synthetic databases. In many cases we noticed that Leap algorithms consume one

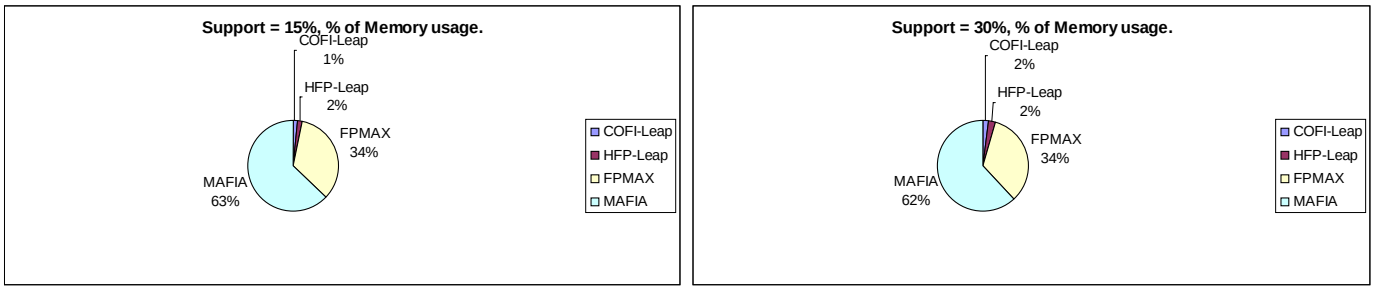


Figure 5: Memory usage while mining Plant-Protein database

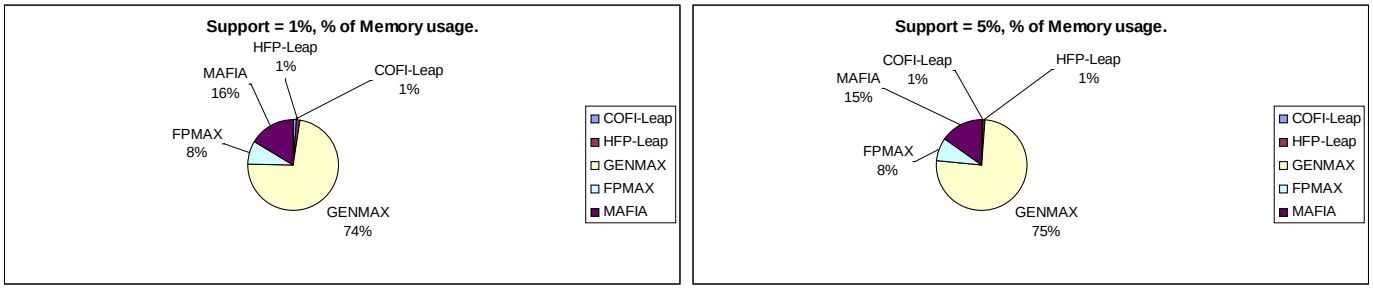


Figure 7: Memory usage while mining retail database

order of magnitude less memory than both FPMAX and MAFIA.

Figure 11 illustrates a sample of the experiments that we conducted where the transaction size, the dimension and the average transaction length are respectively 1000K, 5K and 12. The support was varied from 0.1% to 0.01%.

This low memory usage observed by the Leap approach is due to the fact that HFP-Leap generates the maximal patterns directly from its HFP-tree or from small chunks as in the case of COFI-trees. Also the intersection tree is never physically built. FPMAX, however, uses a recursive technique that keeps building trees for each frequent item tested and thus uses much more memory.

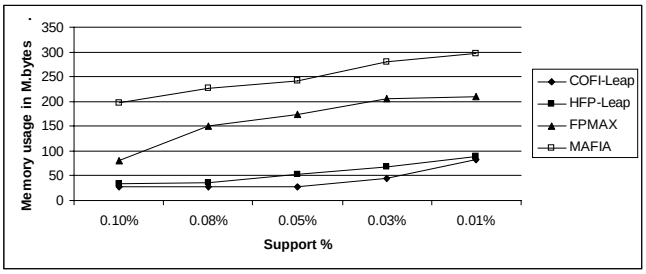


Figure 11: Memory usage

6. CONCLUSION

We presented a new way of traversing the pattern lattice to search for pattern candidates. The idea is to discover maximal patterns. Our new lattice traversal approach dramatically minimized the size of candidate list because it selectively jumps within the lattice toward the frequent pattern border. It also introduces a new method of counting the supports of candidates based on the supports of other candidate patterns, namely the branch supports of FPBs.

The leap-traversal is implemented with two different approaches. Both approaches are based on existing data structures, FP-tree, that we conveniently modified, and COFI-tree. Our contribution is a new way to mine those structures using a tree of pattern intersections with a set of pruning methods to accelerate the discovery process. This tree of intersection is what helps the jumping process.

The leap-traversal approach significantly reduces the number of candidates to check, and lends itself as a good framework for constraint-based mining and parallel processing. Our performance studies show that our approach outperforms some the state of the art methods that have the same objective: discovering all maximal patterns by, in some cases, two order of magnitudes while mining synthetic or extremely large datasets. This algorithm shows drastic saving in terms of memory usage as it has a small footprint in the main memory at any given time.

7. REFERENCES

[1] R. Agrawal, C. Aggarwal, and V. Prasad. Depth first generation of long patterns. In *In 7th Int'l Conference on Knowledge Discovery and Data Mining*, 2000.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data*, pages 207–216, Washington, D.C., May 1993.

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

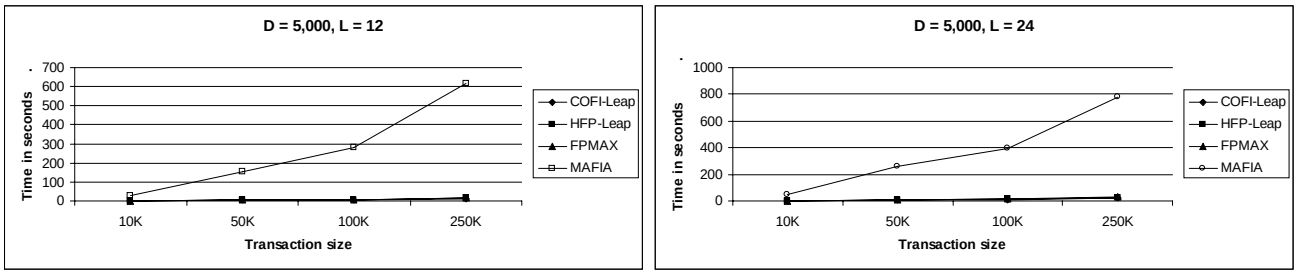


Figure 8: Mining synthetic database. D = 5000 items. Support = 0.5%

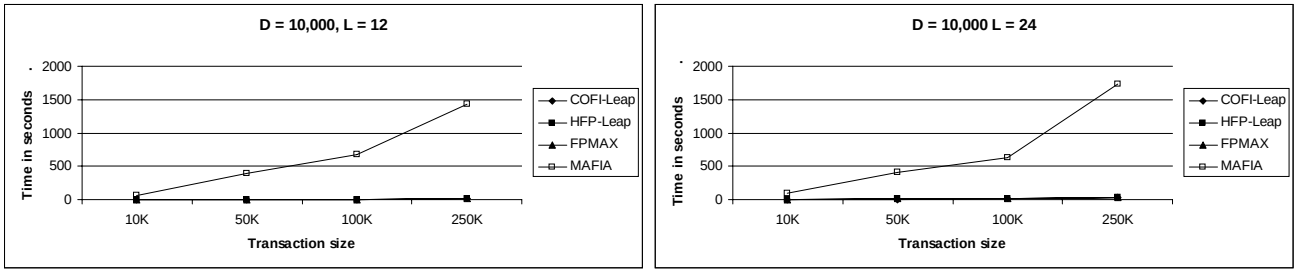


Figure 9: Mining synthetic database. D = 10000 items. Support = 0.5%

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *ACM SIGMOD*, 1998.

[5] F. Beil, M. Ester, and X. Xu. Frequent term-based text clustering. In *Proc. 8th Int. Conf. on Knowledge Discovery and Data Mining (KDD '2002), Edmonton, Alberta, Canada*, 2002.

[6] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, pages 443–452, 2001.

[7] M. El-Hajj and O. R. Zaïane. Inverted matrix: Efficient discovery of frequent items in large datasets in the context of interactive mining. In *In Proc. 2003 Int'l Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD)*, August 2003.

[8] M. El-Hajj and O. R. Zaïane. Non recursive generation of frequent k-itemsets from frequent pattern tree representations. In *In Proc. of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWak'2003)*, September 2003.

[9] R. Feldman and H. Hirsh. Mining associations in text in the presence of background knowledge. In *Proc. 2st Int. Conf. Knowledge Discovery and Data Mining*, pages 343–346, Portland, Oregon, Aug. 1996.

[10] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *ICDM*, pages 163–170, 2001.

[11] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI'03, Workshop on Frequent Itemset Mining Implementations*, November 2003.

[12] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM-SIGMOD*, Dallas, 2000.

[13] IBM_Almaden. Quest synthetic data generation code. http://www.almaden.ibm.com/cs/quest/syndata.html.

[14] H. Mannila. Inductive databases and condensed representations for data mining. In *International Logic Programming Symposium*, 1997.

[15] A. Rungsawang, A. Tangpong, P. Laohawee, and T. Khampachua. Novel query expansion technique using apriori algorithm. In *TREC, Gaithersburg, Maryland*, 1999.

[16] O. R. Zaïane and M. El-Hajj. Pattern lattice traversal by selective jumps. In *In Proc. 2005 Int'l Conf. on Data Mining and Knowledge Discovery (ACM-SIGKDD)*, August 2005.

[17] O. R. Zaïane, J. Han, and H. Zhu. Mining recurrent items in multimedia with progressive resolution refinement. In *Int. Conf. on Data Engineering (ICDE'2000)*, pages 461–470, San Diego, CA, February 2000.

[18] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report Technical Report 01-1, Department of Computer Science, Rensselaer Polytechnic Institute, 2001.

PLWAP Sequential Mining: Open Source Code *

C.I. Ezeife
School of Computer Science
University of Windsor
Windsor, Ontario N9B 3P4
cezeife@uwindsor.ca

Yi Lu
Department of Computer Science
Wayne State University
Detroit, Michigan
luyi@wayne.edu

Yi Liu
School of Computer Science
University of Windsor
Windsor, Ontario N9B 3P4
woddlab@uwindsor.ca

ABSTRACT

PLWAP algorithm uses a preorder linked, position coded version of WAP tree and eliminates the need to recursively re-construct intermediate WAP trees during sequential mining as done by WAP tree technique. PLWAP produces significant reduction in response time achieved by the WAP algorithm and provides a position code mechanism for remembering the stored database, thus, eliminating the need to re-scan the original database as would be necessary for applications like those incrementally maintaining mined frequent patterns, performing stream or dynamic mining.

This paper presents open source code for both the PLWAP and WAP algorithms describing our implementations and experimental performance analysis of these two algorithms on synthetic data generated with IBM quest data generator. An implementation of the Apriori-like GSP sequential mining algorithm is also discussed and submitted. A web log pre-processor for producing real input to the algorithms is made available too.

Keywords

sequential patterns, web usage mining, WAP tree, pre-order linkage

1. INTRODUCTION

Basic association rule mining with the Apriori algorithm [1] finds database items (attributes) that occur most often together in database transactions. Thus, given a set of transactions (similar to database records), where each transaction is a set of items (attributes), an association rule is of the form $X \rightarrow Y$, where X and Y are sets of items and $X \cap Y = \emptyset$. Association rule mining algorithms generally first find all frequent patterns (itemsets) as all combinations of items or attributes with support (percentage occurrence in the entire database), greater or equal to a predefined minimum support. Then, in the second stage of mining, association rules are generated from each frequent pattern by defining all possible combinations of rule antecedent (head) and consequent (tail) from items composing the frequent patterns such that $antecedent \cap consequent = \emptyset$ and $antecedent \cup consequent = frequentpattern$. Then, only rules with confidence (number of transactions that contain the rule divided by the number of transactions containing the antecedent) greater than or equal to a pre-defined minimum confidence are retained as valuable, while the rest are pruned.

Sequential mining is an extension of basic association rule mining that accommodates ordered set of items or attributes, where the same item may be repeated in a sequence. While basic frequent pattern has a set of non-ordered items that have occurred together up to minimum support threshold, frequent sequential pattern has a sequence of ordered items that have occurred frequently in database transactions at least as often as the minimum support threshold. Thus, the measures of support and confidence used in association rule mining for deciding frequent itemsets are used in sequential mining for deciding frequent sequences. Just as an i-itemset contains i items, an n-sequence contains n ordered items (events). One application of sequential mining is web usage mining for finding the relationship among different web users' accesses from web access logs [5], [11], [4] and [19]. Analysis of these access data can help for server performance enhancement and direct marketing in e-commerce as well as web personalization. Before applying sequential mining techniques to web log data, the web log transactions are pre-processed to group them into set of access sequences for each user identifier and to create web access sequences in the form of a transaction database (e.g., *abdac*, *eaebcac*, *babfaec*, *babfaec*). Access sequence $S' = e'_1 e'_2 \dots e'_l$ is called a subsequence of an access sequence, $S = e_1 e_2 ... e_n$, and S is a super-sequence of S', denoted as $S' \subseteq S$, if and only if for every event e'_j in S', there is an equal event e_k in S, while the order that events occurred in S is the same as the order of events in S'. For example, with $S' = ab$, S = *babcd*, S' is a subsequence of S, and ac is a subsequence of S, although there is b occurring between a and c in S. In the sequence *babcd*, while *bcd* is a suffix subsequence of *bab*, *bab* is a prefix subsequence of *bcd*. Techniques for mining sequential patterns from web logs fall into Apriori or non-Apriori.

This paper presents discussions of the implementations of three key sequential mining algorithms PLWAP, WAP and GSP used in [7].

*(A full version of this paper is available in the Journal of Data Mining and Knowledge Discovery 10, 5-38, 2005.)

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

1.1 Related Work

Work on mining sequential patterns in web log include the GSP [3], the PSP [13], the G sequence [18] and the graph traversal [15] algorithms. Agrawal and Srikant proposed three algorithms (Apriori, AprioriAll, AprioriSome) for sequential mining in [2]. The GSP (Generalized Sequential Patterns) [3] algorithm is 20 times faster than the Apriori algorithm. The GSP Algorithm makes multiple passes over data. The first pass determines the frequent 1-item patterns (L_1). Each subsequent pass starts with a seed set: the frequent sequences found in the previous pass (L_{k-1}). The seed set is used to generate new candidate sequences (C_k) by performing an Apriori gen join of L_{k-1} with (L_{k-1}). This join requires that every sequence s in the first L_{k-1} joins with other sequences s' in the second L_{k-1} if the last *k-2* elements of s are the same as the first *k-2* elements of s'. For example, if frequent 3-sequence set L_3 has the following 6 sequences: {((1,2)(3)), ((1,2)(4)), ((1)(3, 4)), ((1,3)(5)), ((2)(3,4)), ((2)(3)(5))}, to obtain frequent 4-sequences, every frequent 3-sequence should join with the other 3-sequences that have the same first two elements as its last two elements. Sequence s=((1,2)(3)) joins with s'=((2)(3,4)) to generate a candidate 4-sequence ((1,2)(3,4)) since the last 2 elements of s, (2)(3), match with the first 2 elements of s'. Similarly, ((1,2) (3)) joins with ((2)(3)(5)) to form ((1,2)(3)(5)). There are no more matching sequences to join in L_3. The join phase is followed with the pruning phase, when the candidate sequences with any of their contiguous (k-1)-subsequences having a support count less than the minimum support, are dropped. The database is scanned for supports of the remaining candidate k-sequences to find frequent k-sequences(L_k), which become the seed for the next pass, candidate (k+1)-sequences. The algorithm terminates when there are no frequent sequences at the end of a pass, or when there are no candidate sequences generated. The GSP algorithm uses a hash tree to reduce the number of candidates that are checked for support in the database.

The PSP (Prefix Tree For Sequential Patterns) [13] approach is much similar to the GSP algorithm [3], but stores the database on a more concise prefix tree with the leaf nodes carrying the supports of the sequences. At each step k, the database is browsed for counting the support of current candidates. Then, the frequent sequence set, L_k is built.

The Graph Traversal mining [14], [15], uses a simple unweighted graph to store web sequences and a graph traversal algorithm similar to Apriori algorithm to traverse the graph in order to compute the k-candidate set from the (k-1)-candidate sequences without performing the Apriori-gen join. From the graph, if a candidate node is large, the adjacency list of the node is retrieved. The database still has to be scanned several times to compute the support of each candidate sequence although the number of computed candidate sequences is drastically reduced from that of the GSP algorithm. Other tree based approaches include [18] called G sequence mining. This algorithm uses wildcards, templates and construction of Aggregate tree for mining.

The FP-tree structure [9] first reorders and stores the frequent non-sequential database transaction items on a prefix tree, in descending order of their supports such that database transactions share common frequent prefix paths on the tree. Then, mining the tree is accomplished by recursive construction of conditional pattern bases for each frequent 1-item (in ordered list called f-list), starting with the lowest in the tree. Conditional FP-tree is constructed for each frequent conditional pattern having more than one path, while maximal mined frequent patterns consist of a concatenation of items on each single path with their suffix f-list item. FreeSpan [8] like the FP-tree method, lists the f-list in descending order of support, but it is developed for sequential pattern mining. PrefixSpan [16] is a pattern-growth method like FreeSpan, which reduces the search space for extending already discovered prefix pattern p by projecting a portion of the original database that contains all necessary data for mining sequential patterns grown from p.

Table 1: Sample Web Access Sequence Database for WAP-tree

TID	Web access sequence	Frequent subsequence
100	abdac	abac
200	eaebcac	abcac
300	babfaec	babac
400	afbacfc	abacc

Web access pattern tree (WAP), is a non-Apriori algorithm, proposed by Pei et al. [17]. The WAP-tree stores the web log data in a prefix tree format similar to the frequent pattern tree [9] (FP-tree). WAP algorithm first scans the web log to compute all frequent individual events, then it constructs a WAP-tree over the set of frequent individual events of each transaction before it recursively mines the constructed WAP tree by building a conditional WAP tree for each conditional suffix frequent pattern found. The process of recursive mining of a conditional suffix WAP tree ends when it has only one branch or is empty.

An example application of the WAP-tree algorithm for finding all frequent events in the web log (constructing the WAP-tree and mining the access patterns from the WAP tree) is shown with the database in Table 1. Suppose the minimum support threshold is set at 75%, which means an access sequence, s should have a count of 3 out of 4 records in our example, to be considered frequent. Constructing WAP-tree, entails first scanning database once, to obtain events that are frequent, a, b, c. When constructing the WAP-tree, the non-frequent (like d, e, f) part of every sequence is discarded. Only the frequent sub-sequences shown in column three of Table 1 are used as input. With the frequent sequence in each transaction, the WAP-tree algorithm first stores the frequent items as header nodes for linking all nodes of their type in the WAP-tree in the order the nodes are inserted. When constructing the WAP-tree, a virtual root (Root) is first inserted. Then, each frequent sequence in the transaction is used to construct a branch from the Root to a leaf node of the tree. Each event in a sequence is inserted as a node with count 1 from Root if that node type does not yet exist, but the count of the node is increased by 1 if the node type already exists. Also, the head link for the inserted event is connected (in broken lines) to the newly inserted node from the last node of its type that was inserted or from the header node of its type if it is the very first node of that event type inserted. Once the frequent sequential data are stored on the complete WAP-tree (Figure 1), the tree is mined for frequent patterns starting with the lowest frequent event in the header list, in our example,

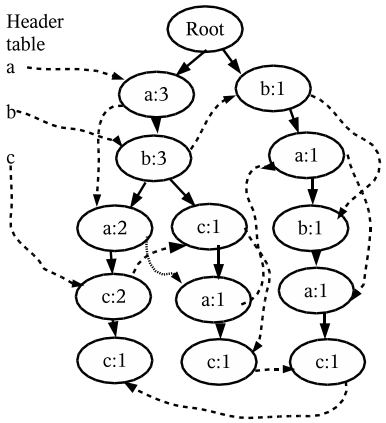


Figure 1: Construction of the Original WAP Tree

starting from frequent event c as the following discussion shows. From the WAP-tree of Figure 1, it first computes prefix sequence of the base c or the conditional sequence base of c as: aba : 2; ab : 1; abca : 1; ab : -1; baba : 1; abac : 1; aba : -1 The conditional sequence list of a suffix event is obtained by following the header link of the event and reading the path from the root to each node (excluding the node). After discarding the non-frequent part c in the above sequences, the conditional sequences based on c are listed below:
aba : 2; ab : 1; aba : 1; ab : -1; baba : 1; aba : 1; aba : -1. Using these conditional sequences, a conditional WAP tree, WAP-tree|c, is built using the same method as shown in Figure 1. The re-construction of WAP trees that progressed as suffix sequences |c, |bc discovered frequent patterns found along this line c, bc and abc. The recursion continues with the suffix path |c, |ac. The algorithm keeps running, finding the conditional sequence bases of *bac*, *b*, *a*. Figure 2 shows the WAP trees for mining conditional pattern base c. After mining the whole tree, discovered frequent pattern set is: {c, aac, bac, abac, ac, abc, bc, b, ab, a, aa, ba, aba}.

Although WAP-tree algorithm scans the original database only twice and avoids the problem of generating explosive candidate sets as in Apriori-like algorithm, its main drawback is recursively re-constructing large numbers of intermediate WAP-trees and patterns during mining taking up computing resources.

Pre-Order linked WAP tree algorithm (PLWAP) [7], [10], is a version of the WAP tree algorithm that assigns unique binary position code to each tree node and performs the header node linkages pre-order fashion (root, left, right). Both the pre-order linkage and binary position codes enable the PLWAP to directly mine the sequential patterns from the one initial WAP tree starting with prefix sequence, without re-constructing the intermediate WAP trees. To assign position codes to a PLWAP node, the root has null code, and the leftmost child of any parent node has a code that appends '1' to the position code of its parent, while the position code of any other node has '0' appended to the position code of its nearest left sibling. The PLWAP technique presents a much better performance than that achieved by the WAP-tree technique as shown in extensive performance analysis.

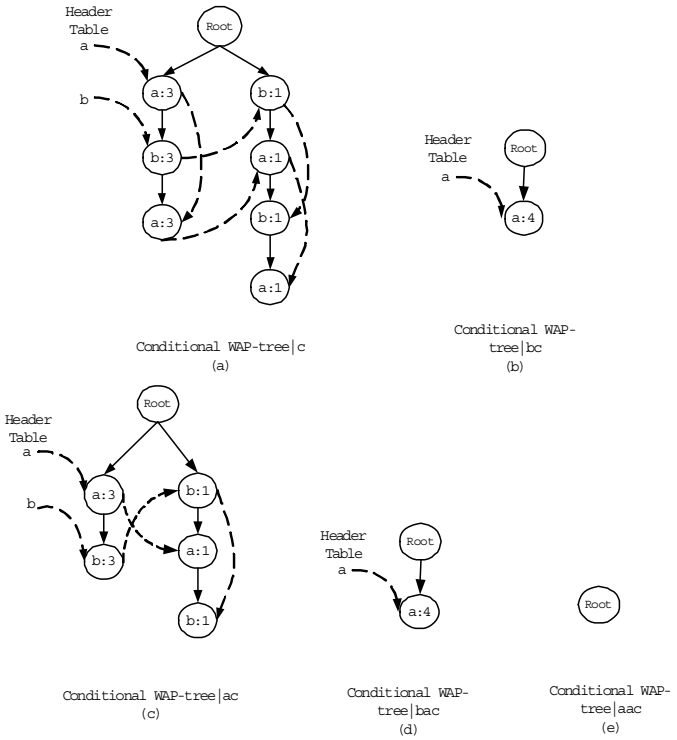


Figure 2: Reconstruction of WAP Trees for Mining Conditional Pattern Base c

1.2 Motivations and Contributions

PLWAP algorithm [7], a recently proposed sequential mining tool in the Journal of Data Mining and Knowledge Discovery has many attractive features that makes it suitable as a building block for many other sophisticated sequential data mining approaches like incremental mining [6], web classification and personalization. This paper supplements the detailed and formal presentations of the PLWAP algorithm, its properties and theorems given in [7] by focusing on details of the code implementations of PLWAP, WAP and GSP sequential miners as well as making available a real web log data pre-processor and providing further indepth analysis. This paper thus, contributes by discussing our code implementations of the PLWAP and two other key sequential mining algorithms (WAP and GSP) used in performance studies of work in [7]. These algorithms have been tested thoroughly on publicly available data sets (http://www.almaden.ibm.com/software/quest/Resources/index.shtml) and with real data. Through this paper, a real web log pre-processor for preparing real input data for the miners is also made available (ask author if needed). A more indepth performance analysis that confirms the stability of the PLWAP algorithm is another contribution of paper. Such code availability motivates adoption of algorithm as a building block for more sophisticated data mining processes like stream and dynamic mining, distributed, incremental, drill-down and roll-up mining, semantic mining and sensor network mining.

1.3 Outline of the Paper

Section 2 discusses example mining with the Pre-Order Linked WAP-Tree (PLWAP) algorithm. Section 3 discusses

the C++ implementations of the PLWAP, WAP and the GSP algorithms for sequential mining. Section 4 discusses experimental performance analysis, while section 5 presents conclusions and future work.

2. AN EXAMPLE SEQUENTIAL MINING WITH PLWAP ALGORITHM

Unlike the conditional search in WAP-tree mining, which is based on finding common suffix sequence first, the PLWAP technique finds the common prefix sequences first. The main idea is to find a frequent pattern by progressively finding its common frequent subsequences starting with the first frequent event in a frequent pattern. For example, if *abcd* is a frequent pattern to be discovered, the WAP algorithm progressively finds suffix sequences *d*, *cd*, *bcd* and *abcd*. The PLWAP tree, on the other hand, would find the prefix event *a* first, then, using the suffix trees of node *a*, it will find the next prefix subsequence *ab* and continuing with the suffix tree of *b*, it will find the next prefix subsequence *abc* and finally, *abcd*. Thus, the idea of PLWAP is to use the suffix trees of the last frequent event in an *m-prefix sequence* to recursively extend the subsequence to *m+1* sequence by adding a frequent event that occurred in the suffix trees. Using the position codes of the nodes, the PLWAP is able to know the descendant and sibling nodes of oldest parent nodes on the suffix root set of a frequent header element being checked for appending to a prefix subsequence if it is frequent in the suffix root set under consideration. An element is frequent if the sum of the supports of the oldest parent nodes on all its suffix root sets is greater than or equal to the minimum support.

Assume we want to mine the web access database (WASD) of Table 1 for frequent sequences given a minimum support of 75% or 3 transactions. Constructing and mining the PLWAP tree goes through the following steps. (1) Scan WASD (column 2 of Table 1 once to find all frequent individual events, L as {a:4, b:4, c:4}. The events d:1, e:2, f:2 have supports less than the minimum support of 3 and
(2) Scan WASD again, construct a PLWAP-tree over the set of individual frequent events (column 3 of Table 1), by inserting each sequence from root to leaf, labeling each node as (node event: count: position code). Then, after all events are inserted, traverse the tree pre-order way to connect the header link nodes. Figure 3 shows the completely constructed PLWAP tree with the pre-order linkages.

(3) Recursively mine the PLWAP-tree using common prefix pattern search: The algorithm starts to find the frequent sequence with the frequent 1-sequence in the set of frequent events(FE) {a, b, c}. For every frequent event in FE and the suffix trees of current conditional PLWAP-tree being mined, it follows the linkage of this event to find the first occurrence of this frequent event in every current suffix tree being mined, and adds the support count of all first occurrences of this frequent event in all its current suffix trees. If the count is greater than the minimum support threshold, then this event is appended (concatenated) to the last subsequence in the list of frequent sequences, F. The suffix trees of these first occurrence events in the previously mined conditional suffix PLWAP-trees are now in turn, used for mining the next event. To obtain this conditional PLWAP-tree, we only need to remember the roots of the current suffix trees, which are stored for next round mining. For example, the

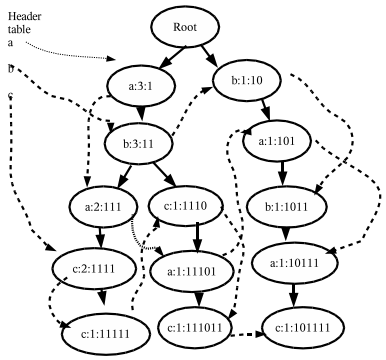


Figure 3: Construction of PLWAP-Tree Using Pre-Order Traversal

algorithm starts by mining the tree in Figure 4(a) for the first element in the header linkage list, *a* following the *a* link to find the first occurrencies of *a* nodes in *a:3:1* and *a:1:101* of the suffix trees of the Root since this is the first time the whole tree is passed for mining a frequent 1-sequence. Now, the list of mined frequent patterns F is {a} since the count of event *a* in this current suffix trees is 4 (sum of a:3:1 and a:1:101 counts), and more than the minimum support of 3. The mining of frequent 2-sequences that start with event *a* would continue with the next suffix trees of *a* rooted at {b:3:11, b:1:1011} shown in Figure 4(b) as unshadowed nodes. The objective here is to find if 2-sequences *aa*, *ab* and *ac* are frequent using these suffix trees. In order to confirm aa frequent, we need to confirm event *a* frequent in the current suffix tree set, and similarly, to confirm *ab* frequent, we should again follow the *b* link to confirm event *b* frequent using this suffix tree set, same for *ac*. The process continues to obtain same frequent sequence set {a, aa, aac, ab, aba, abac, abc, ac, b, ba, bac, bc, c} as the WAP-tree algorithm.

3. C++ IMPLEMENTATIONS OF THREE SEQUENTIAL MINING ALGORITHMS

Although, we lay more emphasis on the PLWAP algorithm developed in our lab, we also provide the source codes of the WAP and GSP algorithms used for performance analysis. The source codes are discussed under seven headings of (1) development and running environment, (2) input data format and files, (3) minimum support format, (4) output data format and files, (5) functions used in the program, (6) data structures used in the program and (7) additional information. All of the codes are documented with information on this section and more for code readability, maintainability and extendability. Each program is stored in a .cpp file and compiled with "g++ filename.cpp". It is worth noting that all three algorithms were implemented in the year 2002, when no other versions of the WAP and GSP algorithms were publicly available. While we cannot still find a publicly available version of the GSP program, there are some subtle differences between our implementation of the WAP algorithm and that now available at http://www.cs.ualberta.ca/ tszhu/software.html. One difference we have noticed is with the input formats of the two implementations: while Alberta version accepts the input in one sequence, we accept a list of sequences belonging to different users as in web log mining. It also seems like Alberta

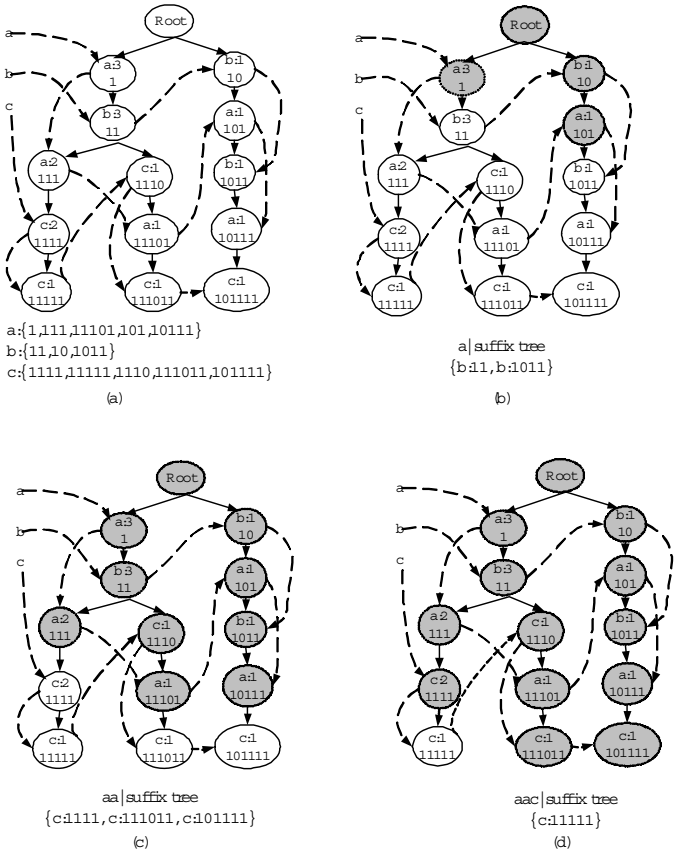


Figure 4: Mining PLWAP-Tree to Find Frequent Sequence Starting with aa

version is memory-only as intermediate pattern input is not saved on disk, while it is saved on disk in our implementation. The implication is that running the memory-only version on an environment with less memory than data size would result in operating system swapping of data onto disk.

3.1 C++ Implementation of the PLWAP Sequential Mining Algorithm

This is the PLWAP algorithm program, which is based on the description in [7], C.I. Ezeife and Y. Lu. "Mining Web Log sequential Patterns with Position Coded Pre-Order Linked WAP-tree" in DMKD 2005.

1. DEVELOPMENT ENVIRONMENT: Although initial version is developed under the hardware/software environment specified below, the program runs on more powerful and faster multiprocessor UNIX environments as well. Initial environment is: (i)Hardware: Intel Celeron 400 PC, 64M Memory; (ii)Operating system: Windows 98; (iii)Development tool: Inprise(Borland) C++ Builder 6.0. The algorithm is developed under C++ Builder 6.0, but compiles and runs under any standard C++ development tool.

2. INPUT FILES AND FORMAT: Input file is test.data: For simplifying input process of the program, we assume that all input data have been preprocessed such that all events belonging to the same user id have been gathered together, and formed as a sequence and saved in a text file, called, "test.data". The "test.data" file may be composed of hundreds of thousands of lines of sequences where each line represents a web access sequence for each user. Every line of the input data file ("test.data") includes UserID, length of sequence and the sequence which are separated by tab spaces. An example input line is: 100 5 10 20 40 10 30 . Here, 100 represents UserID, the length of sequence is 5, the sequence is 10,20,40,10,30.

3. MINIMUM SUPPORT FORMAT: The program also needs to accept a value between 0 and 1 as minimum support. The minimum support input is entered interactively by the user during the execution of the program when prompted. For a minimum support of 50%, user should type 0.5, and for minsupport of 5%, user should type .05, and so on.

4. OUTPUT FILES AND FORMAT: result_PLWAP.data: Once the program terminates, we can find the result frequent patterns in a file named "result_PLWAP.data". It may contain lines of patterns, each representing a frequent pattern.

5. FUNCTIONS USED IN THE CODE: (i)BuildTree: builds the PLWAP tree, (ii)buildLinkage: Builds the pre-order linkage for PLWAP tree, (iii)makeCode: makes the position code for a node, (1v)checkPosition: checks the position between any two nodes in the PLWAP tree, (v)MiningProcess: mines sequential frequent patterns from the PLWAP tree using position codes.

6. DATA STRUCTURE: Three struct are used in this program: (i) the node struct indicates a PLWAP node which contains the following information: a.the event name, b.the number of occurrence of the event, c. a link to the position code, d. length of position code, e. the linkage to next node with same event name in PLWAP tree, f. a pointer to its left son, g. a pointer to its right sibling, h. a pointer to its parent, i. the number of its sons. (ii) a position code struct implemented as a linked list of unsigned integer to make it possible to handle data of any size without exceeding the maximum integer size. (iii) a linkage struct.

7. ADDITIONAL INFORMATION: The run time is displayed on the screen with start time, end time and total seconds for running the program.

3.2 C++ Implementation of the WAP Sequential Mining Algorithm

This is the WAP algorithm program based on the description in [17]: Jian Pei, Jiawei Han, Behzad Mortazavi-asl, and Hua Zhu, "Mining Access Patterns Efficiently from Web Logs", PAKDD 2000.

1.DEVELOPMENT ENVIRONMENT: The same as described for PLWAP and can run on any UNIX system as well.

2. INPUT FILES AND FORMAT:

i) Input file test.data: Pre-processed sequential input records are read from the file "test.data". The "test.data" file, which is the same format as described for PLWAP above.

ii) Input file middle.data: used to save the conditional middle patterns. During the WAP tree mining process, following the linkages, once the sum of support for an event is found greater than minimum support, all its prefix conditional patterns are saved in the "middle.data" file for next round mining. The format of "middle.data" is as follows: Each line includes the length of the sequence, the occurrence of the sequence, and the events in the sequence. For example, given a line in middle.data: 5 4 10 20 40 10 30, the length of sequence is 5, 4 indicates the sequence occurred 4 times in the previous conditional WAP tree and the sequence is 10,20,40,10,30.

3. MINIMUM SUPPORT:

The program also needs to accept a value between 0 and 1 for minimum support or frequency. The user is prompted for minimum support when the program starts.
4. OUTPUT FILES AND FORMAT: result_WAP.data
Once the program terminates, the result patterns are in a file named "result_WAP.data", which may contain lines of patterns.
5. FUNCTIONS USED IN THE CODE:
(i)BuildTree: Builds the WAP tree/conditional WAP tree
(ii)MiningProcess: produces sequential pattern/conditional prefix sub-pattern from WAP tree/conditional WAP tree.
6. DATA STRUCTURE: three struct are used in this program: (i) the node struct indicates a WAP node which contains the following information: a.the event name, b.the number of occurrence of the event, c. the linkage to next node same event name in WAP tree, d. a pointer to its left son, e. a pointer to its rights sibling, f. a pointer to its parent.
(ii) a linkage struct described in the program.
7. ADDITIONAL INFORMATION: The run time is displayed on the screen with start time, end time and total seconds for running the program.

3.3 C++ Implementation of the GSP Sequential Mining Algorithm

This is a GSP algorithm implementation, which demonstrates the result described in [3]: R. Srikant and R. Agrawal. "Mining sequential patterns: Generalizations and performance improvements", 1996.
1.DEVELOPMENT ENVIRONMENT: The same as described for PLWAP and runs in any UNIX system as well.
2. INPUT FILES AND FORMAT: Input file test.data: The main input file "test.data" had the same format as for PLWAP described above.
3. MINIMUM SUPPORT: The program also needs to accept a value between 0 and 1 as minimum support. The user is prompted for minimum support when the program starts.
4. OUTPUT FILES AND FORMAT: result_GSP.data At the termination of the program, the result patterns are in a file named "result_GSP.data", which may contain lines of frequent patterns.
5. FUNCTIONS USED IN THE CODE: (i)GSP: reads the file and mines levelwise according to the algorithm.
6. DATA STRUCTURES: There are struct for i) candidate sequence list and its count and ii) sequence.
7. ADDITIONAL INFORMATION: The run time is displayed on the screen with start time, end time and total seconds for running the program.

4. PERFORMANCE AND EXPERIMENTAL ANALYSIS OF THREE ALGORITHMS

The PLWAP algorithm eliminates the need to store numerous intermediate WAP trees during mining, thus, drastically cutting off huge memory access costs. The PLWAP annotates each tree node with a binary position code for quicker mining of the tree. This section compares the experimental performance analysis of PLWAP with the WAP-tree mining and the Apriori-like GSP algorithms. The three algorithms were initially implemented with C++ language running under Inprise C++ Builder environment. All initial experiments were performed on 400MHz Celeron PC machine with 64 megabytes memory running Windows 98 (for work in [7]). Current experiments are conducted with the same implementations of the programs and still on synthetic datasets generated using the resource data generator code from http://www.almaden.ibm.com/software/quest/Resources/index.shtml. This synthetic dataset has been used by most pattern mining studies [3, 12, 17]. Experiments were also run on real datasets generated from web log data of University of Windsor Computer Science server and pre-processed with our web log cleaner code. The correctness of the implementations were confirmed by checking that the frequent patterns generated for the same dataset by all algorithms are the same. A high speed UNIX SUN microsystem with a total of 16384 Mb memory and 8 x 1200 MHz processor speed is used for these experiments. The synthetic datasets consist of sequences of events, where each event represents an accessed web page. The parameters shown below are used to generate the data sets.
$|D|$ Number of sequences in the database
$|C|$: Average length of the sequences
$|S|$: Average length of maximal potentially frequent sequence
$|N|$: number of events
For example, C10.S5.N2000.D60K means that $|C| = 10$, $|S| = 5$, $|N| = 2000$, and $|D| = 60$K. It represents a group of data with average length of the sequences as 10, the average length of maximal potentially frequent sequence is 5, the number of individual events in the database is 2000, and the total number of sequences in database is 60 thousand. The datasets with different parameters test different aspects of the algorithms. Generally, if the number of these four parameters becomes larger, the execution time becomes longer. Experiments are conducted to test the behavior of the algorithms with respect to the four parameters, minimum support threshold, database sizes, number of database table attributes and the average length of sequences. For each experiment, while one of the parameters changes, others are pegged at some constant values. Observations are made at three levels of small, medium and large (e.g., small database size may consist of a table with records less than 40 thousands, medium size table has between 50 nd 200 thousands records, while large has over 300 thousand).
Experiment 1: Execution time trend with different minimum supports (small size database, 40K records):
This experiment uses fixed size database and different minimum support to compare the performance of PLWAP, WAP and GSP algorithms. The datasets are described as C2.5.S5.N50.D40K, and algorithms are tested with minimum supports between 0.05% and 20% against the 40 thousand (40K) database with 50 attributes and average sequence length of 2.5. The results of this experiment are shown in Table 2 and Figure 5 with the number of frequent patterns found reported as Fp. From this result, it can be seen that at minimum support of 0.05%, the number of frequent patterns found is highest and is 2729, the PLWAP algorithm ran almost 3 times faster than the WAP algorithm. As the minimum support increases, the number of frequent patterns found decreases and the gain in performance by the PLWAP algorithm over the WAP algorithm decreases. It can be seen that the more the number of frequent patterns found in a dataset, the higher the performance gain achieved by the PLWAP algorithm over the WAP algorithm. This is

Table 2: Execution Times for Dataset at Different Minimum Supports (small database)

Alg	Runtime (in secs) at Different Supports(%)						
	0.05 Fp= 2729	0.1 Fp= 1265	0.5 Fp = 268	1 Fp= 111	5 Fp= 23	10 Fp= 10	15 Fp 0
GSP	6663	3646	1054	636	157	30	1
WAP	149	66	20	8	3	1	1
PLWAP	54	26	7	3	1	1	1

because the two algorithms spend about the same amount of time scanning the database and constructing the tree, but the PLWAP algorithm saves on storing and reading intermediate re-constructed tree, which are as many as the number of frequent patterns found. The execution times of the two algorithms are the same when there are nearly no frequent patterns.

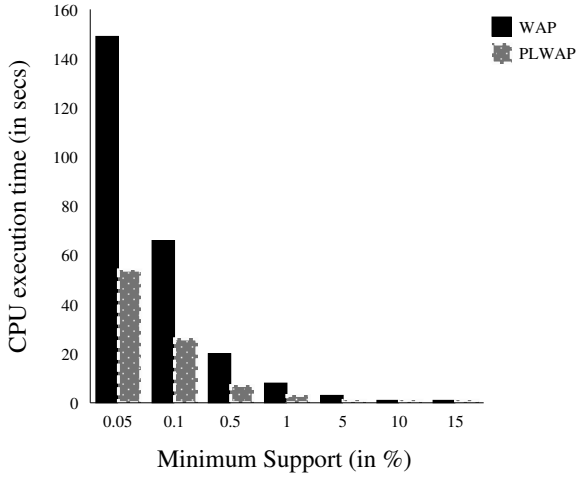


Figure 5: Execution Times Trend with Different Minimum Supports (small database)

Experiment 2: Execution times trend with different minimum supports (Medium size database, 200K records):
This experiment uses fixed size database and different minimum support to compare the performance of PLWAP, WAP and GSP algorithms. The algorithms are tested with minimum supports between 0.05% and 15% against the 200 thousand (200K) database, 50 attributes and average sequence length of 2.5. The results of this experiment are shown in Table 3 and Figure 6 with the number of frequent patterns found reported as Fp. It can be seen that the trend in performance is the same as with small database. When the minimum support reaches 15%, there are no frequent patterns found and the running times of the two algorithms become the same.
Experiment 3: Execution Times for Dataset at Different Minimum Supports (large database):
This experiment uses fixed size database and different minimum support to compare the performance of PLWAP, WAP and GSP algorithms. The algorithms are tested with mini-

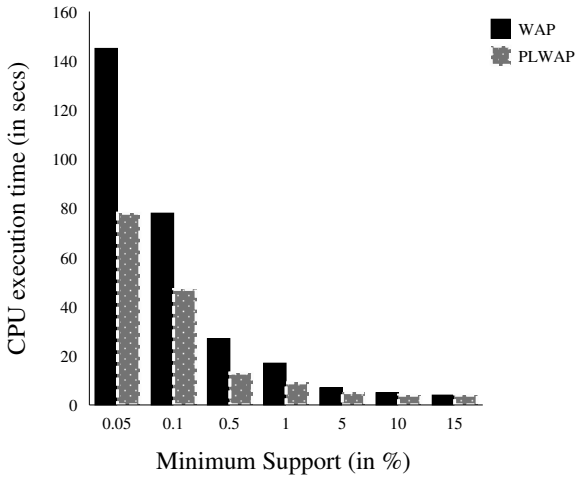


Figure 6: Execution Times Trend with Different Minimum Supports (medium database)

Table 3: Execution Times for Dataset at Different Minimum Supports (medium database)

Alg	Runtime (in secs) at Different Supports(%)						
	0.05 Fp= 2630	0.1 Fp= 1271	0.5 Fp = 20	1 Fp= 114	5 Fp= 23	10 Fp= 10	15 Fp 0
GSP	34275	10021	11742	3320	785	150	4
WAP	145	78	27	17	7	5	4
PLWAP	78	47	13	9	5	4	4

mum supports between 0.05% and 15% against one million (1M) record database. The results of this experiment are shown in Table 4 and Figure 7.
Experiment 3a: Execution Times for Dataset at Different Minimum Supports (large database but very low minimum supports):
This experiment uses fixed size database and different minimum support to compare the performance of PLWAP and WAP algorithms (GSP not included because running times get too big or process is killed). The algorithms are tested with minimum supports between 0.001% and 0.02% against one million (1M) record database. The results of this experiment are shown in Table 5 and Figure 8.

Table 4: Execution Times for Dataset at Different Minimum Supports (large database)

Alg	Runtime (in secs) at Different Supports(%)						
	0.05 Fp= 2646	0.1 Fp= 1269	0.5 Fp = 273	1 Fp= 116	5 Fp= 23	10 Fp= 10	15 Fp 0
GSP	73084	41381	12854	7358	1715	328	9
WAP	387	100	41	27	14	10	9
PLWAP	236	87	28	17	10	9	8

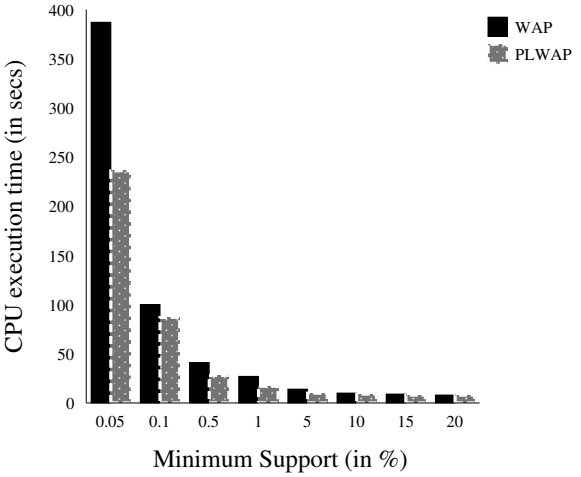


Figure 7: Execution Times for Dataset at Different Minimum Supports (large database)

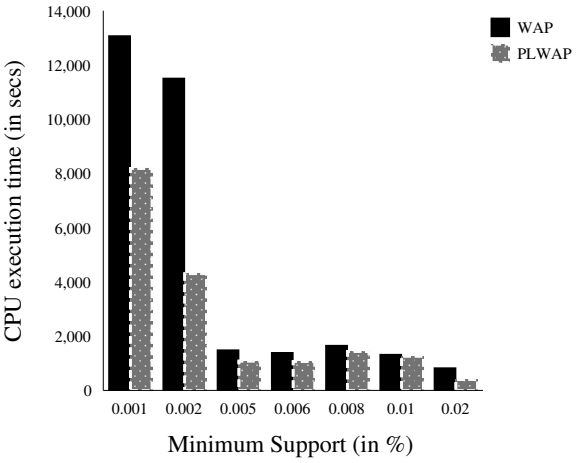


Figure 8: Execution Times for Dataset at Different Minimum Supports (large database and low minsupports)

Experiment 4: Execution times trend with different database sizes (small size database, 2K to 14K):
This experiment uses fixed minimum support and different size database to compare the performance of PLWAP, WAP and GSP algorithms. The algorithms are tested on databases of sizes 2K to 14K at minimum support of 1%. The gain in performance by the PLWAP algorithm is constant across different sizes because the number of frequent patterns in different sized datasets generated by the data generator seem to be about the same at a particular minimum support. The results of this experiment are shown in Table 6 and Figure 9.

Experiment 5: Execution times trend with different database sizes (medium size database, 20K to 200K):
This experiment uses fixed minimum support and different size database to compare the performance of PLWAP and WAP algorithms. The algorithms are tested with database sizes between 20 and 200 thousands at minimum support of 1%. The results of this experiment are shown in Table 7 and

Table 5: Execution Times for Dataset at Different Minimum Supports (large database and low minsupports)

Alg	Runtime (in secs) at Different Supports(%)						
	0.001 Fp= 222K	0.002 Fp= 161K	0.005 Fp = 38174	0.006 Fp= 38174	0.008 Fp= 14931	0.01 Fp= 12228	0.02 Fp 6196
WAP	13076	11512	1494	1396	1661	1329	832
PLWAP	8183	4303	1083	1066	1435	1261	403

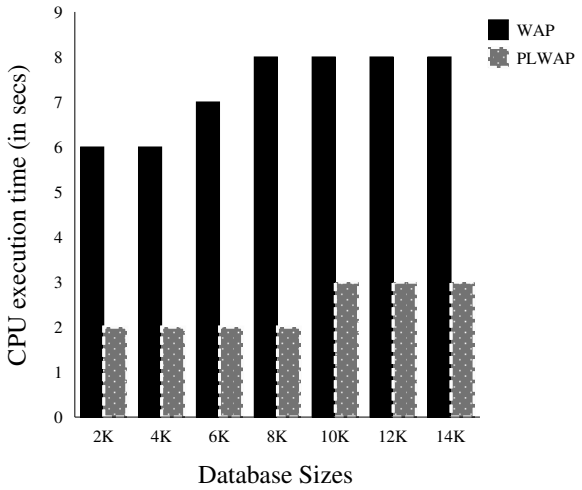


Figure 9: Execution Times Trend with Different Minimum Supports (small database)

Figure 10.

Experiment 6: Execution times trend with different database sizes (large size database, 300K to 900K):
This experiment uses fixed minimum support and different size database to compare the performance of PLWAP, WAP and GSP algorithms. The algorithms are tested with database sizes between 300K and 900K records at minimum support of 1%. The results of this experiment are shown in Table 8 and Figure 11. Since the CPU execution time difference between the WAP and PLWAP from this experiment, seems to diminish as the database size increases, a further experiment was run on larger databases of between one million and 20 million records to check if and when WAP would run faster than the PLWAP. Result of this experiment

Table 6: Execution Times for Different Database Sizes at Minsupport (small database)

Alg	Runtime (in secs) at Different Supports(%)						
	2K	4K	6K	8K	10K	12K	14K
GSP	30	59	91	125	161	193	214
WAP	6	6	7	8	8	8	8
PLWAP	2	2	2	2	3	3	3

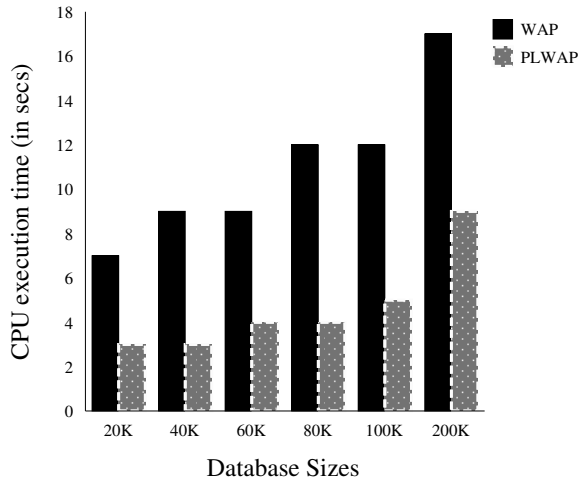


Figure 10: Execution Times Trend with Different Database sizes (medium database)

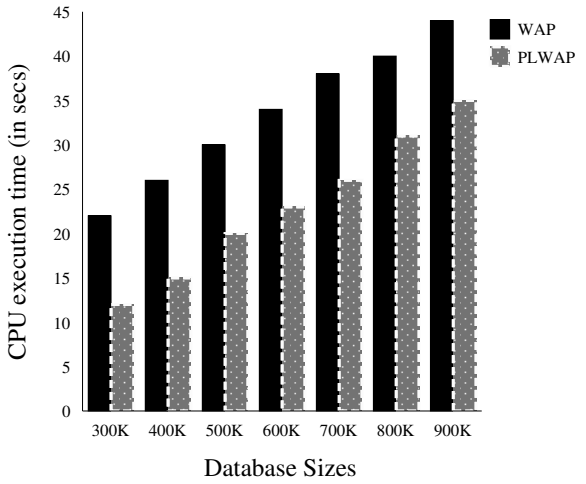


Figure 11: Execution Times Trend with Different Database sizes (large database)

Table 7: Execution Times for Different Database Sizes at Minsupport of 1%(medium database)

Alg	Runtime (in secs) at Different Db sizes(%)					
	20K	40K	60K	80K	100K	200K
GSP	306	639	1015	1338	2004	3320
WAP	7	9	9	12	12	17
PLWAP	3	3	4	4	5	9

Table 9: Execution Times for Different Database Sizes at Minsupport of 1% (larger database)

Alg	Runtime (in secs) at Different DB sizes(%)					
	1M	2M	4M	8M	10M	20M
WAP	23	47	93	186	57	464
PLWAP	23	47	93	186	57	463

on larger databases is given in Table 9. From this experiment, we found that at the same minimum support of 1%, when the database size increases much, there are few or no frequent patterns found because an item needs to appear in about 200 thousand sequential records in the 20 million database to be frequent and that is not very possible in the synthetic data leading to about the same execution times for the two algorithms. However, a run on the same 20M records at a minimum support of 0.1% found 350 patterns and took 560 seconds of CPU time for PLWAP, while the WAP tree algorithm could not complete successfully.

Experiments were also run to check the behavior of the algorithms with varying sequence lengths and number of database attributes, and the PLWAP always runs faster than the WAP algorithm when the average sequence length is less than 20 and there are some found frequent patterns. However, the PLWAP performance starts to degrade when the average sequence length of the database is more than 20 because with extremely long sequences, there are nodes with position codes that are more than "maxint", in our current implementation, we use a number of variables to store a node's position code that are linked together. Thus, managing and retrieving the position code for excessively long sequences could turn out to be time consuming. Our experiment on real data of size 10K having about 2500 attributes with only a few very long sequences of up to 166 items while most of other records are less than 7 items long, reveals PLWAP faster than WAP by a factor of over 11 times at a very low minimum support of 0.05% with their execution times as 449, 5157, at 0.06%, they are 234, 1187 and at 0.07%, 144, 1071 respectively. The execution times of the two algorithms start being the same from minimum support 1% when there are no found frequent patterns. A test on memory usage of the WAP and PLWAP algorithms reveals about the same amount of memory allocated to both programs.

Table 8: Execution Times for Different Database Sizes at Minsupport of 1% (large database)

Alg	Runtime (in secs) at Different DB sizes(%)						
	300K	400K	500K	600K	700K	800K	900K
GSP	5329	7110	8778	12K	13K	15K	17K
WAP	22	26	30	34	38	40	44
PL-WAP	12	15	20	23	26	31	35

5. CONCLUSIONS

This paper discusses the source code implementation of the PLWAP algorithm presented in [7] as well as our implementations of two other sequential mining algorithms WAP [17] and GSP [3] that PLWAP was compared with. Extensive experimental studies were conducted on the three implemented algorithms using IBM quest synthetic datasets. From the experiments, it was discovered that the PLWAP algorithm, which mines a pre-order linked, position coded version of the WAP tree, always outperforms the other two

algorithms and is much more efficient. PLWAP improves on the performance of the efficient tree-based WAP tree algorithm by mining with position codes and their suffix tree root sets, rather than storing intermediate WAP trees recursively. Thus, it saves on processing time and more so when the number of frequent patterns increases and the minimum support threshold is low. PLWAP's performance seems to degrade some with very long sequences having sequence length more than 20 because of the increase in the size of position code that it needs to build and process for very deep PLWAP tree. For mining sequential patterns from web logs or databases, the following aspects may be considered for future work. The PLWAP algorithm could be extended to handle sequential pattern mining in large traditional databases to handle concurrency of events. The position code features of the PLWAP tree provides a mechanism for concisely storing small items not represented in the tree for future incremental refreshment of mined patterns. It also enables easy multilevel mining of frequent patterns at detailed (e.g., item level like city level or word level) to more generalized calss level (e.g., country level or book level) through rollup mining with the same tree by providing level-wise pre-order header linkages. This may make it useful in several sequence oriented applications like frequent word usages in collections of documents (books), cell phone call sequence record data, intrusion detection and sensor network mining as well as biological sequence data. A more efficient implementation of the position code management for long sequences would make the PLWAP more scalable.

6. ACKNOWLEDGMENTS

This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor grant.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on very Large Databases Santiago, Chile*, pages 487–499, 1994.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering (ICDE '95) Taipei Taiwan*, pages 3–14, 1995.

[3] R. S. R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the Fifth International Conference On Extending Database Technology (EDBT '96) Avignon France*, pages 3–17, 1996.

[4] J. Borges and M. Levene. Data mining of user navigation patterns. In *Proceedings of the KDD Workshop on Web Mining San Diego California*, pages 31–36, 1999.

[5] O. Etzioni. The world wide web: Quagmire or gold mine. *Communications of the ACM*, 39(1):65 – 68, 1996.

[6] C. Ezeife and M. Chen. Mining web sequential patterns incrementally with revised plwap tree. In *Proceedings of the fifth International Conference on Web-Age Information Management (WAIM 2004)*, pages 539–548. Springer Verlag, July 2003.

[7] C. Ezeife and Y. Lu. Mining web log sequential patterns with position coded pre-order linked wap-tree. *International Journal of Data Mining and Knowledge Discovery (DMKD) Kluwer Publishers*, 10(1):5–38, 2005.

[8] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the 2000 Int. Conference on Knowledge Discovery and Data Mining (KDD'00)*, pages 355–359, 2000.

[9] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *International Journal of Data Mining and Knowledge Discovery*, 8(1):53–87, Jan 2004.

[10] Y. Lu and C. Ezeife. Position coded pre-order linked wap-tree for web log sequential pattern mining. In *Proceedings of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2003)*, pages 337–349. Springer, May 2003.

[11] S. Madria, S. Bhowmick, W. Ng, and E. Lim. Sampling large databases for finding association rules. In *Proceedings of the first International Data Warehousing and Knowledge Discovery DaWak99*, 1999.

[12] F. Masseglia, F. Cathala, and P. Poncelet. Psp : Prefix tree for sequential patterns. In *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD'98) Nantes France LNAI*, pages 176–184, 1998.

[13] F. Masseglia, P. Poncelet, and R. Cicchetti. An efficient algorithm for web usage mining. *Networking and Information Systems Journal (NIS)*, 2(5-6):571–603, 1999.

[14] A. Nanopoulos and Y. Manolopoulos. Finding generalized path patterns for web log data mining. *Data and Knowledge Engineering*, 37(3):243–266, 2000.

[15] A. Nanopoulos and Y. Manolopoulos. Mining patterns from graph traversals. *Data and Knowledge Engineering*, 37(3):243–266, 2001.

[16] J. Pei, J. Han, B. Mortazavi-Asl, and H. Pinto. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 001 International Conference on Data Engineering (ICDE 01)*, pages 214–224, 2001.

[17] J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. Mining access patterns efficiently from web logs. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00) Kyoto Japan*, 2000.

[18] M. Spiliopoulou. The laborious way from data mining to web mining. *Journal of Computer Systems Science and Engineering Special Issue on Semantics of the Web*, 14:113–126, 1999.

[19] J. Srivastava, R. Cooley, M. Deshpande, and P. Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explorations*, 1, 2000.

On Benchmarking Frequent Itemset Mining Algorithms

from Measurement to Analysis

Balázs Rácz
Computer and Automation Research Institute of the Hungarian Academy of Sciences
Lágymanyosi u. 11., H-1111 Budapest, Hungary
bracz+fp5@math.bme.hu

Ferenc Bodon
Department of Computer Science and Information Theory,
Budapest University of Technology and Economics
Magyar tudósok körútja 2. H-1117 Budapest, Hungary
bodon@cs.bme.hu

Lars Schmidt-Thieme
Computer Based New Media Group (CGNM)
Albert-Ludwigs-Universität Freiburg
Georges-Koehler-Allee, D-79110 Freiburg, Germany
lst@informatik.uni-freiburg.de

ABSTRACT

We point out problems of current practices in comparing Frequent Itemset Mining Implementations, and suggest techniques that can help to avoid the conclusions of measurements being tainted by these problems.

1. INTRODUCTION

Frequent Itemset Mining (FIM) is one of the initial, most basic problems of Data Mining. It has wide applications not only in its natural form, but also as a subroutine in various other problems, the most famous being association rule mining.

During the past decade, over 100 papers were published about Frequent Itemset Mining. Some of these brought novel approaches, while others tuned them with heuristics and data structure optimizations. There is one common feature in all the papers: an implementation was benchmarked and shown to be faster/better (in memory usage or disk access) than... some other (publicly available) implementation, on some mining tasks (datasets and support threshold levels). While based on this approach everybody claimed that their algorithm is the fastest/best on the field, this can obviously be not true.

This called for the 'Frequent Itemset Mining Implementations' (FIMI) workshops[4] held in conjunction with ICDM03 and ICDM04, where members of the community were invited and encouraged to send their implementation. A benchmark was run over all submitted implementations and a wide set of publicly available input data. This raised the evaluation of FIM algorithms to a new level, thus fortunately holding back the flood of proudly presented papers.

Detailed understanding of the problem Frequent Itemset Mining, neither of existing algorithms and approaches is not necessary to read this paper. Nevertheless, we give some pointers to the readers who are not familiar with the field [3, 4].

The rest of the paper is organized as follows: In Section 2 we discuss problems and benchmarking issues of the FIMI contests. Section 3 introduces programming techniques, including the proposal of a unified, highly optimized I/O framework. In Section 4 we present issues concerning the actual measurement, machine dependance, and selection of performance metric. We also show sample analysis diagrams for selected FIM implementations. In Appendix A a short introduction is given to modern computing hardware, which may be necessary to understand some details of Section 4.

2. ON FIMI CONTESTS

There is an important difference between traditional data mining contests and Frequent Itemset Mining evaluation. In tranditional contests, there are two important measures: the *quality* of the algorithm, i.e., how good the output of the algorithm is (like prediction quality), and the running time of the implementation. As in case of the FIM problem the task is mathematically defined and solvable, i.e., there is one correct output, the quality is measured only in terms of resource usage: mostly running time and also memory usage.

PROBLEM 1. *We are interested in the quality of algorithms, but we can only measure implementations.*

This is an issue of all fields of Computer Science where theoretical considerations, such as asymptotical running time analysis can yield only very loose bounds on actual performance.

PROBLEM 2. *If we gave our algorithms and ideas to a very talented and experienced low-level programmer, that could completely re-draw the current FIMI rankings.*

PROBLEM 3. *Seemingly unimportant implementation details can hide all algorithmic features when benchmarking. These details are often unnoticed even by the author and almost never published.*

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

Such differences can be explained by hardware reasons, for example some implementations of the same algorithms may be better served by the memory hierarchy than others. See Figure 1 for an illustration of 10-fold difference in running times.

The history of FIM algorithms is a very good example of people not knowing what to aim for. The initial hypothesis was that I/O size is the factor that primarily determines performance (this resulted in the level-wise apriori-like algorithms). In the meantime focus was moved to reducing the number of false candidates (resulted in different hashing techniques). This hypothesis was also obsoleted by the stunning performance of depth-first mining algorithms like eclat and FP-growth, known to generate more candidates than their predecessors. The question comes naturally: What determines the performance of current algorithms/implementations?

PROBLEM 4. *FIM implementations are complete 'suites' of a basic algorithm and several algorithmic/implementational optimizations. Comparing such complete 'suites' tells us* what *is fast, but does not tell us* why.

Recommendation 1. The suites should be programmed in a way that the optimization features can be turned on/off, and several possible underlying data structures are pluggable. Running benchmarks over these options would give us an insight on what counts and what doesn't.

PROBLEM 5. *All 'dense' mining tasks' run time is dominated by I/O.*

The effect can be as tremendous that 75–90% of the total running time is spent in rendering the mined frequent itemsets to text. The total output size is often on the order of several hundred megabytes of even gigabytes.

PROBLEM 6. *On 'dense' datasets FIMI benchmarks are measuring the ability of submitters to code a fast integer-to-string conversion function.*

Still, a data mining program has no real relevance if it does not output its result. Furthermore, several optimizations could be employed if it is known that the output is discarded – this would harm the fairness of comparison and make our conclusions irrelevant to practical purposes.

Recommendation 2. When comparing benchmark result of different FIM implementations, as much of the code should be identical, as possible.

PROBLEM 7. *The run time differences between different contestants are often very small.*

This is due to the nature of the mining tasks: except of very low support threshold test cases, the run time is on the order of a second, or even a fraction of a second.

PROBLEM 8. *Run time of a particular executable on a particular input is not a number but rather a distribution, i.e., it varies from run to run.*

This is true even when we provide an environment as clean as (from a user's point) possible: disable hyperthreading, use single-user mode, disable all services (including network services, and all non-vital system services), disable the GUI of the operating system, and of course not run any other programs multitasking. It is strange to see, that though almost none of the above requirements can be satisfied under Microsoft Windows OS, still many authors use it as a benchmarking environment. Although the effects may be hidden by the low precision of the OS timer, anyway.

Recommendation 3. 'Winner takes all' evaluation of a mining task on a FIMI benchmark is unfair.

PROBLEM 9. *Traditional run-time (+memory need) benchmarks do not tell us whether an implementation is better than an other in algorithmic aspects, or implementational (hardware-friendliness) aspects.*

PROBLEM 10. *Traditional benchmarks do not show whether on a slightly different hardware architecture (like AMD vs. Intel) the conclusions would still hold or not.*

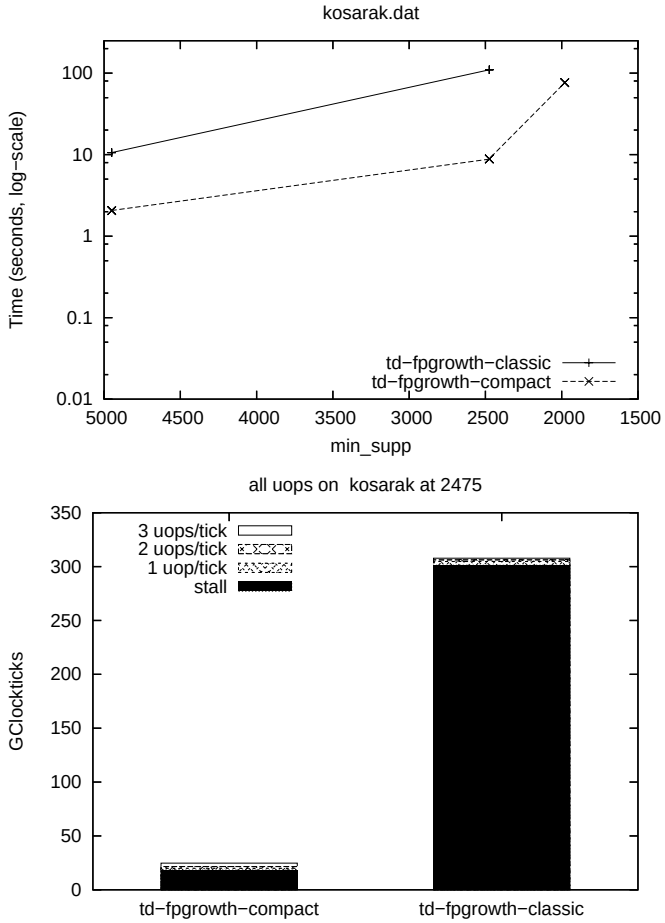
Recommendation 4. Traditional benchmarks should be executed either on virtual machines, or extended with some monitoring of hardware friendliness, most importantly the efficiency of memory hierarchy (caches), and branch prediction.

In the following sections we take the recommendations one-by-one and show our analysis system that employs them without sacrificing performance and applicability.

3. PROGRAMMING TECHNIQUES

Due to personal attitudes or in order to reach marginal performance improvement some competitive FIM algorithms are coded in plain C. The C language only partially supports modularization, hence these codes are very difficult to read, but more importantly, the modification by other researchers is error-prone and laborious. This is not a problem when a code is embedded into a system as a black-box, where FIM is regarded to be an elementary function. Nevertheless, if we want to analyze, understand the performance gains and losses, and reuse a code, then non-flexibility and the procedural approach is a serious obstacle.

In a perfect FPM world anybody who figures out a new solution to a certain subtask of an algorithm, should be able to embed it without duplication or considerable modification into the best code of the algorithm. In other words, the codes should be modular, i.e., they should fulfill object-oriented requirements. It may sound surprising but even a single algorithm can be programmed in an OO manner, where data structures and/or even functions that work on data structures are represented by different classes. For example, in our testbed the APRIORI class has separate subclasses (1.) for representing a data structure, (2.) for determining the frequent items and pairs, (3.) generating candidates, (4.) removing infrequent candidates from the data structure, (5.) caching the transactions. The subclasses further use other classes, for example the trie data structure uses a class that stores the edges, which can be a simple linked list, an indexed array (with or without offset reduction), a hybrid/double representation and so on. Each class conforms to an interface, and if anybody has a good idea that only applies to a certain part of the algorithm, then it is possible to exchange only the respective part of the code, given that the functional description and the interface is followed.



The two implementations of TD-FP-Growth[10] share 90% of the code. The variant titled 'classic' uses separately allocated nodes and pointers as links. The 'compact' variant uses an array of nodes and integers as links, ensured that nodes of the same item occupy an interval of the array.
The bottom diagram shows the total number of clockticks used by the programs on a single mining task, and (with different colors/patterns) the distribution of how many instructions were executed per clocktick. It is clearly seen that the two implementations use roughly the same number of instructions, but the classic node-based one stalls the processor 10 times more than the array-based. These stalls are caused by the memory access pattern: as memory accesses are scattered, a huge amount of cache misses delay execution. In the first case, memory access is contiguous, and the hardware prefetch mechanism loads the required data into the cache before its needed, thus no delay occurs on execution.

Figure 1: Example of difference caused by seemingly unimportant implementational details.

Object oriented approach greatly supports Recommendation 2 and 1 because every part of the code can be reused or separately exchanged and analyzed. If interfaces are strictly kept, code segments could be easily be changed, and we would also better understand why a certain optimization, that is described on algorithmic level, performs different on different implementations of the same algorithm. Although function substitution can be solved by using macros in plain C as well, this solution has many disadvantages (referred to as messy, error-prone, inelegant, etc by the programmer's community) and should be avoided in building a library for FPM.

The advantages of object-oriented approach are beyond dispute, and are required by the FPM community. Opening the black-boxes would make it possible to acquire a better understanding about the algorithms and the performance improvement techniques. Doing this, however, is not trivial, and most believe that the answer is *no* to the following question: *Can we preserve efficiency while rewriting codes so that they fulfill the object-oriented requirements?*

The answer being *no* can easily be justified, if we want to rewrite our code in the classic object-oriented way, i.e by declaring *virtual* those functions that we want to be exchangeable. Figure 2 illustrates this with a small example.

```
class Alg{
  virtual void f(args){ ... }

  void g(){
    ...
    f(values);
    ...
     }
};

class SpecAlg : public Alg{
  void f(args){ ... }
};

int main(){
  Alg* alg;
  if(cond)
    alg = new Alg(...);
  else
    alg = new SpecAlg(...);
  ...
}
```

Figure 2: OO programming with virtual function

There are two reasons for the running-time increase of this solution over the classic C implementation. First, the function call and argument passing to function f requires operations on the stack. Second, calling a virtual function means doing an indirect call, which does not only require a pointer dereference, but is also inefficient to be executed on modern processors. Furthermore, this approach greatly reduces the compiler optimization possibilities. Consequently, this kind of rewriting does not result in a code that is able to compete with the classic C counterparts. Fortunately, there exists another solution.

All drawbacks of the virtual functions can be avoided by using *templates* and *inline* functions. Figure 2 illustrates how we can keep object-oriented features and avoid virtual functions at the same time.

The compiler will actually generate two *Alg* classes that have nothing to with each other, and use no virtual functions. With the help of inline functions we avoid parameter passing and let the compiler do proper code optimization. Using this technique we can do everything that object-oriented programming requires (i.e. abstraction, data encapsulation, polymorphism, inheritance).

```
class AlgBase{
  void f(args);
};
inline void AlgBase::f(args){ ... }

class SpecAlg {
  void f(args);
};
inline void SpecAlg::f(args){ ... }

template <class T>
class Alg : public T{
  void g(){
    ...
    f(values);
    ...
  }
 };

int main(){
  if(cond)
  {
    Alg<Algbase> alg(...);
    ...
  }
  else
  {
    Alg<SpecAlg> alg(...);
    ...
  }
}
```

Figure 3: OO programming with templates and inline

The aforementioned technique has some further advantages and also minor drawbacks. For example, one can declare a boolean template argument, and it acts as a compile-time constant (equivalent to the much less elegant #define), thus any branches on that expression will be optimized away. A slight disadvantage is that the template construct results in a code bloat: all functions (with template arguments) will be compiled separately for all actually used instantiations (template parameter combinations). Furthermore, implementing template-based modularity needs careful design, and in some rare cases (such as circular references of classes) a very good understanding of the C++ programming language is necessary.

3.1 Our IO framework

As many of the authors have noticed that time spent in output routines can dominate the total running time, the FIMI contributions are rich in IO routines and tricks to speed up outputting: buffering, string representation caching, fast integer to string conversion, calling low-level buffer manipulating methods, etc. As we already mentioned the comparisons would be more fair, if all implementations used the same IO routines, ideally the fastest one. In our FIM environment we aimed to develop an IO framework that is as fast and as flexible as possible. Our goal was not only to provide a fast IO framework, but we also wished to maintain the possibility of exchanging any part of it, and therefore to be able to conduct a comprehensive set of experiments. As an example for exchangeability consider file handling in C++ which can be done in three ways: using a file descriptor and low-level OS support routines, a `FILE*` and the standard C I/O library, or the `iostream` library. In our framework the file representation is a template parameter of the class that is responsible for reading in a transaction or writing out an itemset, and the representation is wrapped by a class that exports a unified interface. As the wrapper function is inlined, this does not give any overhead compared to simply incorporating the underlying file representation method into the output class.

The most important part of the IO framework is the *cached depth-first decoder class*. Its basic idea was already used in [8] (and we suppose in [9] as well) but probably due to its technical aspect it was not described in detail in any paper. Our experiments showed that when the output is huge (for example in the case of database connect with low support threshold) then rendering the item identifiers to string and writing them out takes considerable part of the running time. Improving this task has more impact on running time than many algorithmic features.

The functionality of the cached depth-first decoder class is very simple. It maintains a stack of items, whose content, together with a support value, can be written to the file. Items can only be placed to and removed from the top of the stack. For the sake of efficiency a character buffer and a second (parallel) stack that stores positions of this buffer are maintained behind the scenery. When inserting an item, its string representation is copied to the position of the buffer given by the top element of the second stack, and the new last position is pushed on top of it. This way writing out two itemsets of size ℓ that differ only in their last item needs only two conversions from integer to string instead of 2ℓ.

Even this can be saved if the string representations of the frequent items are generated and stored before the decoder class is used. Since most FIM algorithms recode the items so that only frequent items are considered and the new codes are contiguous starting from zero or one, the lookup of the string representation of an item is done in constant time and requires insignificant memory compared to the memory need of the algorithm itself.

Please note, that this depth-first output class not only suites depth-first algorithms (eclat [11], fp-growth [6], etc. and their variants [5]) but also many other algorithms as well. For example APRIORI is a breadth-first algorithm, nevertheless outputting the result is done with a recursive traversal of a tree in a depth-first manner, no matter if the frequent itemsets are written out in candidate generation, infrequent candidate removal phase or at the end of the algorithm.

Figure 4 shows the experiments of our output tester. It generates all subset of a set of a given size, and calls the decoder class to write out the actual subset. The inverse codes of the items were generated with a random number generator. The tested routines are as follows:

- `normal-simple` is the classic method, it renders each itemset and each item separately to string. (However, the int-to-string conversion routine is the optimized one, not the very slow but generic C-library routine.)
- `normal-cache` renders each itemset separately, but caches

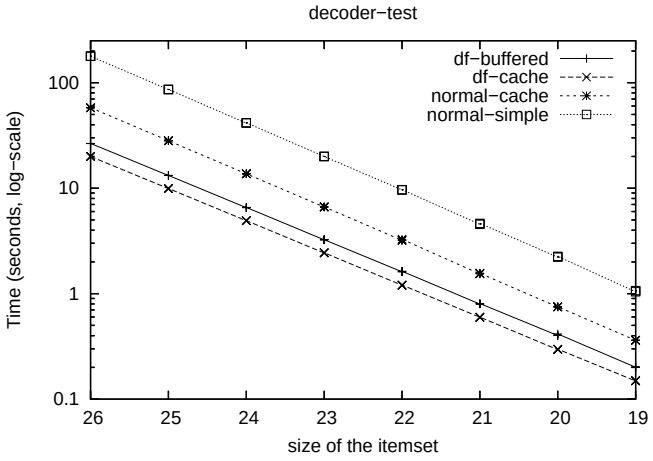


Figure 4: Performance of different output routines.
.

the item identifiers' string representation.

- `df-buffered` uses the depth-first method and reuses the string representation of the previous itemset, but when it appends the new item to the end of the line buffer, it renders it from int to string.
- `df-cache` is the implementation described above, i.e., it reuses the previous line in a depth-first approach, and uses the cached string representation of the upcoming integer.

The results show of the cached depth-first decoder. Generating an output of size 4GB (at set size 25) requires about 10 seconds. There is about a 10-fold running time difference between the original and our optimized method.

Our I/O framework contains routines for other common tasks for frequent itemset mining, such as determining item (or frequent pair) frequencies, filtering infrequent items, recoding item identifiers in frequency ascending or descending order, and inverting that recoding. Our example implementations of the next section use this library and thus only the really algorithm-specific part of the code is different. This enables a fair comparison on all datasets.

4. BENCHMARKING TECHNIQUES

First let us enumerate some desiderata about the benchmarking environment:

1. The benchmark should be **stable**, and **reproducible**. Ideally it should have no variation, surely not on the same hardware.
2. The benchmark numbers should **reflect the actual performance**. The benchmark should be a fairly accurate model of actual hardware.
3. The benchmark should be **hardware-independent**, in the sense that it should be stable against the slight variation of the underlying hardware architecture, like changing the processor manufacturer or model.

It is easy to see that *any two requirements* of the above have serious conflicts and contradictions.

It is clear that different hardware has different performance. The problem is that performance is not linear, thus it is not possible to normalize the measured values with a single performance indicator of the used hardware, and result in a unified performance metric.

PROBLEM 11. *Different algorithms/implementations may stress different aspects of the hardware. A different piece of hardware may be more advanced in one aspect, while provide lower performance in another aspect. Thus it is not possible to migrate a benchmark ranking from a particular hardware to another.*

Recommendation 5. Along benchmark results one should always describe the exact hardware used to measure the performance metrics.

Recommendation 6. Benchmark results should not form a single number based on a ranking is calculated or a graph is plotted. Instead, it should give us a slight idea about which hardware resources are stressed, so that we can extrapolate the performance indicators onto different hardware platform.

Of course this introduces extra complexity into reading benchmark results. But this is required, **performance is not as simple as 'run time in seconds'**.

4.1 Selection of benchmark platform

Basically there are three available platforms to do precise measurements including the hardware-friendliness metrics:

Virtual machine. We define an abstract machine of similar power like our current processors. (Knuth did this with defining the MIX in his famous work[7].) Then we define a cost for each operation also depending on the current state of the hardware (program execution history). This cost function should be as close to actual hardware performance, as possible. We then re-code (or compile) our FIM implementations for this processor and execute it on a proper simulator that counts the total cost and performance metrics.

There are many problems with this approach: definition of the instruction set severely influences implementation performance. The benchmark results will highly depend on the choice of cost function (and possibly its parameters if it is a parameterized cost function). Furthermore, all FIM implementations will have to be recoded or re-compiled for the new architecture. During this re-coding or re-compilation we have to re-invent all the optimization techniques that were (with a huge effort) developed for existing architectures by compiler writers. Thus there will be an additional factor, the quality of the very low-level implementation. Due to these problems, we think using a virtual machine is for the time being out of question for FIMI benchmarks.

Instrumentation. This is a technique that runs an actual executable program on a simulated processor. Each operation is separately checked and executed in the clean environment. Performance-related events can be collected and detailed, or with a proper cost function, it can be transformed to a simple 'run-time'-like aggregated performance metric.

The advantage of instrumentation is, that the environment is completely clean, the results are deterministic and reproducible. Furthermore, the parameters of the architecture (such as cache size) are arbitrarily tunable. However,

the results depend heavily on the correctness of the simulation (how well the simulated processor resembles the actual one, especially in resource availability), and even more on the choice of the event weighting/cost function. There is one more, huge disadvantage: the instrumentation framework (the simulation of the processor) is very slow, up to 100-fold increase is usual compared to the native run time of the program. This means, that even to perform a fairly straightforward benchmark suite supercomputing capacity is needed.

Run-time measurement. This employs special features of the current processors, which allow the performance of the processor to be monitored real-time. This is done by hardware **performance counters** that can be programmed to count specific performance-related events. The actual event to be counted, with the available and usable counter configurations are hardware dependent, and described by the processor manufacturer. However, the event set always includes in some form the efficiency of different architectural parts of the processor, and the possible causes of execution stalls, such as branch mispredictions, cache misses, instruction dependency stalls, etc. The usage of these counters are supported by performance optimizer software released by the processor manufacturer (like AMD CodeAnalyst^{TM} Performance Analyzer[1], or Intel VTune^{TM} Performance Analyzers[2]), or open source software (like PerfMon for linux, or the hardware-independent PAPI). Depending on the versatility and number of available counters and the events to be measured, more that a single run may be necessary to take all measurements.

The framework we release with an open source license uses the third method, the performance counters of Intel Pentium 4 and Xeon processors under Linux OS. It can be run completely user-space, only the PerfCtr patch is required to be installed on the running kernel. However, to avoid other running programs taint the benchmarks, the precautions described after Problem 8 should be adhered to as much as possible.

4.2 Sample analysis and visualization

In this subsection we give sample figures from our benchmark system, and show how the results can be interpreted. We do not go into detailed analysis neither in depth (algorithm and feature selection), nor in breadth (mining task selection), as the focus of this paper is benchmarking. We fully utilize the proposed method in an upcoming work that aims a complete in-depth analysis.

All measurements were taken on a workstation with Intel Pentium 4 2.8 Ghz processor (family 15, model 2, stepping 9) with 512 KB L2 cache, hyperthreading disabled, and 2 GB of dual-channel FSB800 main memory. The system runs a stripped-down installation of SuSE Linux 9.3, kernel 2.6.11.4-20a (SuSE version) with PerfCtr-2.6.15 patch installed.

The sample evaluation uses the dataset BMS-POS. The classic run-time diagram of the analyzed implementations is shown as Figure 5. Note that the metric shown is actually a calculated metric. We measure the number of (user-time) clockticks spent executing the current program with the PerfCtr framework. Thus the figures should be precise independently of the resolution of the OS timer. To get a run time in seconds, we simply normalized the resulting values with the nominal speed of the processor, 2.8 billion clockticks/second.

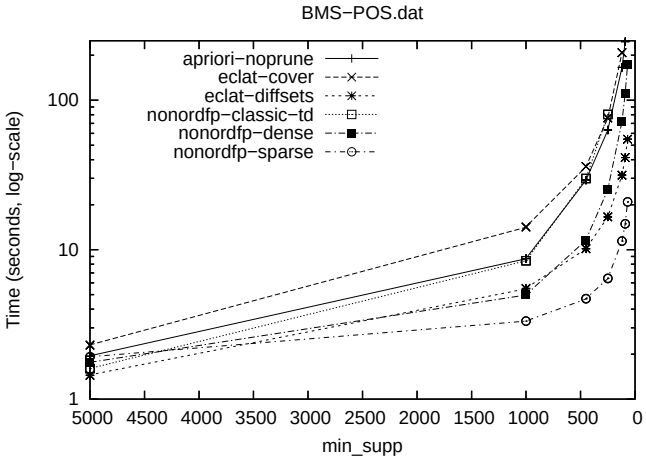


Figure 5: Run time of the example implementations on dataset BMS-POS.

Much more detailed information is available on Figure 6. While it corresponds only to one particular mining task (in this case dataset BMS-POS at support threshold of 1000), it gives us some idea about why one implementation is faster than the other. The exact numbers are listed in the Full Execution Profile on Figure 7. The following causes of performance difference should be noted:

- `nonordfp-classic-td` executes slightly less instructions than the winner, `nonordfp-sparse`. However, its performance is severely hindered by a large amount of cache misses which stall the execution unit. This shows the performance penalty of using a classic node-based representation with linked lists, against a compact array-based representation and linear scan. The non-memory related stalls are on the same order of magnitude.

- `nonordfp-dense` trails behind `nonordfp-sparse` approximately for the same reasons. Furthermore, the effect is slightly (but not considerably) enhanced by an increase in the actual number of executed instructions.

- Also the same effect can be observed when comparing `eclat-diffset` to `apriori-noprune`. While eclat practically never waits for memory, apriori has a huge penalty on cache misses.

- It is important to note that in case of the two `eclat` variants, the prefetch efficiency hides practically all effects of memory latency. This is due to the nature of the algorithm: `eclat` operates by calculating intersections (`eclat-cover`) or differences (`eclat-diffset`) of long sorted arrays. Sequential memory access has a huge advantage over scattered one.

- Nevertheless, `eclat` is not performing very well, because it has another huge disadvantage: the merge routine contains a large amount of conditional branches, which are extremely badly predictable. (These data-dependent branches are almost 50-50% random in the two directions.) This results in a stunning ¿100% overhead of bogus instructions, those instructions that were

executed on mispredicted branches and were rolled back.

- The comparison of `eclat-cover` and `eclat-diffset` gives some really interesting results. Traditionally it is believed that diffsets are well suited to dense datasets, and covers to sparse datasets, because the respective representation gives shorter lists to merge. However, in the displayed case, `eclat-cover` and `eclat-diffset` require roughly the same amount of memory accesses (30% difference at most, depending on which metric we look at), while in the run time we see over 2-fold increase. The amount of memory accesses hint that the total length of the lists to be merged/intersected is the same. The implementation using covers loses particularly on two points:

  1. The inner loop is more complex, or has to be executed more times in case of covers than diffsets (this is shown by the increased number of non-bogus instructions).
  2. There are much more data-dependent conditional branches which are badly predictable in the evaluated cover-implementation than the diffset-implementation. `eclat-cover` has over a billion mispredicted branches as opposed to any of the fp-growth implementations which has roughly 40 million.

5. CONCLUSION

We showed several problems regarding the current practices of evaluating Frequent Itemset Mining algorithms/implementations.

To work around these problems, our contribution is a sophisticated benchmark environment and an initial library for common procedures in Frequent Itemset Mining including an efficient I/O framework. This library, along with implementation modularization techniques pointed out in this paper could reach fair and in-depth comparison, and eventually help us get a better insight not only on *what* is fast for a FIM task, but also *why*.

It is important to note that the problems and possible solution techniques are not strongly connected to the task of Frequent Itemset Mining and mostly apply to other fields of applied algorithmic nature, thereby making our work relevant to a wider audience than the community of Frequent Pattern Mining.

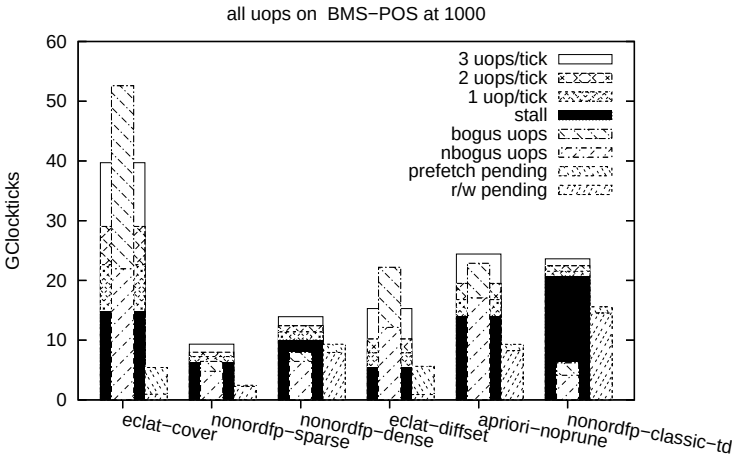
6. REFERENCES

[1] AMD CodeAnalyst^{TM} Performance Analyzer. `http://www.amd.com/us-en/Processors/DevelopWithAMD/0,,30_2252_3604,00.html`.

[2] Intel VTune Performance Analyzers. `http://www.intel.com/software/products/vtune/`.

[3] Bart Goethals. Survey on frequent pattern mining. Technical report, Helsinki Institute for Information Technology, 2003.

[4] Bart Goethals and Mohammed J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining*



Instructions on how to read this diagram:
The height of the wide bars centered around the ticks show the actual run-time (the total clockticks used by the program). The colors/patterns of these bars show how well the program utilized these clockticks: the top-most part shows the amount of clockticks during which three u-ops were executed, while the bottom-most part shows the time during which the program execution was stalled for some reason (i.e., no operations were executed during that clocktick).
The narrow bars centered around the ticks show the total number of u-ops that were executed. The bar is divided into two, the upper part show the bogus u-ops, those u-ops that were speculatively executed on a mispredicted branch, and thus were rolled back. The ratio of the lower-to-upper part of this bar shows the branch prediction inefficiency.
The narrow bars beside the wide ones show the front-side bus activity, the total number of clockticks during whose at least one read/write operation was pending (i.e., data transfer time including memory latency). The upper part of these bars show the time consumed by prefetch reads (when the processor speculatively transfers data from the memory into the cache for further availability), while the lower part shows actual reads or writes. The main difference is that the delivery of data during actual reads and writes presumably stalls the execution pipeline (these are the cache misses). If the ratio of prefetch (top part) to actual wait (bottom part) is high, then a huge amount of cache misses are avoided by the prefetch mechanism, thus achieving a considerable performance gain.

Figure 6: Complex hardware-friendliness diagram of implementations.

FULL EXECUTION PROFILE

	nonordfp dense	nonordfp sparse	nonordfp classic-td	apriori noprune	eclat cover	eclat diffset
Time						
tsc avg	13,946,344,480	9,316,631,548	23,625,371,534	24,418,796,557	39,713,892,504	15,371,783,716
+-	67,047,032	13,729,232	6,967,722	88,581,299	340,152,728	203,764,648
Execution						
Instructions nbogus	5,413,034,086	3,479,408,110	2,782,247,626	12,287,347,990	20,340,660,748	10,265,793,425
Instructions bogus	1,255,528,294	1,320,346,512	1,535,708,709	4,315,850,408	28,792,606,997	8,553,364,068
uops nbogus	6,448,306,722	4,799,136,145	4,139,347,343	17,049,735,428	21,936,853,711	12,134,065,896
uops bogus	1,511,087,197	1,657,126,546	2,077,069,010	5,839,449,756	30,660,104,045	10,071,911,323
uop from TC build	4,316,441	5,555,662	4,998,851	5,408,185	395,994	3,817,971
uop from TC deliver	8,367,199,104	7,030,056,441	8,439,051,063	25,461,603,334	58,919,083,775	24,748,570,113
uop from ROM	213,755,506	327,188,333	213,407,122	434,048,470	121,036,777	120,934,817
FSB/Memory events						
Count of writes	79,566,256	11,341,904	68,774,846	49,640,164	6,812,220	6,522,120
Count of reads (incl. prefetch	168,317,318	39,971,942	251,711,898	146,056,832	100,708,228	112,688,598
Count of r+w+prefetch	246,784,892	51,862,510	320,483,650	198,141,084	107,297,474	119,180,260
Ticks of r+w+prefetch pending	9,333,003,274	2,458,365,028	15,571,190,166	9,318,981,140	5,424,657,616	5,624,807,160
Count of r+w	134,141,834	30,610,382	188,385,644	111,049,656	12,066,130	11,226,952
Ticks of r+w pending	7,947,664,718	2,244,115,972	14,536,000,500	8,239,955,570	970,765,866	900,680,634
store operations (nbogus)	524,358,888	701,508,105	647,941,634	2,569,378,903	455,990,928	344,428,466
load operations (nbogus)	1,379,447,092	1,646,228,087	1,301,423,517	5,118,315,577	4,422,871,852	3,140,679,837
128bit mmx operations	123,298,382	5,068,819	37,679	0	0	0
Branches						
Function calls	13,896,223	16,792,119	13,878,730	134,797,905	5,860,583	5,602,314
Indirect branches	16,747,576	21,078,499	16,725,092	140,985,437	7,545,318	7,443,164
Total conditional branches	1,345,731,870	420,551,853	477,601,235	2,118,233,826	6,991,388,257	3,719,260,359
Mispred cond branches	34,929,891	36,802,712	40,132,062	141,613,646	1,088,360,184	321,259,792
Mispred noncond branches (?)	319,223	453,523	298,023	380,772	2,369	4,861
Stall causes						
MemoryOrderBuffer load (pcs)	830,309,349	202,608,169	5,560,914,552	785,927,299	39,497,734	37,816,201
Lack of store buffer	3,139,504,337	1,444,306,416	885,554,235	2,820,176,466	68,776,266	67,397,197
Memory cancel	2,364,288,764	869,214,080	801,985,760	361,524,915	56,222,156	56,156,928
Split memory access	28,628	28,641	28,637	5,759	871	900
WC buffer evictions	161,435,891	290,548,591	308,327,051	384,695,637	50,740,548	50,014,831
L1 read miss	172,012,034	178,449,046	250,334,607	369,703,235	360,009,247	182,271,597
L2 read miss	57,447,430	32,716,944	129,914,412	76,089,398	25,436,079	25,290,465
Execution histogram						
TickOf uop stall	10,021,440,271	6,240,622,748	20,671,650,354	13,917,119,211	14,801,445,447	5,414,343,311
TickOf uop 1	1,371,043,060	1,023,246,833	859,892,764	2,858,956,464	7,830,923,868	2,458,339,586
TickOf uop 2	1,011,261,551	736,536,448	931,457,183	2,736,975,862	6,418,788,673	2,317,353,459
TickOf uop 3	1,520,275,270	1,323,090,135	1,159,475,923	4,908,240,355	10,668,342,420	5,103,006,616
TickOf nbogus uop stall	10,688,282,384	6,927,514,440	21,520,910,519	16,423,833,026	28,349,705,419	9,709,376,650
TickOf nbogus uop 1	1,206,548,172	891,827,770	736,711,385	2,430,209,501	4,627,072,599	1,607,600,195
TickOf nbogus uop 2	845,852,944	605,173,766	691,985,048	2,082,264,477	2,918,477,521	1,401,800,070
TickOf nbogus uop 3	1,183,336,652	898,980,188	672,869,272	3,484,984,888	3,824,244,869	2,574,266,057
Percent uop stall	71,972	66,935	87,508	56,987	37,264	35,404
Percent uop 1	9,846	10,975	3,640	11,706	19,715	16,074
Percent uop 2	7,262	7,899	3,943	11,207	16,160	15,153
Percent uop 3	10,918	14,191	4,908	20,098	26,859	33,368
Percent nbogus uop stall	76,761	74,302	91,103	67,252	71,374	63,489
Percent nbogus uop 1	8,665	9,565	3,118	9,951	11,649	10,512
Percent nbogus uop 2	6,074	6,490	2,929	8,526	7,347	9,166
Percent nbogus uop 3	8,498	9,642	2,848	14,270	9,628	16,832

Figure 7: Sample execution profile for BMS-POS dataset at support threshold of 1000.

.

Implementations (FIMI'03), volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[5] Gosta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Bart Goethals and Mohammed J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.

[6] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM Press, 2000.

[7] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1968.

[8] Balázs Rácz. nonordfp: An FP-growth variation without rebuilding the FP-tree. In Bart Goethals, Mohammed J. Zaki, and Roberto Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 1. November 2004.

[9] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In Bart Goethals, Mohammed J. Zaki, and Roberto Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 1. November 2004.

[10] Ke Wang, Liu Tang, Jiawei Han, and Junqiang Liu. Top down fp-growth for association rule mining. In *PAKDD '02: Proceedings of the 6th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, pages 334–340, London, UK, 2002. Springer-Verlag.

[11] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In David Heckerman, Heikki Mannila, Daryl Pregibon, Ramasamy Uthurusamy, and Menlo Park, editors, *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 283–296. AAAI Press, 12–15 1997.

APPENDIX

A. SHORT INTRODUCTION TO MODERN PROCESSOR HARDWARE

This section is to give a very short introduction into those properties of modern computing hardware which any programmer optimizing for performance should be aware of.

It is not true that modern processors can execute an instruction (or several instructions) in a single clocktick. The circuits needed to execute an instruction are very deep and complex, but only simple and shallow circuits are able to reach clockspeeds of several GHz. Instruction execution is thus divided into several clockticks (12 for AMD Athlon and 20 for Intel Pentium 4), but to keep the hardware utilized, each of these *pipeline stages* can be processing a different instruction at a particular clocktick. So although as much as 20 clocks might be needed for a particular instruction to finish execution, still (theoretically) in every clocktick an instruction could be finished, giving an average throughput of 1 instruction per clocktick. Furthermore, the pipeline stages are designed so and execution units are multiplied so that theoretically on average more than one instruction can be executed per clocktick.

This pipelined architecture of processors raises several issues, some of which programmers should be aware of, while others should be taken into account by compilers.

Complex instruction set. x86 processors belong to the category of CISC (complex instruction set computer). This means that the elementary instruction the processor can execute can include many operations, like loading a data element from memory, adding another data element to it, and storing back the result into the same memory address. These complex instructions would require more advanced treatment from processors than simple ones. To simplify processor design, instructions are decomposed into one or more μ-operations (or u-ops) and these u-ops are fed into the execution pipeline. Different kind of u-ops use different execution hardware and can be executed concurrently (categories are like: loads, stores, integer arithmetics, floating point/multimedia arithmetics); some units may even be multiplied, like two or even three independent integer ALU/processor.

Data dependency. If there is an instruction in the program that depends on the output of a previous instruction, then its execution cannot begin until that previous instruction is finished.

Branch prediction. When the program reaches a conditional branch, the direction of control flow cannot be known until the branch condition is evaluated. Thus there can be no instructions fed to the pipeline until the condition instruction finishes its execution. This results in wasted processor resources. To enable maximum instruction throughput, processor hardware predicts the condition outcome, and feeds the instructions of the respective branch into the pipeline. However, these instructions will not be committed to the architectural state until the branch condition is evaluated, and if the prediction turns out to be false, these instructions are rolled back, and the pipeline begins with executing the instruction on the correct branch. Branch prediction is based on the previous encounters of the same branch, thus typical branches like loop conditions can be well predicted, as they usually branch towards the inside of a loop, and a misprediction occurs only at the relatively rare condition of exiting that loop.

Another factor that has considerable effects on execution performance is the **efficiency of memory hierarchy**. The basic problem is that processor speeds have increased much faster than main memory access speeds. The difference is so much that today up to 100 clockticks are necessary to load a value from the main memory. Furthermore, memory access is effective only in relatively long bursts of reads and writes. To minimize the latency and data transfer speed effects, processors incorporate relatively small but very fast memories that **cache** the contents of the main memory. There are at least two levels of such cache, reading a data element from the first level (L1) cache takes usually one clocktick, while reading a data element from the second level (L2) cache may take a few clockticks. Typical sizes of these caches are as fol-

lows: few ten KB for L1 cache (16–32 KB in Intel Pentium 4, 64–128 KB in AMD processors), while 512 KB–2 MB for L2 cache. Non-mainstream (value market) processors may have considerably smaller caches.

When a data that is loaded was recently used, there is a high chance of finding it in the cache memory. However, when the program has to process a large amount of data (sequentially), then almost all data accesses will be cache misses, thus the execution engine will wait for the memory, then process the data segment it got, then issue the next memory read request, wait for the memory, etc. The memory interface and the execution units will be alternately idle. To overcome this, the **prefetch** mechanism was introduced to make the memory interface and execution unit concurrently busy. Prefetch operates by loading the data for the next iteration of the processing loop in advance into the fast cache memory so that it will be instantly available when requested by the execution engine. This is implemented in hardware for well predictable memory reads (namely sequential reads), while the programmer has to take care for it in non-sequential memory accesses.

Out of order execution. When an instruction currently in the execution pipeline requires data from the memory that is not found in the cache, it has to wait until the data is loaded from main memory. To keep the execution units of the processor busy, the processor looks ahead for other instructions that have all necessary input data available so that their execution can proceed. Thus only the instructions that depend either on the unavailable data or the result of such instructions are hindered in execution.

Summary. We say that the processor (or the execution pipeline) is *stalled* when there is no instruction whose execution could proceed. This can be caused by various reasons: instructions may depend on the completion of another instruction (by using its output data as input); instructions may require data from main memory that is not available in the fast cache memory, thus they have to wait until the memory access cycle completes; there may have been a mispredicted conditional branch when the speculatively executed instructions had to be flushed from the pipeline; or there may be another resource constraint (lack of some sort of temporary buffers, like store buffers, register renaming buffers, etc.). Execution speed of an algorithm (instruction flow) does not only depend on the number of instructions it contains, but also on the count and severity of the stalls that the particular instruction flow causes on the hardware used.

On the Effectiveness and Efficiency of Computing Bounds on the Support of Item-Sets in the Frequent Item-Sets Mining Problem

Bassem Sayrafi, Dirk Van Gucht [*] and Paul W. Purdom
Computer Science Department
Indiana University
Lindley Hall 215
Bloomington, IN 47405, USA
{bsayrafi,vgucht,pwp}@cs.indiana.edu

ABSTRACT

We study the relative effectiveness and the efficiency of computing support-bounding rules that can be used to prune the search space in algorithms to solve the frequent item-sets mining problem (FIM). We develop a formalism wherein these rules can be stated and analyzed using the concept of differentials and density functions of the support function. We derive a general bounding theorem, which provides lower and upper bounds on the supports of item-sets in terms of the supports of their subsets. Since, in general, many lower and upper bounds exists for the support of an item-set, we show how to the best bounds. The result of this optimization shows that the best bounds are among those that involve the supports of all the strict subsets of an item-set of a particular size q. These bounds are determined on the basis of so called q-rules. In this way, we derive the bounding theorem established by Calders [5]. For these types of bounds, we consider how they compare relative to each other, and in so doing determine the best bounds. Since determining these bounds is combinatorially expensive, we study heuristics that efficiently produce bounds that are usually the best. These heuristics always produce the best bounds on the support of item-sets for basket databases that satisfies independence properties. In particular, we show that for an item-set I determining which bounds to compute that lead to the best lower and upper bounds on $\texttt{freq}(I)$ can be done in time $O(|I|)$. Even though, in practice, basket databases do not have these independence properties, we argue that our analysis carries over to a much larger set of basket databases where local "near" independence hold. Finally, we conduct an experimental study using real baskets databases, where we compute upper bounds in the context of generalizing the Apriori algorithm. Both the analysis and the study confirm that the q-rule (q odd and larger than 1) will almost always do better than the 1-rule (Apriori rule) on large dense baskets databases. Our experiment reveal that on these baskets databases, the 3-rule prunes almost 100% of the search space while, the 1-rule prunes 96% of the search space in the early stages of the algorithm. We also observe a reduction in wasted effort when applying the 3-rule to sparse baskets databases. In addition, we give experimental evidence that the combined use of the lower and upper bounds determine the exact support of many frequent item-sets without counting.

[*]The first two authors were supported by NSF Grant IIS-0082407.

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

1. INTRODUCTION

We consider the relative effectiveness of various support bound rules for the *frequent item-sets mining problem* (FIM) [1, 2], as well as the problem of efficiently computing the best support bounds. The FIM problem is the following: given a set of items $\mathcal{I}$, a list $\mathcal{B}$ of subsets (*baskets*) of $\mathcal{I}$, and a nonnegative integer threshold $k \geq 0$, determine the *frequency status* for each subset of $\mathcal{I}$, that is, determine for each subset of $\mathcal{I}$ whether it is contained in at least k of the baskets in $\mathcal{B}$. A subset that satisfies (violates) this frequency condition is called a *frequent item-set* (*infrequent item-set*, respectively). Given, a threshold k, the problem asking whether there exists a frequent item-set of a certain size in a given database has been shown to be NP-complete [8].

The frequency status of an item-set I can be determined in two ways. *Counting*: count the number of baskets in $\mathcal{B}$ that contain I, and compare this count with the threshold k; or *Deduction*: infer I's frequency status from the (known) frequency status of other item-sets. The best known deduction methods for the FIM problem are based on the *monotonicity property* and, its counterpart, the *anti-monotonicity property*. The monotonicity property states that if an item-set I has a subset that is infrequent, the I is infrequent, and the anti-monotonicity property states its opposite: if an item-set I has a superset that is frequent, then I is frequent. Good examples of how these properties have been harnessed exists in the Apriori, the FP-growth, and the Eclat algorithms [2, 9, 15]. Given an nonempty item-set I whose frequency status is unknown, the Apriori Algorithm consults the frequency status of each of subsets of size $|I| - 1$. If one of these subsets is infrequent, the algorithm deduces that I is infrequent, otherwise, the algorithm determines the frequency status of I by counting.

We will determine support bounding rules which are based on properties of the support (frequency) function beyond just the monotonicity property. The bounding rules can be used in any algorithms

for solving the frequent item-set mining problem, thus our analysis are algorithm independent. We develop a general bounding theorem to approximate the support of a set from the supports of some of its subsets. We construct this theorem through the use of differentials and density functions associated with support functions.

The notion of differentials was consider by the first two authors in the study of the implication problem of differential constraints [13]. To illustrate where these inequalities arise in the FIM problem, consider a list of baskets $\mathcal{B}$ over some set of items $\mathcal{I}$. The *support function* $\mathtt{supp}$ associated with $\mathcal{B}$ gives for each item-set $I \subseteq \mathcal{I}$, the value $\mathtt{supp}(I)$ which is number of times that I is contained in the baskets in $\mathcal{B}$. Given $\mathtt{supp}$, we can reason about $\mathcal{B}$ satisfying certain types of inequalities. For example, $\mathtt{supp}(I) \geq 0$ states that the support of item-set I is nonnegative. More generally, if l and u are values in the interval $[0, 1]$, then the inequality $l|\mathcal{B}| \leq \mathtt{supp}(I) \leq u|\mathcal{B}|$ states that $\mathcal{B}$ has at least $l|\mathcal{B}|$, but not more than $u|\mathcal{B}|$, baskets containing I. These types of support inequalities were studied by Calders and Paredaens [5, 7]. Here we study inequalities of a different type. Consider item-sets K, L_1, and L_2 of $\mathcal{I}$. The inequality $\mathtt{supp}(K) \geq \mathtt{supp}(I)$ states that K occurs at least as frequently as I in $\mathcal{B}$. A more subtle example satisfied by $\mathtt{supp}$ is the inequality $\mathtt{supp}(K) - \mathtt{supp}(K \cup L_1) \geq \mathtt{supp}(K \cup L_2) - \mathtt{supp}(K \cup L_1 \cup L_2)$, which can be proved by an inclusion-exclusion argument. This inequality can be used as a lower bound for $\mathtt{supp}(K \cup L_1 \cup L_2)$ [11]. Observe that we can write these two last inequalities as finite difference equations:

$$\begin{aligned}&\mathtt{supp}(K) - \mathtt{supp}(K \cup L_1) \geq 0.\\ &(\mathtt{supp}(K) - \mathtt{supp}(K \cup L_1))\\ &\quad -(\mathtt{supp}(K \cup L_2) - \mathtt{supp}(K \cup L_1 \cup L_2)) \geq 0.\end{aligned}$$

One of the main results of this paper is a general support bounding theorem which we derive through the use of the differentials of support functions. This is done in Section 3. From this theorem stems a class of support bounding rules that can be used in the deduction of the support status of an item-set. Within this class of rules, we derive special rules (so called q-rules) that lead to the best support bounds. The class of q-rules was previously considered by Calders in his work on deduction rules for the FIM problem [6, 5]. Since finding the best support bounds on the support of an item-set is combinatorially expensive, we propose heuristics that can lead to good approximations of these best bounds and that can be computed efficiently. In Section 4, a detailed analysis of the bounding theorem reveals that the anti-monotonicity property will outperform other bounds except in certain situations where the data is highly frequent. In Section 5, we give a complete solution for the problem of determining which lower and upper bounds are best in the context of basket databases that satisfy independence properties. In particular, we show that for an item-set I, this can be done in $O(|I|)$ (Proposition 4.3). Given these theoretical insights, we develop heuristics based on these ideas (Section 5), and provide experimental results where the heuristics are used (Section 6). The results of these experiments underscore that the theoretical results predict well what happens on real-world basket databases, and that introducing these heuristics in FIM algorithms leads to improved algorithms and that the overhead incurred is small.

$\mathcal{I}$	nonempty finite set set of *items*
I, J, K, L	subsets of $\mathcal{I}$ *item-sets*
$\mathcal{L}$	set of item-sets of $\mathcal{I}$
$[X, Y]$	the set $\{U \mid X \subseteq U \subseteq Y\}$
$A_1 \ldots A_n$	the set $\{A_1, \ldots, A_n\}$
$X \subset Y$	X is a *strict* subset of Y
$\overline{X}$	the set $\mathcal{I} - X$

Table 1: Notations

2. PRELIMINARIES

We review the definitions of the density, support, and frequency functions associated with a basket database. In addition, we will introduce the notion of the *differential* of the support function. Differentials are at the core of deriving bounds on the support (frequency) of item-sets. For ease of reference, in Table 2, we collect some notations used in the paper.

2.1 Density, support, and frequency functions of basket databases

DEFINITION 2.1. *Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database. The* density*, the* support*, and the* frequency *functions associated with $\mathcal{D}$ are defined such that for each $I \subseteq \mathcal{I}$,*

$$\begin{aligned}\mathtt{dens}(I) &= |\{i \mid I = \mathcal{B}[i]\}|\\ \mathtt{supp}(I) &= |\{i \mid I \subseteq \mathcal{B}[i]\}|\\ \mathtt{freq}(I) &= \tfrac{\mathtt{supp}(I)}{|\mathcal{B}|}\end{aligned}$$

The density and support functions of a basket database are related in the following way ($\mathtt{dens}$ is the *Möbius inverse* of $\mathtt{supp}$):

$$\mathtt{supp}(I) = \sum_{I \subseteq J \subseteq \mathcal{I}} \mathtt{dens}(J) \quad (1)$$

$$\mathtt{dens}(I) = \sum_{I \subseteq J \subseteq \mathcal{I}} (-1)^{|J|-|I|} \mathtt{supp}(J) \quad (2)$$

2.2 Differentials of support functions

Reconsider the following inequalities discussed in the introduction:

$$\mathtt{supp}(K) \geq 0 \quad (3)$$

$$\mathtt{supp}(K) - \mathtt{supp}(K \cup L_1) \geq 0 \quad (4)$$

$$\mathtt{supp}(K) - \mathtt{supp}(K \cup L_1) - \mathtt{supp}(K \cup L_2) + \mathtt{supp}(K \cup L_1 \cup L_2) \geq 0 \quad (5)$$

We can write these inequalities in a single format as follows:

$$\sum_{\mathcal{J} \subseteq \mathcal{L}} (-1)^{|\mathcal{J}|} \mathtt{supp}(K \cup \bigcup_{J \in \mathcal{J}} J) \geq 0,$$

where for inequality (3), $\mathcal{L} = \emptyset$, for inequality (4), $\mathcal{L} = \{L_1\}$, and for inequality (5), $\mathcal{L} = \{L_1, L_2\}$. This leads to the definition of differentials first considered in [12, 13, 14].

DEFINITION 2.2. *Let $\mathcal{L}$ be a set of item-sets of $\mathcal{I}$, and let $\mathtt{supp}$ be the support function of a basket database $\mathcal{D} = (\mathcal{I}, \mathcal{B})$. The* $\mathcal{L}$-differential *of $\mathtt{supp}$, denoted $D^{\mathcal{L}}_{\mathtt{supp}}$, is the function from $2^{\mathcal{I}}$ into $\mathbf{N}$, such that for $K \subseteq \mathcal{I}$,*

$$D^{\mathcal{L}}_{\mathtt{supp}}(K) = \sum_{\mathcal{J} \subseteq \mathcal{L}} (-1)^{|\mathcal{J}|} \mathtt{supp}(K \cup \bigcup_{J \in \mathcal{J}} J).$$

EXAMPLE 2.3. Let $\mathcal{I} = \{A, B, C, D\}$. Then,

$$D^{\{B,CD\}}_{\mathtt{supp}}(A) = \mathtt{supp}(A) - \mathtt{supp}(AB) - \mathtt{supp}(ACD) + \mathtt{supp}(ABCD).$$

As shown in [13], density functions and differentials are related. The concepts of *witness sets* and *lattice decompositions* are crucial in establishing their relationship.

DEFINITION 2.4. *A set W is a* witness set *of $\mathcal{L}$ ($\mathcal{L}$ is a set of some subsets of $\mathcal{I}$) if (1) $W \subseteq \bigcup_{L\in\mathcal{L}} L$ and (2) W has a nonempty intersection with each set in $\mathcal{L}$: $\forall L \in \mathcal{L} : W \cap L \neq \emptyset$. The set of all witness sets of $\mathcal{L}$ is denoted by $\mathcal{W}(\mathcal{L})$.*

Let K be a subset of $\mathcal{I}$. The lattice decomposition $\mathbf{L}(K, \mathcal{L})$ *of the pair $(K, \mathcal{L})$ is defined as follows:*

$$\mathbf{L}(K, \mathcal{L}) = \bigcup_{W\in\mathcal{W}(\mathcal{L})} [K, \overline{W}].$$

As shown in [13], the relationship between differentials and density function can now be formulated as follows:

THEOREM 2.5 (DIFFERENTIAL DECOMPOSITION THEOREM). Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database, let K be a subset of $\mathcal{I}$, and let $\mathcal{L}$ be a set of subsets of $\mathcal{I}$. Then,

$$D^{\mathcal{L}}_{\mathtt{supp}}(K) = \sum_{J\in\mathbf{L}(K,\mathcal{L})} \mathtt{dens}(J). \quad (6)$$

Observe that, since dens is a nonnegative function, D is also a nonnegative function.

EXAMPLE 2.6. Reconsider Example 2.3. Then,

$$\begin{aligned}
\mathcal{W}(\{B, CD\}) &= \{BC, BD, BCD\}\\
\mathbf{L}(A, \{B, CD\}) &= [A, \overline{BC}] \cup [A, \overline{BD}] \cup [A, \overline{BCD}]\\
&= [A, AD] \cup [A, AC] \cup [A]\\
&= \{A, AC, AD\}\\
D^{\{B,CD\}}_{\mathtt{supp}}(A) &= \mathtt{dens}(A) + \mathtt{dens}(AC) + \mathtt{dens}(AD).
\end{aligned}$$

3. SUPPORT BOUNDING THEOREMS

We use the results obtained in Section 2 to obtain various support bounding theorems. In particular, we are concerned with obtaining lower and upper bounds on $\mathtt{supp}(I)$ for some item-set $I \subseteq \mathcal{I}$ in terms of the support values of I's subsets. After stating a general bounding theorem, we will derive a bounding theorem which gives the best lower and upper bounds on $\mathtt{supp}(I)$. This bounding theorem was first formulated by Calders [5].

3.1 The support bounding theorem

For a given item-set $I \subseteq \mathcal{I}$, we will consider the set of all $(K, \mathcal{L})$ pairs such that $I = K \cup \bigcup_{L\in\mathcal{L}} L$. For each such pair, we will derive a lower (upper) bound on $\mathtt{supp}(I)$ when $|\mathcal{L}|$ is even (odd, respectively). To state these bounds, we first define the following sets:

DEFINITION 3.1. *Let $I \subseteq \mathcal{I}$. The sets* $\mathtt{Pairs}(I)$, $\mathtt{EvenPairs}(I)$, $\mathtt{OddPairs}(I)$, *as follows:*

$$\begin{aligned}
\mathtt{Pairs}(I) &= \{(K, \mathcal{L}) \mid I = K \cup \bigcup_{L\in\mathcal{L}} L\}\\
\mathtt{EvenPairs}(I) &= \{(K, \mathcal{L}) \in \mathtt{Pairs}(I) \mid |\mathcal{L}| \text{ is even}\}\\
\mathtt{OddPairs}(I) &= \{(K, \mathcal{L}) \in \mathtt{Pairs}(I) \mid |\mathcal{L}| \text{ is odd}\}
\end{aligned}$$

For a pair $(K, \mathcal{L}) \in \mathtt{Pairs}(I)$, we have, by Definition 2.2 and Theorem 2.5 that

$$(-1)^{|\mathcal{L}|}\left[\mathtt{supp}(I) - \sum_{\mathcal{J}\subset\mathcal{L}} (-1)^{|\mathcal{L}|-|\mathcal{J}|-1}\mathtt{supp}(K \cup \bigcup_{J\in\mathcal{J}} J)\right] = \sum_{J\in\mathbf{L}(K,\mathcal{L})} \mathtt{dens}(J). \quad (7)$$

Since dens is a nonnegative function, Eqn. 7 implies that following inequality:

$$(-1)^{|\mathcal{L}|}\left[\mathtt{supp}(I) - \sum_{\mathcal{J}\subset\mathcal{L}} (-1)^{|\mathcal{L}|-|\mathcal{J}|-1}\mathtt{supp}(K \cup \bigcup_{J\in\mathcal{J}} J)\right] \geq 0. \quad (8)$$

Inequality 8, implies upper and lower bounds on $\mathtt{supp}(I)$.

DEFINITION 3.2. *Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database. The* bounding function B *associated with $\mathcal{D}$ is,*

$$B(K, \mathcal{L}) = \sum_{\mathcal{J}\subset\mathcal{L}} (-1)^{|\mathcal{L}|-|\mathcal{J}|-1}\mathtt{supp}(K \cup \bigcup_{J\in\mathcal{J}} J).$$

By Eqn. 7, we can reformulate $B(\mathcal{L}, K)$ as follows:

$$B(K, \mathcal{L}) = \mathtt{supp}(I) - \sum_{J\in\mathbf{L}(K,\mathcal{L})} \mathtt{dens}(J) \quad |\mathcal{L}| \text{ is even} \quad (9)$$

$$= \mathtt{supp}(I) + \sum_{J\in\mathbf{L}(K,\mathcal{L})} \mathtt{dens}(J) \quad |\mathcal{L}| \text{ is odd.} \quad (10)$$

Thus, $B(\mathcal{L}, K)$ is a lower (upper) bound on $\mathtt{supp}(I)$ when $|\mathcal{L}|$ is even (odd, respectively). The error in the bound is given by the summation over the density function in the above equations. Since the summation is nonnegative we arrive at the following bounding theorem.

THEOREM 3.3 (SUPPORT BOUNDING THEOREM). Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database and let I be a subset of $\mathcal{I}$. Then, for each $(K, \mathcal{L}) \in \mathtt{Pairs}(I)$,

$$\begin{aligned}
B(K, \mathcal{L}) &\leq \mathtt{supp}(I) \quad \text{if } (K, \mathcal{L}) \in \mathtt{EvenPairs}(I)\\
B(K, \mathcal{L}) &\geq \mathtt{supp}(I) \quad \text{if } (K, \mathcal{L}) \in \mathtt{OddPairs}(I)
\end{aligned}$$

3.2 The best support bounds theorem

We now consider the problem of finding the best bounds on $\mathtt{supp}(I)$ derivable from $\mathtt{Pairs}(I)$. In fact, we will show that the best bounds come from pairs of the form $(K, \{\{l\} \mid l \in I - K\})$, which we will denote as $(K, \mathbf{I} - \mathbf{K})$.

By Theorem 3.3, for a pair $(K, \mathcal{L}) \in \mathtt{Pairs}(I)$, $B(K, \mathcal{L})$ is either a lower or an upper bound on $\mathtt{supp}(I)$. Some of these pairs,

however, lead to trivial bounds and we will eliminate these from further consideration. A pair $(K, \mathcal{L})$ is *trivial* if there exists an $L \in \mathcal{L}$ such that $L \subseteq K$. For such a pair, it is easy to verify that $B(K, \mathcal{L}) = \mathtt{supp}(I)$. It will be useful to introduce the following subsets of $\mathtt{Pairs}(I)$.

DEFINITION 3.4. *Let $I \subseteq \mathcal{I}$. Then,*

$$\begin{aligned}
\mathtt{NonTrivPairs}(I) &= \{(K, \mathcal{L}) \in \mathtt{Pairs} \mid \forall L \in \mathcal{L} : L \not\subseteq K\} \\
\mathtt{NonTrivEvenPairs}(I) &= \mathtt{NonTrivPairs}(I) \cap \mathtt{EvenPairs}(I) \\
\mathtt{NonTrivOddPairs}(I) &= \mathtt{NonTrivPairs}(I) \cap \mathtt{OddPairs}(I)
\end{aligned}$$

$$\begin{aligned}
\mathtt{AtomicPairs}(I) &= \{(K, \mathbf{I} - \mathbf{K}) \mid K \subseteq I\} \\
\mathtt{AtomicEvenPairs}(I) &= \mathtt{AtomicPairs}(I) \cap \mathtt{EvenPairs}(I) \\
\mathtt{AtomicOddPairs}(I) &= \mathtt{AtomicPairs}(I) \cap \mathtt{OddPairs}(I)
\end{aligned}$$

THEOREM 3.5. Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database and let I be a $\mathcal{I}$. Then,

(1) $\max(\{B(K, \mathcal{L}) \mid (K, \mathcal{L}) \in \mathtt{NonTrivEvenPairs}(I)\}) = \max(\{B(K, \mathcal{L}) \mid (K, \mathcal{L}) \in \mathtt{AtomicEvenPairs}(I)\})$

(2) $\min(\{B(K, \mathcal{L}) \mid (K, \mathcal{L}) \in \mathtt{NonTrivOddPairs}(I)\}) = \min(\{B(K, \mathcal{L}) \mid (K, \mathcal{L}) \in \mathtt{AtomicOddPairs}(I)\})$

Proof: We will establish (1). (The case for (2) is analogous.) Let $K \subseteq I$ with $|I-K|$ even and let $(K, \mathcal{L}) \in \mathtt{NonTrivEvenPairs}(I)$. We will show that $B(K, \mathcal{L}) \leq B(K, \mathbf{I} - \mathbf{K})$. By Eqn. 9, this is equivalent to showing that

$$\sum_{\mathbf{L}(K, \mathcal{L})} \mathtt{dens}(U) \geq \sum_{\mathbf{L}(K, \mathbf{I}-\mathbf{K})} \mathtt{dens}(U).$$

This inequality is true when $\mathbf{L}(K, \mathbf{I} - \mathbf{K}) \subseteq \mathbf{L}(K, \mathcal{L})$, or equivalently, when $\mathcal{W}(\mathbf{I} - \mathbf{K}) \subseteq \mathcal{W}(\mathcal{L})$. From the definition of witness sets (Definition 2.4), it follows that $\mathcal{W}(\mathbf{I} - \mathbf{K}) = \{I - K\}$. Thus, all we need to show is that $I - K \in \mathcal{W}(\mathcal{L})$. In particular, we must show that (1) $I - K \subseteq \bigcup_{L \in \mathcal{L}} L$, and (2) $\forall L \in \mathcal{L} : L \cap (I - K) \neq \emptyset$. To show condition (1), assume that there exists an $l \in I - K$ such that $l \notin \bigcup_{L \in \mathcal{L}} L$. But, since $K \cup \bigcup_{L \in \mathcal{L}} L = I$, l must be in K, but that contradicts $l \in I - K$. To show condition (2), assume that there exists an $L \in \mathcal{L}$ such that $L \cap (I - K) = \emptyset$. But this implies $L \subseteq K$, a contradiction since $(K, \mathcal{L})$ is a nontrivial pair.

We can now use Theorem 3.5 to establish the following theorem which states how to derive the best bounds:

THEOREM 3.6 (BEST SUPPORT BOUNDS THEOREM). Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database and let I be a subset of $\mathcal{I}$. Then,

$$\begin{aligned}
\max(\{B(K, \mathcal{L}) \mid (K, \mathcal{L}) \in \mathtt{AtomicEvenPairs}(I)\}) &\leq \mathtt{supp}(I) \\
\min(\{B(K, \mathcal{L}) \mid (K, \mathcal{L}) \in \mathtt{AtomicOddPairs}(I)\}) &\geq \mathtt{supp}(I)
\end{aligned}$$

Calders [4] was the first to prove the Best Support Bounds Theorem (Theorem 3.6) which, we have shown to be a consequence of the Support Bounding Theorem (Theorem 3.3). He applied the Best Support Bounds Theorem to the problem of mining non-derivable item-sets [4]. Specifically, Calders was interested in item-sets I, where the lower bound on $\mathtt{supp}(I)$ is *equal* to the upper bound on $\mathtt{supp}(I)$. In that case $\mathtt{supp}(I)$ need not be computed by counting it in the basket database. The effectiveness of this technique in the context of mining frequent item-sets was empirically observed in [6].

An alternative way to state the Best Support Bounds Theorem (Theorem 3.6) is in terms of the cardinality of $I - K$. In particular, the best lower (upper) bound on $\mathtt{supp}(I)$ is

$$\max(\{B(K, \mathbf{I} - \mathbf{K}) \mid q = |I - K| \,\&\, q \text{ is even}\}) \quad (11)$$

$$\min(\{B(K, \mathbf{I} - \mathbf{K}) \mid q = |I - K| \,\&\, q \text{ is odd}\}). \quad (12)$$

These bounds lead to the notion of q-rules, where $q \in [0, |I|]$. The q-rule for I is the following statement, relating $\mathtt{supp}(I)$ with the best bounds that are obtainable from sets $K \subseteq I$ for which $|I - K| = q$:

$$\mathtt{supp}(I) \geq B(K, \mathbf{I} - \mathbf{K}) \qquad q \text{ is even} \quad (13)$$

$$\mathtt{supp}(I) \leq B(K, \mathbf{I} - \mathbf{K}) \qquad q \text{ is odd.} \quad (14)$$

For example, the 0-rule and the 2-rule state the following (let i_1 and i_2 be two distinct elements in I):

$q = 0$, $K = I$

$$\mathtt{supp}(I) \geq 0.$$

$q = 2$, $K = I - \{i_1, i_2\}$

$$\mathtt{supp}(I) \geq \mathtt{supp}(K \cup \{i_1\}) + \mathtt{supp}(K \cup \{i_2\}) - \mathtt{supp}(K).$$

And, the 1-rule and the 3-rule state the following (let i_1, i_2 and i_3 be distinct elements in I):

$q = 1$, $K = I - \{i_1\}$

$$\mathtt{supp}(I) \leq \mathtt{supp}(K).$$

$q = 3$, $K = I - \{i_1, i_2, i_3\}$

$$\begin{aligned}
\mathtt{supp}(I) \leq\ & \mathtt{supp}(K \cup \{i_1, i_2\}) + \mathtt{supp}(K \cup \{i_1, i_3\}) \\
& + \mathtt{supp}(K \cup \{i_2, i_3\}) - \mathtt{supp}(K \cup \{i_1\}) \\
& - \mathtt{supp}(K \cup \{i_2\}) - \mathtt{supp}(K \cup \{i_3\}) + \mathtt{supp}(K).
\end{aligned}$$

The q-rules highlight two issues that are relevant in the computation of the best bounds on $\mathtt{supp}(I)$. The first is that if we just consider sets $K \subseteq I$ such that $|I - K|$ is fixed and even (odd), what is an efficient way to compute the best bound obtainable on $\mathtt{supp}(I)$ from these K's? The second is when we have K and K' such that both $|I - K|$ and $|I - K'|$ are even (odd) but of different sizes, which of them leads to the best bound on $\mathtt{supp}(I)$? The first issue will be addressed in the next subsection. The second issue is more advantageously addressed by analyzing it for basket databases that satisfy certain conditions (such as independence, see Section 4). This is because the relative effectiveness of the rules is determined by the comparing the sum $\sum_{K \subseteq J \subseteq \overline{I-K}} \mathtt{dens}(J)$ to the sum $\sum_{K' \subseteq J \subseteq \overline{I-K'}} \mathtt{dens}(J)$. Since the values of these sums are expressed in terms of the density function of the basket database, their comparison is entirely controlled by the properties of the data. Thus, to gain a better understanding of this issue, we need to make certain assumptions on this data. This will be addressed in Section 4.

3.3 Computing and approximating best support bounds

From the Best Support Bounds Theorem (Theorem 3.5), it follows that the best lower bound on $\texttt{supp}(I)$ is given by $\max(\{B(K,\mathcal{L}) \mid (K,\mathcal{L}) \in \texttt{AtomicEvenPairs}(I)\})$, and the best upper bounds is given by $\min(\{B(K,\mathcal{L}) \mid (K,\mathcal{L}) \in \texttt{AtomicOddPairs}(I)\})$. Determining these bounds can be computationally expensive. A naive complexity analysis gives that this can be done in PSPACE. However, we can consider heuristics that would approximate the best bounds, but that may lead to a more efficient search for good (but not necessarily optimum) candidates K for obtaining bounds on $\texttt{supp}(I)$. The following proposition is insightful in this regard.

PROPOSITION 3.7. Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database and let I be a subset of $\mathcal{I}$. Then, determining the best lower (upper) bound on $\texttt{supp}(I)$ can be done by minimizing the sum

$$\sum_{K \subseteq J \subseteq \overline{I-K}} \texttt{dens}(J)$$

over all subsets K of I.

Proof: By Theorem 3.5, the best lower (upper) bound on $\texttt{supp}(I)$ is obtained by maximizing (minimizing) $B(K, \{\{l\} \mid l \in I-K\})$. By Eqn.9 (Eqn.10), this is equivalent to minimizing $\sum_{K \subseteq J \subseteq \overline{I-K}} \texttt{dens}(J)$ over all $K \subseteq I$.

The significance of Proposition 3.7 is that it allows us to determine the error of the best bounds in terms of minimizing a sum over values of the density function. Since, as stated above, this minimization is combinatorially expensive, it is useful to find good approximations for these sums. In particular, we are interested in heuristically determining a $K \subseteq I$ that would be a good candidate for computing the best bounds on $\texttt{supp}(I)$. In this regard, we can use the following inequality:

$$\texttt{supp}(K) = \sum_{K \subseteq J \subseteq \mathcal{I}} \texttt{dens}(J) \geq \sum_{K \subseteq J \subseteq \overline{I-K}} \texttt{dens}(J), \quad (15)$$

and then use the heuristic that K can be selected by finding the smallest value for $\texttt{supp}(K)$, and then use that K to compute a hopefully good bound on $\texttt{supp}(I)$. (In Section 5, we introduce the heuristic called Hq based on these ideas.)

Another technique to more efficiently compute an approximation of the best bounds is to use a specific q-rule. In that case, only K's such that $|I - K| = q$ need to be considered. Thus rather than considering all possible $2^{|I|}$ K sets, only $\binom{|I|}{q}$ (i.e. a polynomial of degree q in $|I|$) K sets need to be considered. (In Section 5, we consider applying this technique for $q = 1$, 2, and 3 and we label these cases by R1, R2, and R3, respectively.)

4. ANALYSIS FOR BASKET DATABASES WITH INDEPENDENCIES

We will now study the relative effectiveness and efficiency of computing lower and upper bounds, as stated in of Theorem 3.6, for basket databases that satisfy a condition of independence. Specially, we say that a basket database $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ satisfies the *independence property* if for each K and L subsets of $\mathcal{I}$,

$$\texttt{supp}(K \cup L)\texttt{supp}(K \cap L) \quad = \quad \texttt{supp}(K)\texttt{supp}(L), \quad (16)$$

or, equivalently, by the definition of freq,

$$\texttt{freq}(K \cup L)\texttt{freq}(K \cap L) \quad = \quad \texttt{freq}(K)\texttt{freq}(L). \quad (17)$$

In the rest of the paper, we will focus on freq rather than supp because certain expressions are more transparent in terms of freq. We use $\texttt{freq}(i)$ as a shorthand for $\texttt{freq}(\{i\})$. With this notation, the independence property can be shown to be equivalent to the following statement: for each $I \subseteq \mathcal{I}$,

$$\texttt{freq}(I) \quad = \quad \prod_{i \in I} \texttt{freq}(i) \quad (18)$$

Using the frequency function we can represent the ideas from Section 3 as follows.

DEFINITION 4.1. *Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database. For each pair $(K, \mathcal{L})$, the* $(K, \mathcal{L})$-bound*, $B_{\texttt{freq}}(K, \mathcal{L})$, on $\texttt{freq}(I)$ is defined such that $B_{\texttt{freq}}(K, \mathcal{L}) = \sum_{\mathcal{J} \subset \mathcal{L}} (-1)^{|\mathcal{L}|-|\mathcal{J}|-1} \texttt{freq}(K \cup \bigcup_{J \in \mathcal{J}} J)$.*

THEOREM 4.2. Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database and let I be a subset of $\mathcal{I}$. Then,

$$\begin{aligned} \max(\{B_{\texttt{freq}}(K,\mathcal{L}) \mid (K,\mathcal{L}) \in \texttt{AtomicEvenPairs}(I)\}) &\leq \texttt{freq}(I) \\ \min(\{B_{\texttt{freq}}(K,\mathcal{L}) \mid (K,\mathcal{L}) \in \texttt{AtomicOddPairs}(I)\}) &\geq \texttt{freq}(I) \end{aligned}$$

Furthermore, in analogy with Proposition 3.7, determining the best lower (upper) bound on $\texttt{freq}(I)$ can be done by minimizing the sum $\sum_{K \subseteq J \subseteq \overline{I-K}} \texttt{dens}(J)$ over all subsets K of I.

In the case where the basket database $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ satisfies the independence condition, the bound $B_{\texttt{freq}}(K, \mathbf{I} - \mathbf{K})$ is such that,

$$B_{\texttt{freq}}(K, \mathbf{I} - \mathbf{K}) = \texttt{freq}(K) \sum_{J \subset I-K} (-1)^{|I-K|-|J|-1} \texttt{freq}(J) \quad (19)$$

In the following subsections, we present an analysis for determining lower an upper bounds on $\texttt{freq}(I)$ for basket databases that satisfy the independence condition. In Subsection 4.1, we study the relative effectiveness of different q-rules on the quality of the bounds. In Subsection 4.2, within the context of a fixed q-rule, we examine different heuristics to efficiently determine a K such that $B_{\texttt{freq}}(K, \mathbf{I} - \mathbf{K})$ is the best possible bound.

4.1 Inter-analysis between different q-rules

From Eqn. 13 and Eqn. 14, we learned that we can enumerate different lower bounds and upper bounds on $\texttt{freq}(I)$ depending on the cardinality q of $I - K$. We are interested in studying the relative effectiveness of different q rules to obtain good bounds. To that end, we consider a comparison between a q-rule and a $q + 2$-rule where q is even (odd). Specifically, let K be such that $|I - K| = q$ and let i_1 and i_2 be two different elements of K, and let K' denote the set $K - \{i_1, i_2\}$. Thus $I - K' = (I - K) \cup \{i_1, i_2\}$ and therefore $|I - K'| = q + 2$. By Eqn. 19, comparing $B_{\texttt{freq}}(K, \mathbf{I} - \mathbf{K})$

and $B_{\texttt{freq}}(K', \mathbf{I} - \mathbf{K'})$ leads to the comparison:

$$\texttt{freq}(K)(-1)^{|I-K|-1} \sum_{J \subset I-K} (-1)^{|J|}\texttt{freq}(J) \quad vs \quad \texttt{freq}(K')(-1)^{|I-K'|-1} \sum_{J' \subset I-K'} (-1)^{|J'|}\texttt{freq}(J')$$

After some algebraic simplification, this is equivalent to,

$$(-1)^{|I-K|-1}\texttt{freq}(K)\left[\prod_{j \in I-K}(1-\texttt{freq}(j))\right] \quad vs \quad (-1)^{|I-K|-1}\texttt{freq}(K')\left[\prod_{j \in (I-K)\cup i_2 i_3}(1-\texttt{freq}(j))\right],$$

which, after further algebraic reductions, yields the following comparison:

$$(-1)^{|I-K|-1}\texttt{freq}(K')\left[\prod_{j \in I-K}(1-\texttt{freq}(j))\right] [\texttt{freq}(i_1)+\texttt{freq}(i_2)-1] \quad vs \quad 0. \quad (20)$$

This shows that for q even (q odd), the bound obtained using the $q+2$-rule will be better than the bound obtained using the q-rule if and only $\texttt{freq}(i_1)+\texttt{freq}(i_2) > 1$.

4.2 Intra-analysis within a fixed q-rule

Within a context of a fixed q-rule, we now consider the problem of determining a K such that $B_{\texttt{freq}}(K, \mathbf{I}-\mathbf{K})$ is the best possible bound for that q. To that end, let X be a subset of I, and let i and i' be two different elements in $I - X$. Let $K = X \cup \{i\}$ and let $K' = X \cup \{i'\}$. Furthermore, assume that $|I - K| = q$ (clearly, therefore $|I - K'| = q$). We are interested in comparing $B_{\texttt{freq}}(K, \mathbf{I}-\mathbf{K})$ and $B_{\texttt{freq}}(K', \mathbf{I}-\mathbf{K}')$. By Eqn. 19, this leads to the following comparison:

$$(-1)^{|I-K|-1}\texttt{freq}(X)\texttt{freq}(i)\left[\prod_{j \in I-K}(1-\texttt{freq}(j))\right] \quad vs \quad (-1)^{|I-K'|-1}\texttt{freq}(X)\texttt{freq}(i')\left[\prod_{j \in I-K'}(1-\texttt{freq}(j))\right],$$

which, after algebraic simplification is equivalent to the following comparison:

$$(-1)^{|I-K|-1}\texttt{freq}(X)\left[\prod_{j \in I-(K\cup K')}(1-\texttt{freq}(j))\right] \left[\texttt{freq}(i)-\texttt{freq}(i')\right] \quad vs \quad 0. \quad (21)$$

Assuming that $\texttt{freq}(X)\left[\prod_{j \in I-(K\cup K')}(1-\texttt{freq}(j))\right] \neq 0$, the above comparison shows that, for q even (q odd) $B_{\texttt{freq}}(K, \mathbf{I}-\mathbf{K})$ will be the better lower bound (upper bound) if and only if $\texttt{freq}(i) \leq \texttt{freq}(i')$.

Using this fact inductively implies that, given I, a $K \subseteq I$ with $|I - K| = q$, which leads to the best q-bound can be obtained by letting K consists of those elements of I from which have been removed the q elements of I with the highest frequencies. For example, if $I = \{i_1, i_2, i_3, i_4\}$ and $\texttt{freq}(i_1) \leq \texttt{freq}(i_2) \leq \texttt{freq}(i_3) \leq \texttt{freq}(i_4)$, and $q = 1$, the best K will be the set $I - \{i_4\} = \{i_1, i_2, i_3\}$. When $q = 2$, $K = \{i_1, i_2\}$, when $q = 3$, $K = \{i_1\}$, and when $q = 4$, $K = \emptyset$.

Combining the inter-analysis and intra-analysis results yields the following proposition:

PROPOSITION 4.3. Let $\mathcal{D} = (\mathcal{I}, \mathcal{B})$ be a basket database and let $I = \{i_1, \ldots, i_n\}$ ($n \geq 1$) be a subset of $\mathcal{I}$. Furthermore, without loss of generality, assume that $\texttt{freq}(i_1) \leq \cdots \leq \texttt{freq}(i_n)$, with $n \geq 2$. If $\mathcal{D}$ satisfies the independence property, then a K that leads to the best lower (upper) bound on $\texttt{freq}(I)$ consists of those elements of I from which have been removed the q elements of I with the highest frequencies, where q is the largest even (odd) value in $[0, n-1]$ at which the condition $\texttt{freq}(i_{n-q})+\texttt{freq}(i_{n-q-1}) \geq 1$ holds. Furthermore, this search for a best K can be done in $O(|I|)$, provided that the frequencies on the items in I are known.

REMARK 4.4.

1. As should be clear from the formulas used in the previous analysis, requiring the independence property on $\mathcal{D}$ is not necessary. For example, for Proposition 4.3 to hold at a particular I, it is sufficient that the independence property holds locally at I. In addition, examining the previous analysis in more detail shows that the results still hold for situations where *near* independence holds. We are currently working on formally determining precise definitions of "near" which are sufficient to determine these results.

2. If there exists no q that satisfies the condition $\texttt{freq}(i_{n-q}) + \texttt{freq}(i_{n-q-1}) \geq 1$, then the 0-rule, which yield the bounds 0, and the 1-rule (i.e., the Apriori rule) give the best lower and upper bound on $\texttt{supp}(I)$.

3. Notice that determining the best q does *not* require computing the bound values. This is in contrast with methods that explicitly need the computation of the bound values to determine the best bounds.

5. ALGORITHMIC IMPACT

We will now investigate methods of applying the theoretical results about bounding the FIM problem. We propose using heuristics in the FIM problem in two places. The first place is to use different q-rules as heuristics in pruning the search space of the FIM problem. We will call these *FIM heuristics*. The second place is, in the context of a fixed q-rule, to use heuristics that can efficiently compute good candidates for K that lead to bounds that well-approximate or equal the best bounds for the q-rule. We call these *bounding rules heuristics*.

FIM heuristics

FIM heuristics have been used in many FIM algorithms, such as Apriori, FP-growth and Eclat [2, 9, 15]. In these algorithms, the FIM heuristics that is used is the 1-rule to bound $\texttt{supp}(I)$ in terms of the supports of its subsets of size $|I| - 1$. Obviously, if one of these supports has been found to fall below threshold, then I is an infrequent item-set and can be pruned from the search space. However, if all that is determined is that the supports of some or all of these subsets are above threshold, then the 1-rule proposes to determine the $\texttt{supp}(I)$ by counting, and depending on its value to determine whether it is frequent or not. We propose using other q-rules (q odd) to possibly compute better bounds to further reduce

Function: R3
Input: Item-Set I.
Output: Return true if I needs to be counted.

1) For each $|K| = |I| - 3$
2) Get support of J where $K \subseteq J \subset I$.
3) If J is infrequent return false.
4) If $\texttt{upperbound}(K, \mathbf{I} - \mathbf{K}) > threshold$
5) return true
7) Else
8) return false.

Function: O3
Input: Item-Set I.
Output: Return true if I needs to be counted.

1) Set K equal to I without is 3 most frequent items
2) Get support of J where $K \subseteq J \subset I$.
3) If J is infrequent return false.
4) If $\texttt{upperbound}(K, \mathbf{I} - \mathbf{K}) > threshold$
5) return true
6) Else
7) return false.

Figure 1: Pseudo-code of the functions R3 and O3 test_candidate Function

the search space in FIM problems. Furthermore, when, in the computation of such bounds, it is discovered (by counting) that a subset of I is infrequent, then we stop and do not further compute the bound. Thus the algorithm harnesses both the monotonicity property, and the value of the bounds against the threshold. Using this breakout technique the q-rules will always do as well as the 1-rule.

Bounding rules heuristics

Given a fixed q, the best bound on $\texttt{supp}(I)$ is obtained by selecting a K that minimizes $\sum_{K \subseteq J \subseteq \overline{I-K}} d(J)$ (see Proposition 3.7). The crude way of doing that is by computing the bound for every possible K (of fixed size) and selecting the best bound. We describe this method below.

Rq The exhaustive way to determine K is to search through $\binom{|I|}{q}$ sets and for each of these sets, compute the bound $B(K, \mathbf{I} - \mathbf{K})$ is computed and tested against the support threshold. We label this approach Rq (e.g. R1 and R3 when $q = 1$ and $q = 3$ respectively). This will involve obtaining the support of the 2^{q-1} subsets of I. Thus the total cost of Rq is no more than $\binom{|I|}{q} 2^{q-1}$.

In the quest for a set K of a fixed size so that the bound computed is the best among the bounds computed using other K sets of the same size, we propose two different heuristics.

Hq We can reduce the cost needed for finding K such that $\sum_{K \subseteq J \subseteq \overline{I-K}} d(J)$ is minimized if we use Eqn. (15) and approximate that sum with $\texttt{supp}(K)$. Thus, this approach will first search through the $\binom{|I|}{q}$ subsets of I and determine a K with minimum support. For such a K, the bound $B(K, \mathbf{I} - \mathbf{K})$ is computed and tested against the support threshold. We label this approach Hq (e.g. H3 for $q = 3$). The total cost of Hq is no more than $\binom{|I|}{q} + 2^{q-1}$.

Oq (Oq*) We develop a heuristic aimed at reducing the cost of Hq. The idea is to assume that $\texttt{freq}(K)$ and $\texttt{supp}(K)$ are minimized when the frequency of the individual items comprising K are minimal. This idea is based on the theoretical results of Section 4. To implement this heuristic, the frequencies of the individual items of an item-set of size $|I|$ are obtained and sorted by frequency. Then K is selected such that the items with the highest q frequencies are filtered out. Once K is selected, one can proceed in the usual way to compute the bound $B(K, \mathbf{I} - \mathbf{K})$ and compare it against the support threshold. We label this approach Oq (e.g. O3 when $q = 3$). The total cost of Oq is no more than $|I| + 2^{q-1}$. Note that unlike Rq neither Hq nor Oq have R1 built in.

Oq* is the same as Oq, except that, when computing the bound on $\texttt{supp}(I)$, the minimum of the R1 bound and the Oq bound is used. Thus Oq* has both the R1 and Oq built in.

6. EXPERIMENTS

We now report on experiments we performed which use the heuristics discussed above. We emphasize that our ideas are not specific to the Apriori algorithm but rather target FIM algorithms in general.

Experimental setup

We used Ferenc Bodon's implementation of the Apriori algorithm [3], and extended it to include the heuristics in two places as discussed in Section 5. For FIM heuristics, we only use the 3-rule enumerated from our bounding theorem instead of the 1-rule. We assume that, in a pre-processing procedure, the frequencies of the items in $\mathcal{I}$ have been computed and sorted. For the bounding rules heuristics, we use H3, O3 and O3* (recall that O3* is the best of R1 and O3). These heuristics take place when generating and testing candidates. Figure 1 shows the test-candidate functions which incorporated the R3 and O3 heuristics, respectively. (The function incorporating H3 is very similar to O3. H3 looks up the supports of every K such that $|K| = |I| - 3$). Notice that only pre-candidates are subjected to these various rules. This means that in all cases the item-sets have been subjected to the most important part of the R1 rule, namely we know that the sets $I - \{i_1\}$, and $I - \{i_2\}$ are frequent (where i_1 and i_2 are the first and second most frequent items). This reduces the possibility of the 3-rule from being effective.

Baskets databases

We ran experiments on chess data, and census data (PUMS) which were provided by Roberto Bayardo from the UCI baskets databases [11]. Finally, we ran our approaches on webdocs - a 1.5GB baskets database donated by Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri to the FIMI repository [10] and built from a spidered collection of web html documents. Table 2 provides the essential statistics of baskets databases involved. Pumsb$*$ is the same baskets database as pumsb minus all the items with 80% or more frequency.

Experiments were run on a $3.06Ghz$ Intel Xeon system with $4GB$ of memory running RedHat Enterprise Linux. We ran three types of experiments describe below; the results are shown in Tables 3–6 and Appendix A. In all these experiments, we do not include

Table 2: Statistics of the baskets databases sizes

Baskets Database	Baskets	Avg. Basket Length	Distinct Items
chess	3,196	37	76
pumsb	49,046	74	7117
pumsb∗	49,046	50	7117
webdocs	1,692,082	177	5,267,657

Table 3: chess results at 50% threshold

Set		Freq	Exp1: Wasted effort				Exp2: Derivable items		
size	Pre-cand.	item-sets	R1	O3	O3*	R3	R1=O2	O3=O2	O3*=O2
3	4,504	4,000	503	69	69	69	366	1,560	1,729
4	19,847	18,565	1,198	55	55	11	3,442	11,741	12,029
5	60,960	58,172	2,408	54	48	0	18,089	45,576	46,408
6	134,191	129,952	3,468	34	29	0	57,633	115,407	116,670
7	218,411	214,297	3,347	18	17	0	122,979	204,138	205,167

Table 4: PUMSB results at 80% threshold

Set		Freq	Exp1: Wasted effort				Exp2: Derivable items		
size	Pre-cand.	item-sets	R1	O3	O3*	R3	R1=O2	O3=O2	O3*=O2
3	1,842	1,760	82	3	3	3	252	99	340
4	7,668	6,999	566	8	8	2	906	1,475	1,584
5	20,458	18,215	1,733	8	4	0	2,613	4,834	5,229
6	35,717	31,532	2,873	20	17	0	6,177	10,235	11,659
7	41,290	36,382	2,666	13	1	0	11,016	13,653	16,612

Table 5: PUMSB∗ results at 25% threshold

Set		Freq	Exp1: Wasted effort				Exp2: Derivable items		
size	Pre-cand.	item-sets	R1	O3	O3*	R3	R1=O2	O3=O2	O3*=O2
3	8,817	6,106	2,551	1,051	1,051	1,051	1,172	112	1,231
4	27,436	23,331	829	286	286	119	8,710	6,218	9,771
5	72,715	65,625	432	196	52	5	33,441	30,191	37,115
6	155,887	142,594	280	281	35	0	85,187	79,353	91,110
7	267,801	245,939	171	452	13	0	160,024	147,127	165,838

Table 6: webdocs results at 10% threshold

Set		Freq	Exp1: Wasted effort				Exp2: Derivable items		
size	Pre-cand.	item-sets	R1	O3	O3*	R3	R1=O2	O3=O2	O3*=O2
3	26,125	12,343	13,711	13,711	13,711	13,711	0	0	0
4	40,619	31,857	8,480	8,361	8,345	8,345	0	0	0
5	57,336	52,084	5,011	3,135	3,130	2,919	0	0	0
6	58,560	55,867	2,346	739	739	565	0	50	50
7	40,057	39,299	650	160	160	101	36	787	810
8	17,828	17,710	109	109	18	8	80	2,266	2,280

results for H3 since it produced identical results to O3 except in one instance.

Exp1 (See Tables 3–6.) In this experiment, we use the 3-rule as an FIM heuristic and use it to prune the search space. The idea is for every item-set we compute the upper bound using R3, O3 or O3* instead of the 1-rule, and count the number of candidates that the rules report as possibly frequent, yet, when counting them in the basket database, turn out to be infrequent (i.e. wasted effort).

Exp2 (See Tables 3–6.) We count the number of derivable item-sets [6] that have their lower bound and upper bound equal. We also present below this data, the remaining candidates that need to be counted as a percentage of the pre-candidates. We computed the upper bound on the support of an item-set using R1, O3, O3* and R3 with the lower bound being computed using O2. This experiment is interesting since for the item-sets that have their upper bounds and lower bounds equal, one does not need to count their supports.

Exp3 (See Appendix A.) We modify **Exp1** by using the lower bound rule (O2 heuristic) to further prune the search space. The idea is that, if the lower bound is greater than the threshold, then we know that item-set is frequent without counting. Thus, in this experiment we only report the number of candidates that require counting. These are the item-sets whose lower bound is below the threshold and their upper bound is above the threshold. Note that when implementing this algorithm one may reach a stage where the supports of subsets of an item-set are not available (since the frequency status of these subsets may have been determined without counting) [11]. We do not address this issue.

Interpretation of results

From the data presented in Tables 3–4, it is apparent that the $3-$rule prunes the search space and reduces the number of infrequent sets that need to be counted almost perfectly in **Exp1**. The 1-rule doing a good job in eliminating the infrequent item-sets but the 3-rule is doing an almost perfect job. This is consistent with our theoretical results in Section 4, which predict that the 3-rule will do better on dense baskets databases. The other observation deals with the performance of our proposed heuristics: O3 prunes the search space and reduces wasted effort almost as good as O3* and R3 in **Exp1**. In Tables 3–4 of **Exp2** we find more derivable item-sets when computing the lower and bounds using the O2 and O3 than using O2 and R1. The number of derivable items found using O3 and O3* (in conjunction with the lower bound) is very close to the number of derivable items found using R3 with the lower bound. This demonstrates the performance of our proposed approaches. In Appendix A in **Exp3** we find that the remaining candidates that need to be counted after using O3 (or O3*) and O2 to prune the search space is: (1) noticeably less than those remaining candidates that need to be counted after using only R1 and O2; (2) very close to those remaining candidates that need to be counted after using R3 and O2 (the best).

For the data presented in Tables 5–6 of **Exp1**, computing the upper bound using 3-rule with R3 is still reducing wasted effort more than the 1-rule. However the pruning relative effectiveness of these rules is less as can be clearly seen when comparing O3 to R1 in Table 6. Observe there exists an outlier in our results in Table 5 where O3 does slightly worse than the 1-rule at stage 7 of the algorithm. We do not see this as a major problem since it is not observed in O3*. The decrease in pruning effectiveness of these rules can also be observed in Table 6 of **Exp2**, where all rules are unsuccessful in both pruning wasted effort and finding derivable item-sets in the early stages of the algorithm. More importantly, we observe that the use of R1 and O2 to find derivable item-sets is more effective than using O3 and O2 (Observe this does not occur in O3* or R3 with O2 since both these approaches have R1 built into them). This is consistent with our theoretical results in Section 4, which predict the 1-rule be more effective than the 3-rule on sparse baskets databases. We also observe a similar reduction in pruning effectiveness in Appendix A of **Exp3** where we find the remaining candidates that need to be counted using either of O3, O3* or R3 (in conjunction with O2) very close to those candidates that need to be counted using R1 (in conjunction with O2).

7. SUMMARY

- The computational cost of computing a particular bound is equivalent to finding a K such that $\sum_{K \subseteq J \subseteq \overline{I-K}} d(U)$ is minimized and then computing the bound $\bar{B}(K, \mathbf{I} - \mathbf{K})$. While the cost of computing $B(K, \mathbf{I} - \mathbf{K})$ is the same for each heuristic, the cost of finding a K-set such that the sum over density is minimized can be reduced significantly by using the heuristics we propose in this paper.

- The performance of the approaches suggests that all algorithms produce more or less the same number of false candidates, with O3 typically performing better.

- For dense baskets databases, O3 and O3* produce less false candidates than R1, and thus resulting in better performance. For baskets databases too big to fit in memory, it is desirable to avoid false candidates as much as possible. O3 (or O3*) will perform better since the computational cost of O3 (in memory) will usually outperform the cost of false candidates (since that involves I/O access).

- The heuristics we suggest will work for both lower bounds and upper bounds. This will have a positive impact on performance of algorithms like MAXMINER and AprioriLB [11], and concise representation algorithms such as the NDI-Algorithm [6].

- We recommend that one may just as well run the 3-rule in the Apriori algorithm, at least at level 3. This does not present a significant overhead. The advantages are that for dense baskets databases, it will pay off reducing wasted effort to almost 0. Furthermore, even on sparse baskets databases such as webdocs, a reduction in wasted effort can be expected.

Acknowledgment: We thank Ferenc Bodon for use of his implementation of the Apriori algorithm [3].

8. REFERENCES

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216. ACM Press, 1993.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.

[3] Ferenc Bodon. A fast apriori implementation. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

[4] Toon Calders. *Axiomatization and Deduction Rules for the Frequency of Itemsets*. PhD dissertation- University of Antwerp, 2003.

[5] Toon Calders. Computational complexity of itemset frequency satisfiability. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 143–154, 2004.

[6] Toon Calders and Bart Goethals. Mining all non-derivable frequent itemsets. In *Proceedings European Conference on Principles of Data Mining and Knowledge Discovery*, volume 2431 of *LNCS*, pages 74–85. Springer-Verlag, 2002.

[7] Toon Calders and Jan Paredaens. Axiomatization of frequent sets. In *Proceedings of the international conference on database theory*, pages 204–218, 2001.

[8] Dimitrios Gunopulos, Heikki Mannila, and Sanjeev Saluja. Discovering all most specific sentences by randomized algorithms. In *Proceedings of the 6th International Conference on Database Theory*, pages 215–229. Springer-Verlag, 1997.

[9] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM Press, 2000.

[10] FIMI Repository. http://fimi.cs.helsinki.fi/data.

[11] Jr. Roberto J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 85–93. ACM Press, 1998.

[12] Bassem Sayrafi and Dirk Van Gucht. Inference systems derived from additive measures. *Workshop on Causality and Causal Discovery, London, Canada*, 2004.

[13] Bassem Sayrafi and Dirk Van Gucht. Differential constraints. In *Proceedings of the twenty fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM Press, 2005.

[14] Bassem Sayrafi, Dirk Van Gucht, and Marc Gyssens. Measures in databases and datamining. Tech. Report TR602, Indiana University Computer Science, 2004.

[15] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

APPENDIX

A. LOWER BOUND RESULTS

We present some results from combining the $2-$rule (denoted by LB in the tables below) with R1 and O3. Remaining candidates refers to the candidate sets that remain after pruning the search space using the rules noted. It is only for these sets that we are required to visit the database and count their support. Observe in the chess and PUMSB results, that the number of remaining candidates is reduced significantly when computing the lower and upper bounds using O2 and O3 versus O2 and the 1-rule. On the other baskets databases (sparse baskets databases), we observe that the same effect only on a smaller scale.

chess results at 50% support

			Remaining candidates		
Set size	Pre-cand.	Freq	LB + R1	LB + O3	LB+O3*
3	4,504	4,000	596	162	162
4	19,847	18,565	1,388	245	245
5	60,960	58,172	2,716	356	356
6	134,191	129,952	3,763	324	324
7	218,411	214,297	3,539	209	209

PUMSB results at 80% support

			Remaining candidates		
Set size	Pre-cand.	Freq	LB + R1	LB + O3	LB+O3*
3	1,842	1,760	97	18	18
4	7,668	6,999	622	64	64
5	20,458	18,215	1,917	192	188
6	35,717	31,532	3,445	592	589
7	41,290	36,382	3,635	982	970

PUMSB$*$ results at 25% support

			Remaining candidates		
Set size	Pre-cand.	Freq	LB + R1	LB + O3	LB+O3*
3	8,817	6,106	3,967	2,467	2,467
4	27,436	23,331	3,008	2,465	2,465
5	72,715	65,625	4,046	3,810	3,666
6	155,887	142,594	5,941	5,942	5,696
7	267,801	245,939	7,849	8,130	7,691

webdocs results at 10% support

			Remaining candidates		
Set size	Pre-cand.	Freq	LB + R1	LB + O3	LB+O3*
3	26,125	12,343	19,422	19,422	19,422
4	40,619	31,857	12,743	12,624	12,608
5	57,336	52,084	7,177	5,301	5,296
6	58,560	55,867	3,207	1,600	1,600
7	40,057	39,299	858	368	368
8	17,828	17,710	125	34	34

A Trie-based APRIORI Implementation for Mining Frequent Item sequences

Ferenc Bodon*
Department of Computer Science and Information Theory,
Budapest University of Technology and Economics and
Computer and Automation Research Institute of the
Hungarian Academy of Sciences
bodon@cs.bme.hu

ABSTRACT

In this paper we investigate a trie-based APRIORI algorithm for mining frequent item sequences in a transactional database. We examine the data structure, implementation and algorithmic features mainly focusing on those that also arise in frequent itemset mining. In our analysis we take into consideration modern processors' properties (memory hierarchies, prefetching, branch prediction, cache line size, etc.), in order to better understand the results of the experiments.

Keywords

Frequent item sequence mining, APRIORI algorithm, trie.

1. INTRODUCTION

Algorithm APRIORI [1] is one of the oldest and most versatile algorithms of Frequent Pattern Mining (FPM). With sound data structures and careful implementation it has been proven to be a competitive algorithm in the contest of Frequent Itemset Mining Implementations (FIMI) [8]. Although it was beaten most of the time by sophisticated DFS algorithms, such as `lcm` [19], `nonordfp` [15] and `eclat` [17], its merits are undisputable. Its advantages and its moderate traverse of the search space pay off when mining very large databases, where `eclat` requires too much memory and CPU in handling TID-lists of frequent pairs. APRIORI also outperforms FP-growth based algorithms in databases that include many frequent items, but not many frequent itemsets, because generating the conditional FP-trees takes too long.

APRIORI is not only an appreciated member of the FIMI community and regarded as a baseline algorithm, but its variants to find frequent sequences of itemsets [2], episodes, [12], boolean formulas [9] and labeled graphs [10, 11] have proven to be efficient algorithms as well.

Mining frequent item sequences (also called as serial episodes) in transactional data (FSM) is a neglected field of FPM in spite of its theoretical significance. It is an immediate generalization of frequent itemset mining, hence it is useful to investigate what difficulties arise when we take into consideration the ordering and when we allow duplicates both in the transactions and in the patterns. Throughout this paper, we focus on the differences between trie-based APRIORI of FIM and FSM.

2. PROBLEM STATEMENT

Frequent item sequence mining is a special case of *Frequent Pattern Mining*. Let us first describe this general case. We assume that the reader is familiar with the basics of poset theory. We call a poset $(P, \preceq)$ *locally finite*, if every interval $[x, y]$ is finite, i.e. the number of elements z, such that $x \preceq z \preceq y$ is finite. The element x *covers* y, if $y \preceq x$ and for any z such that $y \preceq z$, we have $z \not\preceq x$.

DEFINITION 1. *We call the poset $\mathcal{PC} = (\mathcal{P}, \preceq)$* pattern context*, if there exists exactly one minimal element, $\mathcal{PC}$ is locally finite and* graded*, i.e. there exists a* size *function $|\ |: \mathcal{P} \to \mathbb{Z}$, such that $|p| = |p'| + 1$, if p covers p'. The elements of $\mathcal{P}$ are called* patterns *and $\mathcal{P}$ is called the* pattern space *or* pattern set.

Without loss of generality, we assume that the size of the minimal pattern is 0 and it is called the *empty pattern*.

In the *frequent pattern mining problem*, we are given the set of input data $\mathcal{T}$, the pattern context $\mathcal{PC} = (\mathcal{P}, \preceq)$, the anti-monotonic function $supp_{\mathcal{T}} : \mathcal{P} \to \mathbb{N}$ and $min\_supp \in \mathbb{N}$. We have to find the set $F = \{p \in \mathcal{P} : supp_{\mathcal{T}}(p) \geq min\_supp\}$ and the support of the patterns in F. Elements of F are

*This work was supported in part by OTKA Grants T42481, T42706, TS-044733 of the Hungarian National Science Fund, NKFP-2/0017/2002 project Data Riddle and by a Madame Curie Fellowship (IHP Contract nr. HPMT-CT-2001-00251).

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

called *frequent* patterns, $supp_{\mathcal{T}}$ is the *support function* and *min_supp* is referred as *support threshold*.

A large family of the FPM is mining frequent patterns in a *transactional database*, i.e. the input data is a set of *transactions*, and the support function is defined on the basis of a *containment relation*. The support of a pattern equals to the number of transactions that *contain* the pattern. In the case of frequent itemset and frequent item sequence mining, the type of the patterns and the type of the transactions are the same, i.e. itemsets and item sequences and the containment relation is $\preceq$ (i.e. an itemset/item sequence p is contained in a transaction t, if p is a subset/subsequence of t). Containment relation of itemsets corresponds to the traditional set inclusion ($\subseteq$) relation. In the case of item sequences we say that item sequence $s = \langle i_1, i_2, \ldots, i_n \rangle$ is a subsequence of $s' = \langle i'_1, i'_2, \ldots, i'_m \rangle$ if there exist integers $1 \leq j_1 < j_2 < \cdots < j_n \leq m$, such that $i_1 = i'_{j_1}, i_2 = i'_{j_2}, \ldots, i_n = i'_{j_n}$, i.e. we can get s by deleting some items from s'. For example $\langle e, a, a, b \rangle \prec \langle f, e, a, b, c, a, a, c, b \rangle$ because $i_1 = 2$, $i_2 = 3$, $i_4 = 6$, $i_4 = 9$ meet the requirements. We can regard this FSM problem statement as a generalization of the FIM or as a specialization of the frequent sequence of itemset mining [2].

We denote the set of items by $\mathcal{I}$. Without loss of generality we assume that the elements of $\mathcal{I}$ are consecutive integers starting form zero.

3. APRIORI IN A NUTSHELL

APRIORI scans the transaction dataset several times. After the first scan, the frequent items are found, and in general after the ℓ^{th} scan, the frequent item sequences of size ℓ (we call them ℓ-sequences) are extracted. The method does not determine the support of every possible sequence. In an attempt to narrow the domain to be searched, before every pass it generates *candidate* sequences. A sequence becomes a candidate if every subsequence of it is frequent. Obviously every frequent sequence is a candidate too, hence it is enough to calculate the support of candidates. Frequent ℓ-sequences generate the candidate $(\ell+1)$-sequences after the ℓ^{th} scan.

Candidates are generated in two steps. First, pairs of ℓ-sequences are found, where the elements of the pairs have the same prefix of size $\ell - 1$. Here we denote the elements of such a pair with $\langle i_1, i_2, \ldots, i_{\ell-1}, i_\ell \rangle$ and $\langle i_1, i_2, \ldots, i_{\ell-1}, i'_\ell \rangle$. Depending on items i_ℓ and i'_ℓ we generate one or two *potential candidates*. If $i_\ell \neq i'_\ell$ then they are $\langle i_1, i_2, \ldots, i_{\ell-1}, i_\ell, i'_\ell \rangle$ and $\langle i_1, i_2, \ldots, i_{\ell-1}, i'_\ell, i_\ell \rangle$, otherwise it is $\langle i_1, i_2, \ldots, i_{\ell-1}, i_\ell, i_\ell \rangle$ [12]. In the second step the ℓ-subsequences of the potential candidate are checked. If all subsequences are frequent, it becomes a candidate.

After all the candidate $(\ell+1)$-sequences have been generated, a new scan of the transactions is started and the precise support of the candidates is determined. This is done by reading the transactions one-by-one. For each transaction t the algorithm decides which candidates is contained by t. After the last transaction is processed, the candidates with support below the support threshold are thrown away. The algorithm ends when no candidates are generated.

The choice of the data structure to store candidates is a primary factor that determines the efficiency of algorithm. Trie-based APRIORI implementations for mining frequent itemsets are the most competitive ones [6, 3, 4]. Since the itemsets are treated as special item sequences, it is a natural approach to adopt a trie-based implementation to find frequent item sequences.

3.1 The Trie Data Structure

A trie is a rooted, labeled tree. In the FIM and FSM setting each label is an item. The root is defined to be at depth 0 and a node at depth d can point to nodes at depth $d+1$. A pointer is also referred to as *edge* or *link*. If node u points to node v, then we call u the *parent* of v, and v the *child* node of u. Nodes with the same parent are *siblings* and nodes that have no children are called *leaves*. Each node represents an item sequence that is the concatenation of labels of the edges that are on the path from the root to the node. In the rest of the paper, the representation of the node is sometimes called the sequence of the node.

For the sake of efficiency – concerning insertion and lookup – a total order on the labels of edges is defined. Figure 1 shows tries in the case of itemsets and item sequences, along with some important differences.

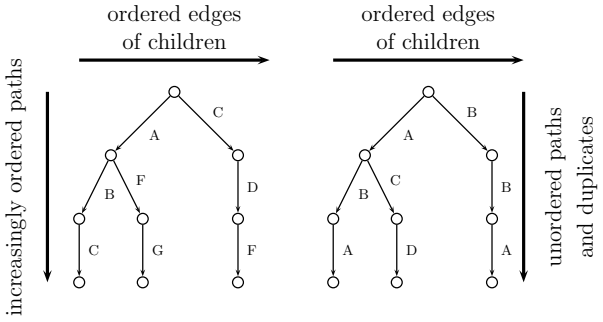


Figure 1: Tries of itemsets and item sequences

Edges can be stored in many ways. The two most important are the so called *linked-list representation* and the *offsetindex-based tabular representation* [6]. In the first solution, all edges of a node are described by (label, pointer) pairs that are stored ordered by labels in a vector. In the second solution only the pointers are stored in a vector, whose length equals to $l_{max} - l_{min}$, where l_{min} and l_{max} denote the smallest and the largest labels of the edges respectively. An element at index i belongs to the edge whose label is $l_{min} + i$. If there is no edge with such a label, then the element is `NIL`.

3.1.1 The Trie of APRIORI

For the sake of fast support counting the candidates are stored in a trie. Determining the supports of the candidates does not make much difference in the itemset and item sequence cases. We take the transactions one-by-one. With a recursive traversal we travel some part of the trie and reach those leaves that are contained in the actual transaction t. The support counters of these leaves are increased. The traverse of the trie is driven by the elements of t. No step is performed on edges that have labels that are not contained in t. More precisely, if we are at a node at depth d by following a link labelled with the j^{th} item in t, then we move

forward on those links that have the labels $i \in t$ with index greater than j, but less than $|t| - \ell + d + 1$.

It would be inefficient to build a new trie in each iteration of APRIORI. Instead, one trie is maintained during the algorithm. In the candidate generation phase new leaves are added, and in the infrequent candidate removal phase, leaves are deleted. Obviously "dead-end" paths (paths that do not lead to any leaves) can also be removed, since they do not play any role in the latter steps of the algorithm. Removing dead-end paths (which may mean removing whole branches) speeds-up the support counting method and decreases memory need. This is due to two facts. First, finding the corresponding edge of a node is proportional to the number of edges of the node. Second, by removing unnecessary edges we need less cache lines to store the list of edges, that results in less cache misses and improve data locality.

Some paths can also become dead-ends during the candidate generation phase. If a leaf cannot be extended – because it has no extension whose all subsequences are frequent – then the path of this node is a dead-end path. There is a difference between itemsets and item sequences regarding the removal of this node. Since the leaves are visited in a depth first manner, the itemset represented by the dead-end node is not required in the latter subset checks. This is a straightforward consequence of the following property.

PROPERTY 1. *For a given depth d, the depth-first ordering of the nodes' representation at depth d is the same as if we lexicographically order these representations, where the order used in the lexicographical ordering corresponds to the edge ordering of the trie.*

Removing dead-end paths during the candidate generation phase speeds-up subset test of other potential candidates. The property, however, does not hold for item sequences, thus the technique can not be applied. Since the dead-end nodes at depth ℓ are only needed in the subset checks of $(\ell + 1)$-sequences and never again in the later phases, they can be removed, however, after the candidate generation step. This requires one extra scan of the trie.

3.2 Routing Strategies at the Nodes

In support counting methods we have to find all leaves that represent ℓ-item candidates that are contained in a given transaction t. As already described, this is done by a recursive traversal of the trie. The main step of the recursion is the following: given a part of the transaction (t') and a current node of the trie, we have to find the edges that correspond to an item in t'. Routing strategy refers to the method of finding the edges to follow. This is the step that primarily determines the run-time of the algorithm.

There exist many routing strategies. In the following we describe the most important ones. The notations of the methods used in the experiments are given after the descriptions. We denote the number of edges of the current node by n.

search for corresponding item: here we take the edges one-by-one, and check if there exists an element of t' that equals to the label of the edge. Since the transaction is not ordered, we can do early stops only if the label item is found; otherwise we have to go over all the items of t'. This requires in the worst-case $n|t'|$ comparisons and index increases. We refer to this method as `lookup_seq` in our experiments.

The linear time of finding a given item in the transaction can be improved if we use a tabular representation of the transaction. In the case of itemsets, only the existence of an item is important, hence an indexvector or a bitvector is enough to serve a proper support counting. This does not hold for item sequences, we also need all the positions of the occurrences. For this we have to use a position array, the row i stores the positions of occurrences of item i. To avoid scanning row i as many times as i appears on an edge of a visited node we do the following.

At each recursive step of the support counting, we keep track of a pointer of each row. Initially, all pointers point to the first elements of the rows. At a recursive step only the pointed position is considered, and incremented as long as a position is reached that is greater than the position of the item that led to the current trie node. Before entering to a recursive step (going down one step on the trie) the original value of pointer has to be stored, and after the return from the recursive step the original value has to be set back. Since the pointers can only increase along a path of a trie, we save many superfluous pointer increases with this solution (this method is referred as `lookup_seq_array`).

search for corresponding label: For each element i of t', we check if there exists an edge with label i. Since duplicates may occur in t', we have to keep track of those items that have already occurred in the transaction. For this, we make a bitvector initialized with `true` values. The element at index i belongs to item i. Search for edge with label i is only started if the boolean value at index i is true. After the search the boolean value of i is set to `false`.

If the tabular representation is used (`lookup_edge_oi`), then finding the edge with the given label requires one step ($|t'|$ comparisons). In the case of linked list a binary search can be used (`lookup_edge_bin`). This approach requires in the worst case $|t'| \log_2 n$ comparisons. Although binary search is theoretically faster than a linear search, this does not necessarily hold in the case of modern processors, especially not for short lists. Binary search performs assignments that depends on the outcome of a comparison, which is hardly predictable, thus the pipeline of the processor has to be often flushed. Also prefetching (data locality) is more effective in the case of linear search.

The naive linear search (always scanning the edges from the first until the edge is found whose label is greater than or equal to the item – `lookup_edge_lin`) can be improved if we store the index of edge where the last linear search terminated. If the next element of the transaction is greater than the label of the stored edge, then we continue the search from this edge. Otherwise our search is continued backwards (`lookup_edge_commute`). Note, that this method meets

the data locality requirement better and causes less cache misses than binary search, which is the reason why it sometimes outperforms its binary search counterpart.

simultaneous traversal: In the case of frequent *itemset* mining, we have seen that simultaneous traversal (also called *merging*) is the best choice [4]. On most of the datasets it finishes in the first place, and in cases when it is just the runner-up the advantage of the winner is not significant. This is again attributed to processor's prefetch and data locality features and the fact that in most cases the number of elements in t' and the number of edges of the nodes is small. Simultaneous traversal can only be applied if both sets are ordered. To guarantee this, we have to sort t' and remove duplicates. This can be done in two ways. On one hand, we can sort the elements, then with a single traversal we remove duplicates (`merge_sort_remove`). On the other hand, we can apply the bitvector-based approach to generate the list of items of t' that contains no duplicates, and then perform the sorting (`merge_bitvec_sort`). Although simultaneous traversal is linear in $|t'|$ and n, the preprocessing (i.e. the sorting) may require $|t'|\log(|t'|)$ steps.

When a bitvector is used to avoid double traverse through the same edge (all `lookup_edge` and the `merge_bitvec_sort` methods), it is important to use the offsetindex approach, i.e use a vector of length $l_{max} - l_{min}$, where l_{max} and l_{min} denote the maximal and minimal label of the actual node. These two values can be computed very quickly on any edge representation (ordered linked list or offset index vector) we use. When we decide if item i is already used, we first check if $l_{min} \leq i \leq l_{max}$, and if this holds, we read the value of the bitvector at position $i - l_{min}$. Our experiments show that this small optimization has a large impact on the run time. This is due to the overhead of initializing extra boolean values and more importantly due to the smaller vectors, thus less cache line requirement and less cache misses.

3.3 Candidate Generation

Originally APRIORI uses complete pruning, i.e after generating a potential candidate, it checks all subsets of the potential candidate if they are frequent. The subsequence checks can be solved in two ways.

3.3.1 Simple Pruning

In the *simple pruning strategy* we check each ℓ-subsequence of the potential $(\ell+1)$-element candidates one-by-one. If all subsequences are found to be frequent, then the potential candidate becomes a real candidate. Two straightforward modifications can be applied to reduce unnecessary work. On one hand, we do not check those subsequences that are obtained by removing the last and the one before the last elements. On the other hand, the prune check is terminated as soon as a subsequence is infrequent, i.e. not contained in the trie.

3.3.2 Intersection-based Pruning

A problem with the simple pruning method is that it unnecessarily travels some part of the trie many times. We illustrate this by an example. Let $ABCD$, $ABCE$, $ABCF$, $ABCG$ be frequent 4-sequences. When we check the subsequences of potential candidates $ABCDE$, $ABCDF$, $ABCDG$ then we travel through nodes ABD, ACD and BCD three times. This gets even worse if we take into consideration all potential candidates that stem from node ABC. We travel to each subsequence of ABC 6 times.

To save these superfluous traverses we have proposed an *intersection-based pruning* method [5] that can be directly used for item sequences as well. We denote by u the current leaf that has to be extended, the depth of u by ℓ, the parent of u by P and the label that is on the edge from P to u by i. To generate new children of u, we do the following. First determine the nodes that represent all the $(\ell-2)$-subsequences of the $(\ell-1)$-prefix. Let us denote these nodes by $v_1, v_2, \ldots, v_{\ell-1}$. Then find the child v'_j of each v_j that is pointed by an edge with label i. If there exists a v_j that has no edge with label i (due to dead-end branch removal), then the extension of u is terminated and the candidate generation continues with the extension of u's sibling (or with the next leaf, if u does not have any siblings). The complete pruning requirement is equivalent to the condition that only those labels can be on an edge that starts from u, which are labels of an edge starting from v'_j and labels of one starting from P. This has to be fulfilled for each v'_j, consequently, the labels of the new edges are exactly the intersection of labels starting from v'_j and P nodes.

The siblings of u have the same prefix as u, thus, in generating the children of siblings, we can use the same nodes $v_1, v_2, \ldots v_{\ell-1}$. It is enough to find their children with the proper label (the new v'_j nodes) and compute the intersection of the labels of edges that start from the prefix and the new $v'_1, v'_2, \ldots v'_{\ell-1}$. This is the real advantage of this method. The $(\ell-2)$-subsequence nodes of the prefix are reused, hence the paths representing the subsequences are traversed only once, instead of $\binom{n}{2}$, where n is the number of children of the prefix.

As an illustrative example let us assume that the trie that is obtained after removing infrequent sequences of size 4 is depicted in Fig. 2.

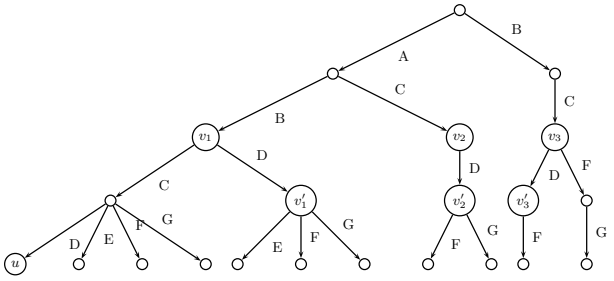


Figure 2: Example: intersection-based pruning

To extend the node $ABCD$, we find the nodes that represent the 2-subsequences of the prefix (ABC). These nodes are denoted by v_1, v_2, v_2. Next we find their children that are reached by edges with label D. These children are denoted by v'_1, v'_2 and v'_3 in the trie. The intersection of the label

sets associated to the children of the prefix, v'_1, v'_2 and v'_3 is: $\{D, E, F, G\} \cap \{E, F, G\} \cap \{F, G\} \cap \{F\} = \{F\}$, hence only one child will be added to node $ABCD$, and F will be the label of this new edge.

The intersection-based solution can easily be generalized. In generating descendants of the sibling we used the fact that the subsequences of the potential candidates can quickly be obtained from the subsequences of the $(\ell-1)$-element common prefix. Hence, it is enough to determine the subsequences of the prefix only once. Even more redundant traversals can be spared if we not only generate descendants of the siblings, but also the descendants of the cousin nodes (nodes that have the same grandparent node). All required subsequences can be reached from the $(\ell-3)$-subsequences of the $(\ell-2)$-element common prefix. The idea can be further generalized.

3.3.3 No Pruning

Complete pruning is an inherent feature of APRIORI. Our recent research [5], however, showed that complete pruning in the case of itemsets does not necessarily decrease running time. In fact, if we omit subset containment check we get a faster algorithm in most cases. This is due to the following inequality that holds in most of the known test databases:

$$|NB^{\prec_A}(F) \setminus NB(F)| \ll |F|,$$

where $\prec_A$ denotes the ascending order according to the frequencies. Here F denotes a set of frequent itemsets, $NB(F)$ the negative border [18] of F, and $NB^{\prec}(F)$ the order-based negative border (an itemset I is element of $NB^{\prec}(F)$, if I is not frequent, but the two smallest $(|I|-1)$-subsets of I are frequent. Here "smallest" is understood with respect to $\prec$ ordering of items.)

The left-hand side of the inequality is proportional to the extra work to be done if each potential candidate was automatically regarded as a candidate, i.e., the extra work of determining the support of those itemsets that would not be candidates in the original APRIORI. The right-hand side is proportional to the work done by pruning. This suggests that the extra work done by subset check is more than it saves. The following figure shows some result of our experiments.

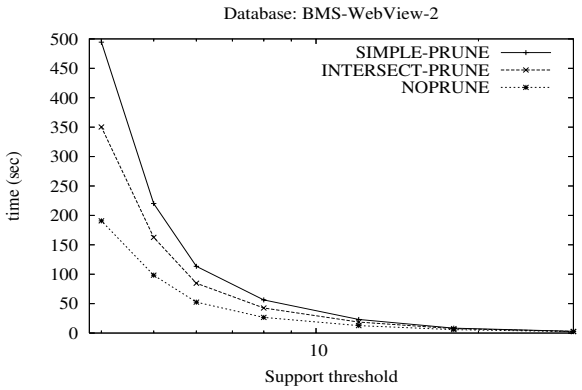


Figure 3: Candidate generation of itemsets with different pruning strategies

Although the $|NB^{\prec_A}(F) \setminus NB(F)| \ll |F|$ observation holds for item sequences as well (definition of $NB(F)$ and $NB^{\prec}(F)$ can easily be generalized to sequences), the left-hand side is weighted with a much larger factor due to the deterioration of support count. Determination of a subset of an itemset and a subsequence of an item sequence in the candidate generation take exactly the same time. Support counting, however, is slower in the case of item sequences. This is also suggested if we compare worst-cases of the best routing strategy of itemset (simultaneous traversal) and sequences (`lookup_seq`), which are $n + |t'|$ and $n \cdot |t'|$. Consequently, our expectation is that omitting complete pruning does not speed-up APRIORI in general, but just in those cases where the size of the transaction is small (an thus the difference between time requirement of subset checks and support count is not significant) and the above observation holds.

3.4 Omitting Equisupport Extensions

An important FIM optimization technique is the equisupport pruning. Omitting equisupport extension means excluding from support counting the superset of those ℓ-itemsets that have the same support as one of their $(\ell-1)$-subsets. This comes from the following simple property.

PROPERTY 2. *Let $X \subset Y \subseteq \mathcal{I}$. If $supp(X) = supp(Y)$ then $supp(Y \cup Z) = supp(X \cup Z)$ for any $Z \subseteq \mathcal{I} \setminus Y$.*

If candidate Y has the same support as its prefix, then it is not necessary to generate any superset of Y as new candidate. The support of the prefix is available in all depth-first algorithms and in APRIORI as well, and can be obtained very quickly, which is the main reason why omitting the prefix-equisupport extensions (denoting the method *prefix-equisupport pruning*) is one of the most versatile speed-up tricks in FIM implementations. In the case of databases that contain no non-closed itemsets (and hence this pruning is never used), the degradation of performance is insignificant, while in dense databases the improvement can be of several orders of magnitude. The following figure illustrates the speed-up gained when this technique is applied in a very dense dataset.

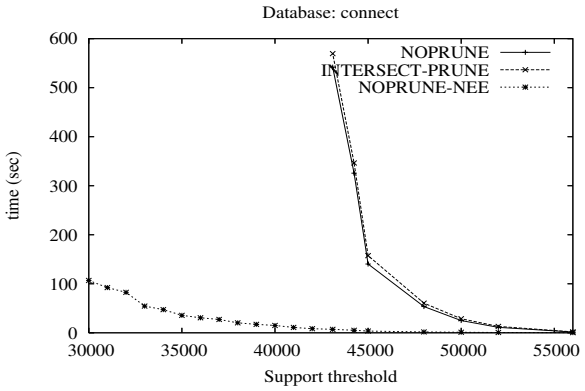


Figure 4: Omitting equisupport extensions (itemset case)

Note, that omitting equisupport extensions does not mean that we simply remove the leaves that represent equisupport

itemsets. This would not lead to a complete FIM algorithm, as complete pruning and candidate generation depends on the existence of frequent leaves. A list – called `ee_list` – is associated with each node storing the labels of edges that lead to children with the same support as the node considered. When an equisupport extension is found the label of the last edge is added to the `ee_list` of the parent, and then the leaf is deleted. It is like changing the edge to a loop edge and deleting the originally pointed node. When a representation of a leaf is written out, we also print the representation extended by each subset of the set that is obtained by taking the unions of the `ee_lists` of nodes that are on the path from the root to the leaf.

Due to its versatility and efficiency, it is required to examine if the trick can be applied in the case of item sequences. First, we have to determine if the above property holds if X, Y and Z are item sequences, $\subseteq$ denotes the containment relation of item sequences and union means concatenation. The following simple example proves that the property does not hold for item sequences. Let $t_1 = \langle A \rangle$ and $t_2 = \langle B, A \rangle$. Then $supp(\langle\rangle) = supp(\langle A \rangle) = 2$ but $supp(\langle B \rangle) = 1 \neq 0 = supp(\langle A, B \rangle)$. Note, that the empty sequence is the prefix of $\langle A \rangle$ which means that the property surely does not hold in general and in the case we restrict the subsequence equality condition to prefixes.

If duplicates were not allowed in the transactions, then the property would hold for non-prefix subsequences. This can be easily proven based on the definition of the contain relation. Due to the legitimacy of duplicates the property, however, does not hold. This is shown by the following example. Let the database consists of a single sequence $t = \langle B, x, A, B \rangle$. Here $supp(\langle B \rangle) = supp(\langle A, B \rangle) = 1$, however $supp(\langle B, x \rangle) \neq supp(\langle A, B, x \rangle)$.

3.5 Transaction Caching

Let us call the item sequence that is obtained by removing infrequent items from t the *filtered transaction* of t. All frequent item sequences can be determined even if only filtered transactions are available. To reduce IO and parsing costs and speed up the algorithm, the filtered transactions can be stored in main memory instead of on disk. It is ineffective to store the same filtered transactions multiple times. Instead, store them once and employ counters which store the multiplicities. This way, memory is saved and run-time can be significantly reduced.

Collecting filtered transactions has a significant influence on run-time. This is due to the fact that finding candidates that occur in a given transaction is a slow operation and the number of these procedure calls is considerably reduced. If a filtered transaction occurs n times, then the expensive procedure will be called just once (with counter increment n) instead of n times (with counter increment 1).

Different data structures are used for storing filtered transaction in the competitive APRIORI implementation for FIM. Today's fastest APRIORI implementation [6] uses a trie, our previous implementation adopted a red-black tree. The same problem occurs in FP-growth based algorithms, where Patricia-tree based solution [14] showed prominent results.

Transaction caching in the case of item sequences is a bit different. For two filtered transaction to be equal, not only the items are important but their order as well. In other words, there are more requirements for equivalence, hence we do not expect so many contractions – and thus such speed-up – like in the case of itemsets. Also the increased number of different filtered transactions results in a larger trie and in a larger memory need.

3.6 Further Implementation Issues

In this section we briefly describe those techniques that are used in our FIM implementation and can be used in FSM directly or with slight modifications.

3.6.1 Candidate Sequences of Length One and Two

We use a counter vector and a counter array to determine the support of one- and two-element candidates. In the case of item sequences a bitvector and a bitarray is also required to avoid multiple increments of a candidate in transactions that contain the candidate many times. This means that each transaction is scanned twice, first for counter increments, second for reinitializing modified elements of the bitvector and bitarray. Note, that this second step requires insignificant time compared to the first step, and the parsing of the string representation of the transactions' elements to integers. This is again attributed to the hierarchical memory architecture of the processors. In the second step the transaction will still be in the first level cache (thus accessing its elements requires almost no time) and due to the small memory need of bitvectors and bitarrays, they will be in the worst case in the second level cache.

3.6.2 Stack-based Output

The FIMI competition has shown, that in very dense datasets with low support threshold (for example database `connect` with $min\_supp = 30000$), the procedures that output frequent itemsets affect running times significantly. Therefore we developed an output class, that spares slow integer to string conversions by applying a stack-based approach in storing string representations. Our stack-based approach suits well for depth-first algorithms. Although APRIORI is a breadth-first algorithm, outputting the result is done in a depth-first manner in the candidate generation step. Thus this class is used in our implementation. Further details and experimental results on this issue can be found in [16].

4. EXPERIMENTS

Due to the lack of public databases for testing frequent item sequence mining algorithms, we have generated some data from the weblog file of the largest Hungarian web news portal. Different generation techniques were applied to obtain databases with different characteristics. We make these databases publicly available and submit them to the OSDM repository. Beside this, we have used a FIM database `BMS-POS`, because its transactions are originally unordered and hence sequence mining make sense (although it does not contain any duplicates).

All implementations were tested on several min_supp values. A complete account of the results would require too much space, thus only the most typical ones are shown below. All

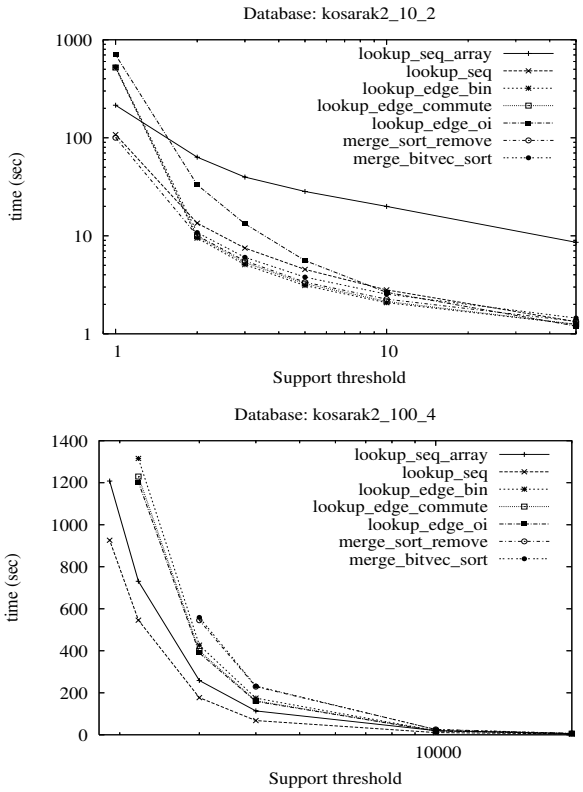


Figure 5: Routing strategies

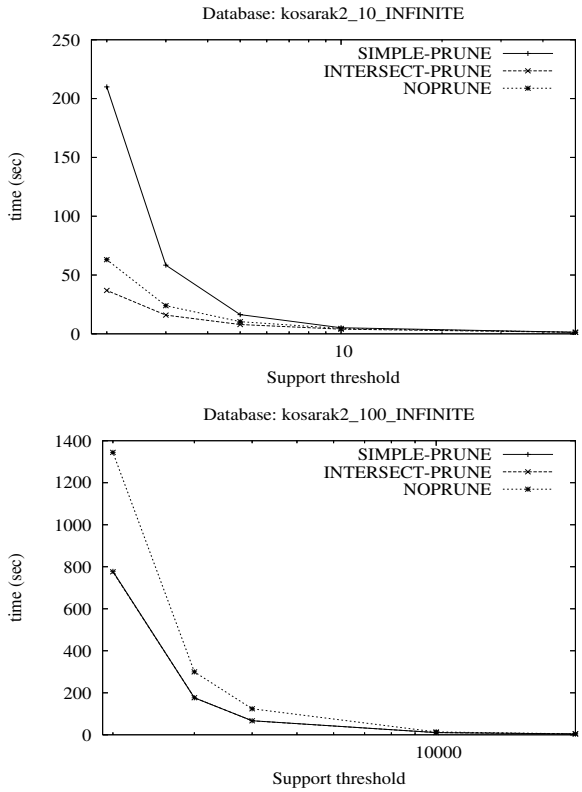


Figure 6: Candidate generation with different pruning strategies

results together with the test script can be downloaded from `http://www.cs.bme.hu/~bodon/en/fsm/test.html`.

Each measurement was taken on a workstation with Intel Pentium 4 2.8 Ghz processor (family 15, model 2, stepping 9) with 512 KB L2 cache, hyperthreading disabled, and 2 GB of dual-channel FSB800 main memory. The system runs a stripped-down installation of SuSE Linux 9.3, kernel 2.6.11.4-20a (SuSE version) with PerfCtr-2.6.15 patch installed. Run-times and memory usage were obtained using the GNU `time` and `memusage` command respectively.

First we tested the routing strategies. The notations were given in the description of the methods (see Sec. 3.2).

The results show that there exist no single routing strategy that always outperforms all the other methods, however, `lookup_seq` performs good most of the times. It always finishes in the first place on databases with long transactions, and also performs well when transactions are short. Concerning the other methods, we make the following observations:

- Method `lookup_seq_array` is not competitive at high support threshold (when the trie is small), especially not when the transactions are short. This is due to the overhead of building the index array. The results on database `kosarak2_10_2` meet our expectation; the smaller the *min_supp*, the better the relative performance of this method compared to the other solutions.
- In case of short transactions `merge_sort_remove` was the winner. Methods that perform sorting on the transactions (`merge_sort_remove` and `merge_bitvec_sort`), however, are not competitive with long transactions.
- The effectiveness of `merge_sort_remove` compared to `merge_bitvec_sort` depends on the transactions. Obviously, if the transaction contains many duplicates then `merge_bitvec_sort` performs better, because it performs sorts on a shorter lists.
- Method `lookup_edge_oi` performs bad when the transactions are small. This is due to the fact that this method requires more memory than linked-list-based solutions, hence the nodes are more scattered in the memory. This is not good for prefetching due to the lack of data locality, and also results many cache misses.

Figure 6 shows the running time of our APRIORI with different pruning strategies.

Intersection-based pruning always resulted in a faster implementation than simple-pruning (obviously in those cases when the support count procedure determined the running time, the difference was insignificant). The efficiency of pruning depends on the database characteristic. When the transactions are short then the support counting is fast, and the extra time spent on determining the support of candidates that have infrequent subsequences is less than applying complete pruning. This was the case with database

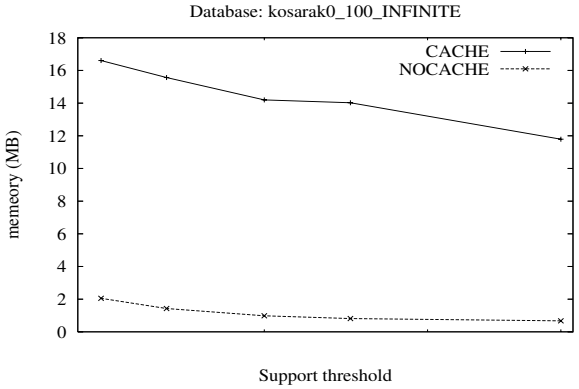


Figure 7: Transaction caching: effect on memory need

kosarak2_10_∞. The second database contained much longer transactions, hence determining the candidates in a transaction is much slower compared to determining the inclusion of a subsequence. In such cases, it is important to start support count as few times as possible. For such databases, complete pruning is advised.

Next, we have investigated the efficacy of the transaction caching (see Figure 7). The results show that transaction caching is by far not such an efficient technique in the case of item sequences like in the case of itemsets. It never resulted in a significantly faster algorithm, in the meantime it many time increased memory need seriously.

In our last experiments (Tables 1 and 2) we have investigated if our implementation is competitive with other FSM open source implementations. For this, we have used a prefixspan [13] implementation made by Taku Kudo. The source code can be downloaded from `http://chasen.org/~taku/software/prefixspan`, we have used the latest version (`0.4`) with parameters `-a -t int`. We denote the running time with ∞ if the program was stopped due to time limit (1500 seconds) exceed. In the tables, time is given in seconds and memory need in Mbytes.

Our APRIORI implementation always outperformed prefixspan with high support thresholds and also with low thresholds on databases with long transactions. This applies to running time and memory usage as well.

5. FURTHER IMPROVEMENTS

Our efforts have focused on building a FIM/FSM environment that is efficient and still flexible, in the sense that techniques can be switched on and off (for example omitting equisupport extension or transaction caching) and methods can be changed easily. This flexibility without computational penalty was reached by a class template based approach with inline functions. In this paper we have outlined and compared the basic possibilities. We believe, however, that by using more sophisticated solutions, the implementation can be improved further. Below are some issues that could be investigated.

Table 1: Comparison of FSM implementations: running times

implem.	min_supp			
	12000	2000	250	120
apriori	1.9	8.4	142.3	375.9
prefixspan	19.1	61.9	270.3	∞

BMP-POS

implem.	min_supp			
	200000	100000	60000	40000
apriori	6.2	9.7	18.2	69.4
prefixspan	42.5	117.4	678.2	∞

*kosarak*_100_∞

implem.	min_supp			
	50	5	2	1
apriori	1.5	5.1	14.1	108.7
prefixspan	3.7	4.6	6.2	27.0

*kosarak*2_10_2

implem.	min_supp			
	20000	10000	4000	2000
apriori	5.5	11.5	72.1	769.5
prefixspan	88.9	288.0	∞	∞

*kosarak*2_100_∞

Table 2: Comparison of FSM implementations: memory needs

implem.	min_supp			
	12000	2000	250	120
apriori	0.3	0.6	13.2	66.2
prefixspan	63.1	76.8	82.4	

BMP-POS

implem.	min_supp			
	200000	100000	60000	40000
apriori	0.6	0.6	0.6	1.0
prefixspan	124.8	124.9	130.8	

*kosarak*_100_∞

implem.	min_supp			
	50	5	2	1
apriori	2.7	21.6	75.6	249.0
prefixspan	15.9	16.1	16.1	16.4

*kosarak*2_10_2

implem.	min_supp			
	20000	10000	4000	2000
apriori	0.8	0.8	1.5	18.8
prefixspan	143.8	143.8		

*kosarak*2_100_∞

- Offsetindex and linked list based approaches can be combined to get a hybrid representation [6]. The selection of the approach can be made dynamically according to the number of the children. This way we could reach constant lookup time in some cases without sacrificing extra memory and avoid data scattering in the memory.
- Dynamic selection can also be applied in the routing strategies. The effectiveness of the different solutions depends on the size of the transactions, the number of children of nodes, the number of duplicates, etc. Characterizing the existing routing solutions, one may be able to set up an improved selection method.
- If an item of the transaction is not an element of any candidate then this item can be removed from the transaction. Processing a shorter transaction is faster, however, to get an overall performance improvement, we have to take into consideration the overhead of removing and reinserting a transaction into our database cacher (a Patricia tree in our case) as well.
- Current research [7] showed that trie based algorithms that perform their main operation in a depth-first manner can be accelerated by using a *cache-conscious trie*. Although APRIORI is called a breadth-first algorithm due to its search space traversal, the support count is done in a depth first manner, thus this technique is expected to reduce running time to its fourth.

6. CONCLUSION

In this paper we present how to modify a trie-based APRIORI algorithm for mining frequent item sequences from a transactional database. We also investigate the applicability of some well-known speed-up tricks, such as omitting prefix-equisupport extension, not applying complete pruning, etc. We have seen, that some parts of the algorithm do not have to be modified in the new pattern setting, while some techniques cannot be applied. We have described a wide assortment of routing strategies. In the analysis of most techniques we also considered the specialties of the modern processors, which has proved to be a more precise approach than simply calculating the required number of operations. Our results are summarized in the Table 3.

Table 3: Summary of the contributions

technique	FIM	FSM
dead-end pruning during candidate generation	possible	not possible
complete-pruning	in most cases unnecessary and slows down the algorithm	always speeds up the algorithm
omitting prefix-equisupport extension	possible	not possible
best routing strategy according to experiments	simultaneous traversal	for each label finding the corresponding item of the transaction
worst case comparisons of the best routing strategy	$n + \|t'\|$	$n \cdot \|t'\|$
influence of transaction caching on run-time	many times it results in a speed-up	it never resulted in a significant speed-up

7. ACKNOWLEDGEMENT

The author would like to thank Balázs Rácz, Lajos Rónyai and Lars Schmidt-Thieme for their helpful comments.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *The International Conference on Very Large Databases*, pages 487–499, 1994.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proc. 11th Int. Conf. Data Engineering, ICDE*, pages 3–14. IEEE Press, 6–10 1995.

[3] F. Bodon. A fast apriori implementation. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.

[4] F. Bodon. Surprising results of trie-based fim algorithms. In B. Goethals, M. J. Zaki, and R. Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 2004.

[5] F. Bodon and L. Schmidt-Thieme. The relation of closed itemset mining, complete pruning strategies and item ordering in apriori-based fim algorithms. In *Proceedings of the 9th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'05)*, Porto, Portugal, 2005.

[6] C. Borgelt. Efficient implementations of apriori and eclat. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.

[7] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, Y.-K. C. A. Nguyen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *Proceedings of the 31st International Conference on Very Large Date Bases (VLDB'05)*, Trondheim, Norway, 2005.

[8] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*,

volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.

[9] K. Hatonen, M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen. Knowledge discovery from telecommunication network alarm databases. In S. Y. W. Su, editor, *Proceedings of the twelfth International Conference on Data Engineering, February 26–March 1, 1996, New Orleans, Louisiana*, pages 115–122, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[10] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer-Verlag, 2000.

[11] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the first IEEE International Conference on Data Mining*, pages 313–320, 2001.

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

[13] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of the 17th International Conference on Data Engineering*, pages 215–224, Washington, DC, USA, 2001. IEEE Computer Society.

[14] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using patricia tries. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 2003.

[15] B. Rácz. nonordfp: An FP-growth variation without rebuilding the FP-tree. In B. Goethals, M. J. Zaki, and R. Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 2004.

[16] B. Rácz, F. Bodon, and L. Schmidt-Thieme. On benchmarking frequent itemset mining algorithms: from measurement to analysis. In B. Goethals, S. Nijssen, and M. J. Zaki, editors, *Proceedings of the ACM SIGKDD Workshop on Open Source Data Mining on Frequent Pattern Mining Implementations*, Chicago, IL, USA, 2005.

[17] L. Schmidt-Thieme. Algorithmic features of eclat. In B. Goethals, M. J. Zaki, and R. Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 2004.

Table 4: Some statistics of the databases

name	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{I}\rvert$	$\overline{\lvert t\rvert}$
kosarak2_10_∞	238 209	29 464	3
kosarak2_10_2	238 209	6 591	3
kosarak2_10_4	238 209	23 541	3
kosarak2_100_∞	604 280	71 260	16
kosarak2_100_2	604 280	14 288	16
kosarak2_100_4	604 280	54 225	16
kosarak_100_∞	820 771	38 593	11

[18] H. Toivonen. Sampling large databases for association rules. In *The VLDB Journal*, pages 134–145, 1996.

[19] T. Uno, M. Kiyomi, and H. Arimura. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In B. Goethals, M. J. Zaki, and R. Bayardo, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04)*, volume 126 of *CEUR Workshop Proceedings*, Brighton, UK, 2004.

APPENDIX

A. DATABASE OF SEQUENTIAL TRANSACTION

The following databases were generated from a weblog of a major Hungarian news portal by different filtering methods. The original raw database contained users' visits of four weeks. Each transaction belongs to a user, items represent a coded element of the portal. The items of the transaction are ordered by download time. The item that represents `index.html` was removed.

The names of the databases contain some information about the filtering method. In the name `kosarak2_x_y` the `x` stands for upper limit of the element of a transaction. Transactions with items more than `x` were removed. Variable `y` has connection with url handling, i.e. the part after the y^{th} backslash was cut of. The more this number is the more urls are distinguished. If `y` equals to ∞ then no urls were contracted.

Databases with `y=1` are dense datasets (and the distribution of the item's support is very steep) while databases with `y=`∞ are sparse ones. Table 4 gives the major parameters of the generated databases, i.e. number of transactions, number of items, average size of the transactions.

MoSS: A Program for Molecular Substructure Mining

Christian Borgelt
Dept. of Knowledge Processing and Language Engineering
University of Magdeburg
Universitätsplatz 2,
39106 Magdeburg, Germany
borgelt@iws.cs.uni-magdeburg.de

Thorsten Meinl
Computer Science Department 2
University of Erlangen-Nuremberg
Martenstraße 3,
91058 Erlangen, Germany
meinl@cs.fau.de

Michael Berthold
Dept. of Computer and Information Science
University of Konstanz
78457 Konstanz, Germany
berthold@inf.uni-konstanz.de

ABSTRACT

Molecular substructure mining is currently an intensively studied research area. In this paper we present an implementation of an algorithm for finding frequent substructures in a set of molecules, which may also be used to find substructures that discriminate well between a focus and a complement group. In addition to the basic algorithm, we discuss advanced pruning techniques, demonstrating their effectiveness with experiments on two publicly available molecular data sets, and briefly mention some other extensions.

1. INTRODUCTION

A frequent task in biochemistry is the search for common features in large sets of molecules. Examples are drug discovery, where one is interested in identifying properties shared by molecules that have been classified as "active" (and rarely shared by those classified as "inactive") w.r.t., for example, the protection of human cells against a virus, and compound synthesis, where one tries to identify properties that enable or inhibit the synthesis of new molecules, so that one can predict the chances for a successful synthesis.

Since the features one may use to describe molecules are manifold, there are approaches in abundance, ranging from simple one-dimensional measurements to complex thousand-dimensional descriptors. The molecular weight and the number of hydrogen donors or acceptors are examples of simple features. More complex ones include binary vectors, which can be several thousand bits long with each bit representing a specific constellation of atoms like aromatic rings or amino groups, as well as shape descriptors that try to capture geometric properties of a molecule.

In this paper we focus on an approach that models molecules as attributed graphs, thus taking the connection structure, though not the 3-dimensional structure into account. The resulting set of graphs is then searched for common subgraphs, that is, molecular fragments that appear with a user-specified minimum frequency. For this approach several algorithms have been proposed recently, with many of them based on methods developed for association rule mining. In particular, the Apriori algorithm [1] and the Eclat algorithm [21] are taken as starting points. The general ideas of these algorithms can be transferred to molecular substructure mining, even though the fact that the input consists of graphs instead of sets poses some problems. Examples of algorithms developed in this way are Subdue [5], MolFea [12], FSG [13], MoFa [2], gSpan [20], FFSM [9], and Gaston [16]. Other approaches rely, for instance, on principles from inductive logic programming [6] and describe the graph structure by logical expressions.

Of course, all of the mentioned approaches are generally applicable to attributed graphs, with molecules being only one specific application example. However, the Java implementation discussed in this paper is geared to molecules as it supports reading molecules in standard description languages like SMILES (Daylight, Inc.) and SLN (Tripos, Inc.). It also has to be mentioned that the first version of the presented program was developed in cooperation with the biochemical software company Tripos, Inc., St. Louis, MO, USA. Finally, it should be noted that the algorithm described here is also known as MoFa (**Mo**lecular **Fra**gment miner). The Java program presented here, however, is called MoSS (**Mo**lecular **S**ub**S**tructure miner) in order to distinguish it from other implementations.

2. BASIC ALGORITHM

In order to capture the bond structure of molecules, we model them as attributed graphs, in which each vertex represents an atom and each edge a bond between atoms. Each vertex carries attributes that indicate the atom type (i.e., the chemical element), a possible charge, and whether it is part of an aromatic ring. Each edge carries an attribute that indicates the bond type (single, double, triple, or aromatic).

Our goal is to find substructures that have a certain minimum support in a given set of molecules, i.e., are part of at least a certain percentage of the molecules. However, in order to restrict the search space, we usually consider only

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

connected substructures, i.e., graphs having only one connected component. For most applications, this restriction is harmless, because connected substructures are most often exactly what is desired. (Note, however, that the program considered here also allows for starting from a disconnected core structure, though not for extending a substructure by an unconnected atom.) We do *not* constrain the connectivity of the graph in any other way: The graphs may be chains or trees or may contain an arbitrary number of cycles.

2.1 Search Tree Traversal

Most naturally, the search is carried out by traversing a tree of fragments of molecules, similar to the tree of item sets that is traversed in frequent item set mining. The root of the tree is a core structure to start from, which for now we assume to be a single atom (more complex cores are discussed below). Going down one level in the search tree means to extend a substructure by a bond (and maybe an atom, if the bond does not close a ring), just like going down in an item set tree means adding an item to an item set. That is, with a single atom at the root of the tree, the root level contains the substructures with no bonds, the second level those with one bond, the third level those with two bonds and so on.

There are basically two ways in which such a search tree can be traversed: We can use either a breadth first search and explicit subset tests (like Apriori) or a depth first search and intersections of transaction lists (like Eclat). For our task the Eclat approach seems preferable, because the Apriori approach has certain drawbacks: even subset tests can be costly, but substructure tests, which consist mathematically in checking whether a given attributed graph is a subgraph of another attributed graph, can be extremely costly. Furthermore, the number of small substructures (1 to 4 atoms) can be enormous, so that even storing only the topmost levels of the tree can require a prohibitively large amount of memory. Of course, the Eclat approach also has its drawbacks, for example, the transaction lists are now lists of embeddings of a substructure into the given molecules. Since there can be several embeddings of the same substructure into one molecule, these lists tend to get fairly long. This drawback can make it necessary to start from a reasonably sized core structure (see below).

To be more specific, our algorithm searches as follows: The given core structure is embedded into all molecules, resulting in a list of embeddings. Each embedding consists of references into a molecule that point out the atoms and bonds that form the substructure. Remember that a list of embeddings may contain several embeddings for the same molecule if the molecule contains the substructure in more than one place or if the substructure is symmetric. In a second step each embedding is extended in every possible way. This is done by adding all bonds in the corresponding molecule that start from an atom already in the embedding (to ensure connectedness and, of course, to reduce the number of bonds that have to be considered). This may or may not involve adding the atom the bond leads to, because this atom may or may not be part of the embedding already. More technically, by following the references of an embedding the atoms and bonds of the corresponding molecule are marked and only unmarked bonds emanating from marked atoms are considered as possible extensions.

The resulting extended embeddings are then sorted into equivalence classes, each of which represents a new substructure. This sorting is very simple, because only the added bond and maybe the added atom have to be compared. In our implementation we use a sorted array of lists of embeddings to group the extensions. After all extended embeddings have been processed, each array element contains the list of embeddings of a new substructure. Each of these new substructures corresponds to a child node in the search tree, each of which is then processed in turn by searching recursively on the list of embeddings corresponding to it.

2.2 Basic Search Tree Pruning

Of course, subtrees of the search tree can be pruned if they refer to substructures not having enough support, i.e., if too few molecules are referred to in the associated list of embeddings. We call this *support based pruning*. We may also prune the search tree if a user-defined threshold for the number of atoms in a fragment has been reached. We call this *size based pruning*. However, the most important pruning type is *structural pruning*, which is meant to ensure that each substructure is considered only once in the search tree. In frequent item set mining such structural pruning can easily be achieved by drawing on an (arbitrary, but fixed) order of the items and disallowing extensions by items preceding the item added last to the set (cf. [3] for more details).

In molecular substructure mining, however, such an approach is impossible, since there is no possible *global* order of, say, the atoms, as we have to take the bond structure into account. However, we can number the atoms *in a substructure* and record how a substructure was constructed in order to constrain its extensions. The number we assign to an atom reflects the step in which it was added. That is, the core atom is numbered 0, the atom added with the first bond is numbered 1 and so on. Note that this number does not tell anything about the type of the atom, as two completely different atoms may receive the same number, simply because they were added in the same step.

Whenever an embedding is extended, we record in the resulting extension the number of the atom from which the added bond started. When the extended embedding is to be extended itself, we consider only bonds that start from atoms having numbers no less than this recorded number. That is, only the atom extended in the preceding step and atoms added later than this atom can be the starting point of a new bond. This rule is directly analogous to the rule that only items following the item added last may be added to an item set. With this simple scheme we immediately avoid that two bonds, call them A and B, which start from different atoms, are added in the order A, B in one branch of the search tree and in the order B, A in another. Since either the atom A starts from or the atom B starts from must have a smaller number, one of the orders is ruled out.

However, two or more bonds can start from the same atom. Therefore we also have to define an order on bonds, so that we do not add two different bonds A and B that start from the same atom in the order A, B in one branch of the search tree and in the order B, A in another. This order on bonds is, of course, arbitrary. In our implementation, single bonds precede aromatic bonds, which precede double bonds, which

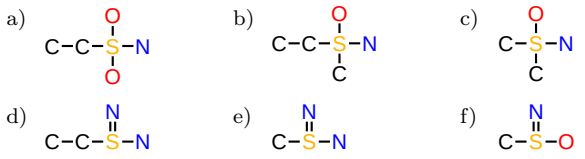


Figure 1: A set of six example molecules.

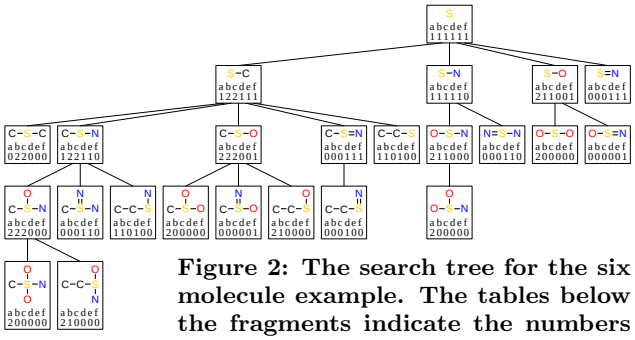


Figure 2: The search tree for the six molecule example. The tables below the fragments indicate the numbers of embeddings per molecule.

precede triple bonds. Finally, within extensions by bonds of the same type starting from the same atom, the order is determined by (1) whether the atom the bond leads to is already in the substructure or not and (2) the type of this atom. To take care of the bond type etc., we record in each embedding which bond was added last.

As a final rule, we disallow extensions by bonds leading to an atom already in the substructure if the number of the destination atom is higher than the number of the source atom. All bonds leading to already captured atoms (that is, all bonds closing rings) must lead "backward", that is, from an atom with a higher number to an atom with a lower one.

The above rules provide us with a structural pruning scheme, but unfortunately this scheme is not perfect and making it perfect would be very expensive computationally. The problem is that we do not have any precedence rule for two bonds of the same type starting from an atom with the same number and leading to atoms of the same type, and that it is not possible to give any precedence rule for this case that is based exclusively on locally available information. We consider the problems that result from this imperfection and our solution below, but think it advisable to precede this consideration by an illustrative example of the search process as we defined it up to now.

2.3 An Illustrative Example

As an illustration we consider how our algorithm finds the frequent substructures of the six example molecules shown in Figure 1[1], starting from a sulfur atom. We use a minimum support of 50%, i.e., a substructure must occur in at least three of the six molecules to qualify as frequent.

[1]Please note that these structures were made up to demonstrate certain aspects of the search scheme. None of them has any real (i.e., chemical) meaning.

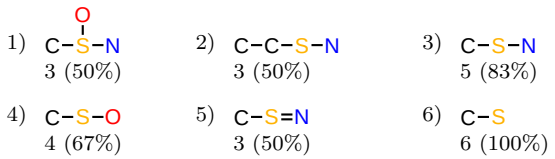


Figure 3: The six frequent substructures that are found in the order in which they are generated.

First the sulfur atom is embedded into the six molecules. This results in six embeddings, one for each molecule, which form the root of the search tree (see Figure 2; the table in the root node records that there is one embedding for each molecule). Then the embeddings are extended in all possible ways, which leads to the four different substructures shown on the second level (i.e., S–C, S–N, S–O, S=N). These substructures are ordered, from left to right, as they are considered by our algorithm, i.e., extensions by single bonds precede extensions by double bonds, and within extensions by bonds of the same type the element type of the atom a bond leads to determines the order. Note that there are two embeddings of S–C into both the molecules b and c and two embeddings of S–O into the molecule a.

In the third step the extensions of the substructure S–C are constructed. This leads to the first five substructures on the third level (i.e., C–S–C, C–S–N, C–S–O, C–S=N and C–C–S). Again the order of these substructures, from left to right, reflects the order in which they are considered. Since we search depth first, the next substructure to be extended is C–S–C.[2] However, this substructure does not have enough support and therefore the subtree is pruned.

The substructure C–S–N is considered next etc. However, we confine ourselves to pointing out situations in which specific aspects of our method become obvious. Effects of structural pruning can be seen, for instance, at the fragment C–S–N, which does not have a child in which a second carbon atom is attached to the sulfur atom. The reason is that the extension by the bond to the nitrogen atom rules out all single bonds leading to atoms of a type preceding nitrogen (like carbon). Similarly, C–S=N does not have children with another atom attached to the sulfur atom by a single bond, not even an oxygen atom, which follows nitrogen in the periodic table of elements. The reason is that a double bond succeeds a single bond and thus the extension by the double bond to the nitrogen atom rules out all single bonds emanating from the sulfur atom. Finally, the structure C–C–S has no children at all, even though it has enough support. The reason is that in this substructure a bond was added to the carbon atom adjacent to the sulfur atom. This carbon atom is numbered 1 and thus no bonds can be added to the sulfur atom, which has number 0. Only the carbon atoms can be starting points of a new bond, but there are no such bonds in the molecules a, b, and d. Note that ruling out extensions of the sulfur atom in this branch is not harmful, since all fragments having another atom attached to the sulfur atom are considered in the subtrees to the left of C–C–S.

[2]It may seem strange that there are two embeddings of this substructure into both the molecules b and c. The reason for this is explained below.

During the recursive search all frequent substructures encountered are recorded. The resulting set of six frequent substructures, together with their absolute and relative support is shown in Figure 3. Note that C–C–S is not reported, because it has the same support as its superstructure C–C–S–N. Likewise, O–S–N, S=N, and S are not reported. Transferring a common term from frequent item set mining, we may say that our algorithm reports only *closed fragments*, that is, fragments no superstructure of which has the same support (see also below, Section 3.1).

The example makes it clear that our algorithm can find arbitrary substructures, even though it does not show how cyclic structures are treated. Unfortunately, search trees for cyclic structures are too big to be depicted here.

2.4 Incomplete Structural Pruning

We indicated above that our structural pruning is not perfect. In order to understand the problems that can arise, consider two molecules A and B with the common substructure N–C–S–C–O. We try to find this substructure starting from the sulfur atom. Since the two bonds emanating from the sulfur atom are equivalent, we have no precedence rule and thus the order in which they are added to an embedding depends on the order in which they occur in the corresponding molecule (which we have no real control over) .

Suppose that in molecule A the bond going to the left precedes the bond going to the right, while in molecule B it is the other way round. As a consequence, in embeddings into molecule A the left carbon atom will precede the right one, while in embeddings into molecule B it will be the other way round. Now consider the substructure C–S–C and its extensions. In molecule A the carbon numbered 1 (the left one) will be extended by adding the nitrogen atom and thus the oxygen atom can be added in the next step (to the carbon on the right, which is numbered 2), resulting in the full substructure. However, in molecule B the nitrogen atom has to be added by extending the carbon atom numbered 2 (again the left one; in embeddings into molecule B the right carbon is numbered 1). Hence it is not possible to add the oxygen atom in the next step, because this would mean adding a bond starting at an atom with a lower number than the atom extended in the preceding step. Therefore the common substructure is not found. This example also shows that it does not help to look "one step ahead" to the next atom, because there could be arbitrarily long equivalent chains, which differ only at the ends. There are no "local" criteria, which would enable us to decide how to order and thus number equivalent extensions of the same atom.

If, however, we accept to reach identical substructures in different branches of the search tree in cases like this, we can correct the imperfection of our structural pruning. Whenever we extended an embedding by following a bond, we allow adding an equivalent bond in the next step, regardless of whether it precedes or succeeds, in the corresponding molecule, the bond added in the preceding step. This relaxation explains why there are two embeddings of the substructure C-S-C into both the molecules b and c of our example. In one embedding the left carbon atom is numbered 1 and the one at the bottom is numbered 2, while in the other it is the other way round (cf. Figure 1).

Note that considering the same substructure several times cannot lead to wrong results, only to multiple reporting of the same substructure. Multiple reporting, however, can be suppressed by maintaining a list of frequent substructures and suppressing new ones that are identical to already known ones. It is more important that the missing rule for equivalent bonds can lead to considerable redundant search in certain structures, especially molecules containing one or more aromatic rings. (A solution to this is discussed below.)

However, it should be noted that even if we could amend the weakness of our structural pruning, we would still be unable to guarantee that each substructure is considered only once. If, for instance, some substructure X can be embedded twice into some molecules and if there are frequent substructures that contain both embeddings (and thus X twice), then these substructures can be grown from either embedding. If the connection between the two embeddings of X is not symmetric, the same substructure is reached in two different branches of the search tree in this case. Obviously, there is no simple way to avoid such situations.

2.5 Embedding a Core Structure

Up to now we assumed that we start the search from a single atom. This usually works fairly well as long as this atom is rare in the molecules to work on. For example, sulfur or phosphorus are often good starting points in biochemical applications, while starting with carbon is a bad idea: Every organic molecule contains several carbon atoms, often twenty or more, and thus we end up with an already very high number of embeddings of the initial atom. As a consequence, the algorithm is likely to run out of memory before reaching substructures of reasonable size.

However, if we cannot start from a rare element, it is sometimes possible to specify a core—for instance, an aromatic ring with one or two side chains—from which the search can be started. Provided the core structure is specific enough, there are only few, at best only one embedding per molecule, so that the list of embeddings is short.

While it is trivial to embed a single atom into a molecule, embedding a core structure can be much more difficult. In our implementation we rely on a simple observation: embedding a core structure is the same as finding a common substructure of the molecule and the core that is as big as the core itself. This leads to the idea to grow a substructure into both the core and the molecule until it completely covers the core. That is, we do a substructure search for the core and the molecule starting from an arbitrary atom of the core and requiring a support of 100% (i.e., both the core and the molecule must contain the substructure). In addition, we can restrict the search to one embedding of a substructure into the core at all times, since we know that it must be completely covered in the end. (For the molecule, however, we must consider all possible embeddings.)

Note that the same mechanism of growing a substructure into two molecules can also be used for substructure tests as they are needed to suppress multiple reporting of the same fragment (see above) as well as reporting redundant fragments (fragments that are substructures of some other fragment and have the same support as this fragment).

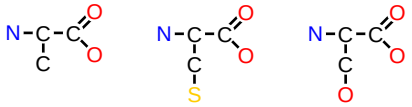


Figure 4: The amino acids clycin, cystein and serin.

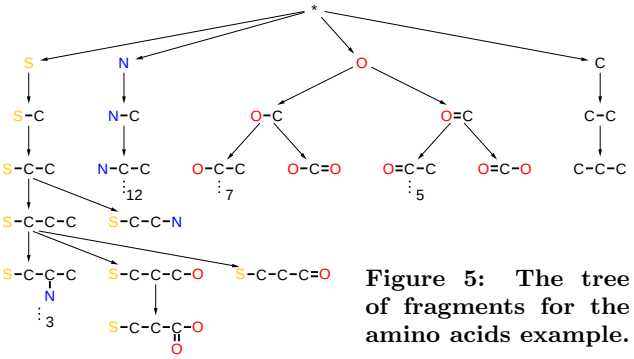


Figure 5: The tree of fragments for the amino acids example.

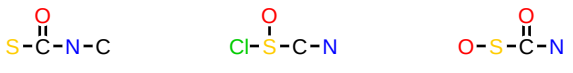


Figure 6: Some (fictitious) example molecules.

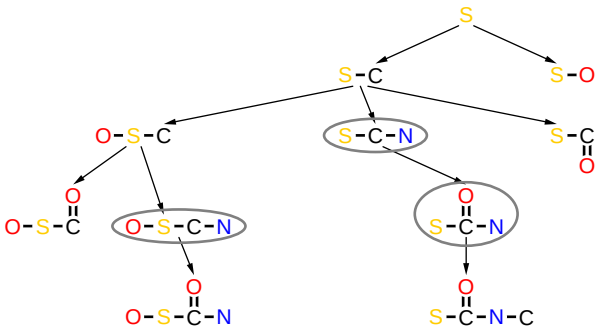


Figure 7: Search tree for the molecules in Figure 6.

2.6 Starting with an Empty Core

Up to now we assumed that we are given a core structure to start the search from. However, in [7, 8] an extension to empty cores was suggested. We explain this approach with a simple example. Figure 4 shows the amino acids clycin, cystein and serin. The upper part of the tree (or forest if the empty fragment at the root is removed) that is traversed by our algorithm for these molecules is shown in Figure 5. The first level contains individual atoms, the second connected pairs and so on. The dots indicate subtrees that are not depicted in order to keep the figure understandable. The numbers next to these dots state the number of fragments in these subtrees, indicating the total size of the tree.

The order, in which the atoms on the first level of the tree are processed, is determined by their frequency of occurrence in the molecules. The least frequent atom type is considered first. Therefore the algorithm starts on the left by embedding a sulfur atom into the example molecules. That is, the molecules are searched for sulfur atoms and their locations are recorded. In our example there is only one sulfur atom in cystein, which leads to one embedding of this (one atom) fragment. This fragment is then extended in the way described above. Next the subtree rooted at the nitrogen atom is processed. However, in this subtree extensions by a bond to a sulfur atom are ruled out, since all fragments containing a sulfur atom have already been considered in the tree rooted at the sulfur atom. Similarly, neither sulfur nor nitrogen are considered in the tree rooted at the oxygen atom, and the rightmost tree contains fragments that consist of carbon atoms only. The program offers the option to skip this last tree, because for biochemical molecules it can be very large and thus may govern the search time.

3. ADVANCED PRUNING

In the preceding section we considered basic search tree pruning, which is meant to ensure that each substructure is considered as few times as possible. In this section we discuss further optimizations, suggested in [14, 4], which consist in two advanced pruning strategies, namely *equivalent sibling pruning* and *perfect extension pruning*. The first of these optimizations can be applied even if the search is not restricted to closed fragments (see below), while the second presupposes that non-closed fragments can be discarded.

3.1 Closed Molecular Fragments

As already mentioned above, the notion of a *closed fragment* is derived from the corresponding notion of a *closed item set*, which is defined as an item set no superset of which has the same support, i.e., is contained in the same number of transactions. Analogously, a *closed fragment* is a fragment no superstructure of which has the same support, i.e., is contained in the same number of molecules. As an example consider the three example molecules shown in Figure 6 and the corresponding (unpruned) MoFa search tree (starting from sulfur as a seed) shown in Figure 7: the closed fragments (for a minimum support of two molecules, i.e., 66%) in inner nodes are circled. (It should be clear that every leaf of the search tree is necessarily a closed fragment).

As for item sets, restricting the search for molecular fragments to closed fragments does not lose any information: all frequent fragments can be constructed from the closed ones by simply forming all substructures of closed fragments that are not closed fragments themselves and assigning to them as their support the maximum of the support values of those closed fragments of which they are substructures. Consequently, closed fragment mining is a very convenient way to reduce the size of the output. In addition, chemists are usually not interested in all frequent fragments, but only in the closed ones, presumably because they contain all relevant information without redundancy. Here, however, we are interested in the fact that in closed fragment mining certain pruning techniques become applicable, which can speed up the search considerably. These and other advantages of closed fragment mining were also studied in [20, 14, 4].

3.2 Equivalent Sibling Pruning

In order to suppress all redundant search, one would have to check whether any two subtrees of the search tree have the

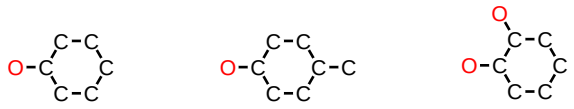


Figure 8: Three phenols: phenol, p-cresol, and catechol.

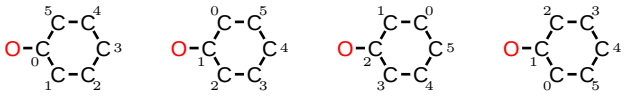


Figure 9: Equivalent extended embeddings.

same fragment as their root (or fragments which only differ in the restrictions placed on their possible extensions), so that only one (namely the least restricted one) is actually searched. Such a general check, however, is costly, because one would have to store all fragments that have been considered in the search so far. In addition, one needs an efficient way of checking whether these stored fragments contain one that is equivalent to the one currently considered. Finally, it is difficult to find out whether a subtree to be searched is the least restricted one appearing in the whole search tree, even though the depth first traversal applied in our algorithm always considers the least restricted sibling node first.

However, what can be checked fairly easily is whether *two sibling nodes* in the search tree correspond to fragments that are equivalent (represent the same substructure) and differ only in the restrictions placed on their possible extensions. That such a situation can actually occur can be seen from the example molecules shown in Figure 8. Suppose we start the search by embedding a benzene ring into these molecules. Since such a ring exhibits a high symmetry (which is a necessary prerequisite for equivalent siblings), it can be embedded into each of the molecules in twelve different ways (the ring can be rotated to six positions and it can be traversed in two directions). Consider now the extensions of this benzene ring: each of the twelve embeddings is extended by a bond and an oxygen atom. This leads to twelve new fragments, all of which are equivalent and differ only in the label of the ring atom that was extended (see Figure 9 for examples). Obviously it suffices in this situation to consider the fragment in which the ring atom with the smallest number was extended. All other fragments must lead to redundant search, because they allow for a subset of the possible extensions, but no additional ones.

Since our algorithm considers sibling nodes w.r.t. a local order that is based on the extension information of the parent fragment (i.e. bond and atom type, number of the atom the bond is incident to) and processes the least restricted extensions first, it is fairly easy to implement the described pruning approach: for each node its preceding sibling nodes are checked for equivalence and if an equivalent sibling is found, the current node is skipped.

The equivalence test is carried out as follows: for the fragment corresponding to the current node an arbitrary embedding into an arbitrary molecule is selected. In the corresponding molecule all bonds and atoms belonging to this embedding are marked and all other bonds and atoms are unmarked. Then all embeddings of the fragment corresponding to a sibling node, which is to be checked for equivalence, are traversed. For each of these embeddings it is checked whether it refers only to marked atoms and bonds, that is, whether it essentially represents the same substructure. If such an embedding can be found, the two fragments are equivalent and consequently the current node (the extensions of which are more severely restricted than those of its already processed sibling) can be skipped.

Note that it suffices to check one molecule, because if there are identical embeddings into one molecule there must be identical embeddings into all molecules referred to by the fragment. As a consequence the check is comparatively cheap and thus does not degrade performance much even if the pruning cannot be carried out.

3.3 Perfect Extension Pruning

Perfect extension pruning is based on the observation that sometimes there is a fairly large common fragment in *all* currently considered molecules. As long as the search has not grown a fragment to this maximal common one, it is not necessary to branch in the search tree.

From the definition of a closed fragment it is clear that if there is a common substructure in *all* currently considered molecules of which the current fragment is only a part, then any extension that does not grow the current fragment towards the maximal common one can be postponed until this maximal common fragment has been reached. The reason is, obviously, that the maximal common fragment is part of all closed fragments that can be found in the currently considered set of molecules. Consequently, it suffices to follow one path in the search tree that leads to this maximal common fragment and to start branching only from there.

As an example consider again the set of molecules shown in Figure 6. If the search is seeded with a single sulfur atom, considering extensions by a single bond starting at the sulfur atom and leading to an oxygen atom can be postponed until the structure `S-C-N` common to all molecules has been grown (provided that the extensions of this maximal common fragment are not restricted in any way—see below).

Technically, the search tree pruning is based on the notion of a *perfect extension*. An extension of a fragment, consisting of a bond and possibly an atom (if the bond does not close a ring), is called *perfect* if all of its embeddings can be extended in exactly the same way by this bond and atom and if it is a *bridge*[3] in all embeddings into the molecules or if it closes a ring. (Note that there may be multiple ways of extending an embedding by this bond and atom. In this case all embeddings must be extendable in the same number of ways and in all the bond must be a bridge or must close a ring. The additional condition that the bond has to be a bridge or must close a ring in all embeddings was unfortunately not recognized in [4].) If there is a perfect ex-

[3] A bridge in a graph is an edge that, if removed, splits the graph into two unconnected parts.

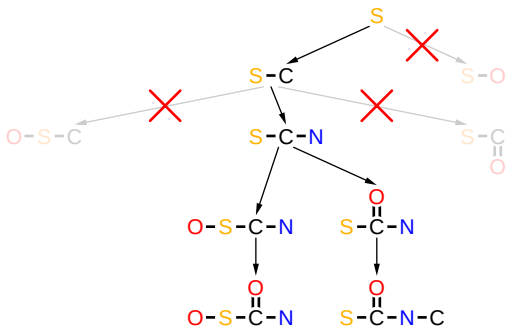


Figure 10: Search with perfect extension pruning.

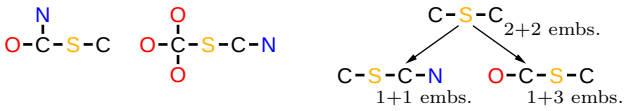


Figure 11: Examples of non-perfect extensions.

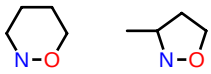


Figure 12: Perfect extensions may fail in rings.

tension of a fragment, all closed fragments can, in principle, be found by searching only the corresponding branch.

There is, however, a minor complication, because perfect extension pruning interferes with the normal structural pruning done in MoFa. Normal structural pruning prevents extensions of a fragment by bonds that start from atoms with smaller numbers than the one extended in the preceding step. However, a perfect extension should not lead to such a restriction, because otherwise some search results are lost.

As an example consider again the search tree shown in Figure 7. If we simply confined the search to the subtree rooted at the fragment `S-C-N`, we would lose the fragment `O-S-C-N` in the left branch. The reason is that the extension of `S-C` to `S-C-N`, due to the normal structural pruning rules, prevents an extension of the sulfur atom in this subtree, because an atom with a higher number, namely the carbon atom, has been extended in the preceding step.

However, a perfect extension should not restrict possible future extensions of the fragment in this way. Therefore the extension information of a fragment obtained by a perfect extension are set to those of its parent fragment, bypassing the normal structural pruning rules. In the considered example, this allows us to extend the fragment `S-C-N` by bonds starting from the sulfur atom and thus we get the search tree shown in Figure 10, in which the fragment `O-S-C-N` is found.

When it comes to implementing perfect extension pruning, one should bear in mind that checking whether an extension is perfect by testing the extensions of all embeddings is costly. Therefore we precede the actual test by cheap tests in order to, if possible, *fail early* or *succeed early*. The ideas underlying these tests are fairly simple: an extension leading to a fragment cannot be perfect (1) if the number of molecules referred to by the fragment differs from those referred to by its parent and (2) if the number of embeddings of the fragment is not an integer multiple of the number of embeddings of its parent. On the other hand, if these tests do not indicate that the extension is not perfect, we can immediately conclude that it is perfect if the fragment refers only to one molecule. The costly test whether each parent embedding leads to the same number of extended embeddings is, of course, carried out only if the number of molecules referred is not equal the number of embeddings (otherwise there is exactly one embedding per molecule). Only afterwards we test whether the bond either closes a ring or is a bridge in all embeddings. (Bridges are, of course, found and marked in a preprocessing step, so that the test only has to check this flag for the bonds referred to by the embeddings.)

Note that it is actually necessary to count the embeddings per molecule. Checking only whether the total number of embeddings into all molecules coincides with (or is an integer multiple of) the parent embeddings does not suffice as can be seen from the example shown in Figure 11. Even though the total number of embeddings is the same in the right branch, the extension is not perfect. The left extension is not perfect, because the number of extended embeddings, even though the same for each parent embedding, is reduced from the number of extensions of its parents. This indicates that some symmetry has been destroyed by the extension, which therefore cannot be perfect.

Why it is necessary to constrain perfect extensions to bridges if the extension does not close a ring can be seen from the example in Figure 12. Suppose we want to find all common substructures of these two molecules (minimum support 100%). We seed the algorithm with a nitrogen atom. Then adding the oxygen atom is a perfect extension in the sense that it can be done in the same way in all molecules. The same holds for a subsequent extension of the oxygen atom with a carbon atom, or generally, to walk around the rings in counterclockwise order. This yields the fragment `N-O-C-C-C-C` as the only result. However, `C-C-N-O-C-C` and `C-C-C-N-O` are also common substructures. These we would lose if we allowed such extensions in rings to be considered as perfect.

4. OTHER EXTENSIONS

Apart from the advanced pruning strategies described above, there are several ways in which the basic algorithm can be extended, some of which are already incorporated in the Java implementation referred to here. Due to reasons of space these extensions are only briefly mentioned here. Details can be found in [2, 7, 8, 14, 15].

4.1 Finding Discriminative Fragments

Our approach to find frequent substructures can easily be extended to find *discriminative fragments*, that is, substructures that are frequent in a predefined subset of the molecules and infrequent in the complement of this subset. Finding discriminative fragments requires two parameters: a minimum support for the focus subset and a maximum support for the complement. The search is carried out in exactly the same way as described above. The only difference is that two

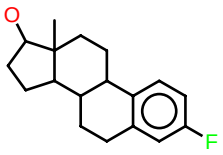


Figure 13: Example of a steroid, which can lead to problems due to the large number of rings.

Figure 14: Aromatic and Kekulé representations of benzene.

support numbers are determined: one for the focus subset and one for the complement. Only the support in the focus subset is used to prune the search tree. The support in the complement determines whether a frequent substructure is recorded or not, thus filtering out substructures that do not satisfy the requirements for a discriminative structure.

4.2 Rings

An unpleasant problem of the search algorithm as we presented it up to now is that the local ordering rules for suppressing redundant searches are not complete. Two extensions by identical bonds leading to identical atoms cannot be ordered based on locally available information alone (see above). As a consequence, a certain amount of redundant search has to be accepted in this case, because otherwise incorrect support values would be computed and it could not be guaranteed that all frequent substructures are found.

Unfortunately, situations leading to such redundant search occur almost always in molecules with rings, whenever the fragment extension process enters a ring. Consequently, molecules with a lot of rings, like the steroid shown in Figure 13—despite all clever search tree pruning—still lead to a prohibitively high search complexity.

To cope with this problem, we added an (optional) preprocessing step to the algorithm, in which the rings of the molecules are found and marked. The implementation described here offers the possibility to specify a range for the ring size, so that this feature can be restricted to chemically meaningful rings, for example, of size 5 and 6. In the search process itself, whenever an extension adds a bond to a fragment that is part of a ring, not only this bond, but the whole ring is added (all bonds and atoms of the ring are added in one step). As a consequence, the depth of the search tree is reduced (the basic algorithm needs five or six levels to construct a ring bond by bond) and many of the redundant searches of the basic algorithm are avoided, which leads to enormous speed-ups. Details can be found in [7, 8].

Besides a considerable reduction of the running time, treating rings as units when extending fragments has other advantages as well. In the first place, it makes a special treatment of atoms and bonds in rings possible, which is especially important for aromatic rings. For example, benzene can be represented in different ways as shown in Figure 14. On the left, all six bonds are represented as aromatic bonds, while the other two representations—called

pruning	# nodes	# frags.	# embs.
neither	48762	48762	78209
equivalent sibling	22423	23122	38475
perfect extension	4355	6581	16731
both	1577	2947	7786

Table 1: Effects of pruning on the steroids data set.

Kekulé structures—use alternating single and double bonds. A chemist, however, considers all three representations as equivalent. With the basic algorithm this is not possible, because it would require treating single, double, and aromatic bonds alike, which obviously leads to undesired side effects. By treating rings as units, however, these special types of rings can be identified in the preprocessing step and then be transformed into a standard aromatic ring.

4.3 Variable Length Chains

Often the exact length of a carbon chain in a molecule is not important, as long as it is within a certain range. In this case, the fragment may contain only an indication of a chain, which match a certain number of chained carbon atoms in a molecule, where the number may differ for different molecules. This type of imprecise match can be handled in a similar way as the ring extension we studied in the previous section: Instead of bond by bond, a carbon chain is added in one step, allowing for different lengths. The implementation described here offers this possibility, but does not yet take a range for the acceptable lengths of the chain.

4.4 Wildcard Atoms

In principle it is possible to extend the search algorithm to handle wildcard atoms, which may match any atom in a user-defined group of atoms. Details can be found in [7, 8]. This possibility is available in a sibling implementation of the algorithm described here. This implementation is, however, property of Tripos, Inc., and thus cannot be made available. The Java implementation described here currently lacks this capability.

5. EXPERIMENTAL RESULTS

In our experiments we restrict ourselves to demonstrating the effects of the two advanced pruning approaches described above. We carried out experiments on three data sets, all of which are publicly available. The first is a very small data set consisting of 17 steroid molecules.[4] The effects of the two pruning methods are shown in Table 1 (all experiments were carried out with a minimum support of one molecule). As can be seen, both pruning methods have considerable effects, with those of perfect extension pruning being much more pronounced. However, this is presumable due to the very low minimum support, which makes perfect extension pruning highly effective. At higher support values equivalent sibling pruning seems to be more effective (see below).

In a second test we ran the program on the well-known HIV data set available from the National Cancer Institute [10]. This library contains about 44,000 molecules tested for their activity against the HI-virus, which are grouped into three

[4]See Section 7 below for a download URL.

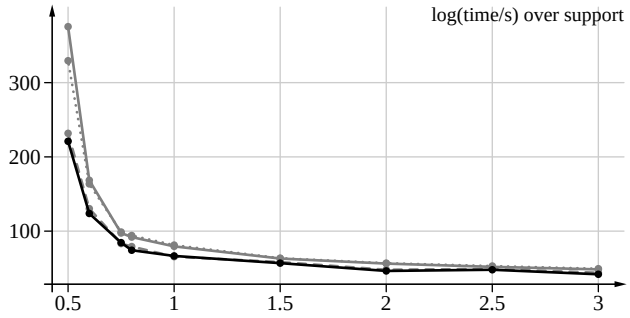


Figure 15: Execution times on NCI's HIV data.

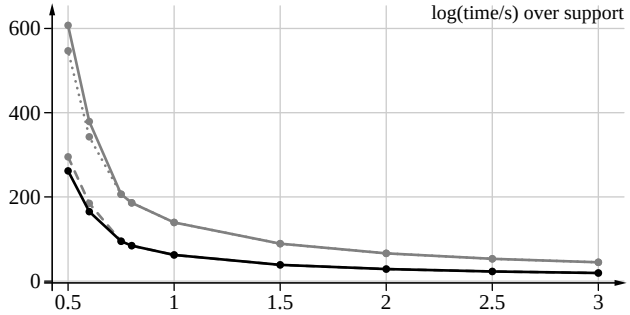


Figure 16: Numbers of nodes on NCI's HIV data.

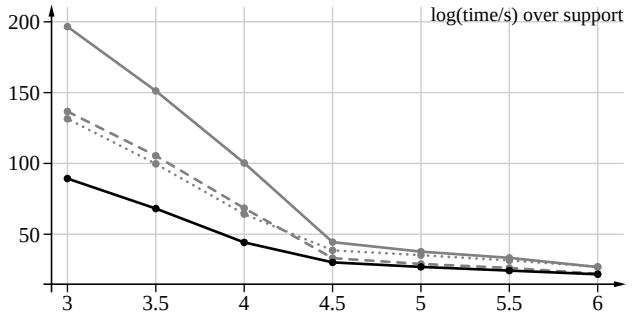


Figure 17: Execution times on the IC93 data.

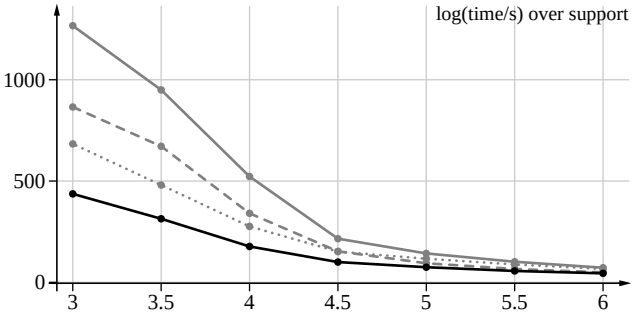


Figure 18: Numbers of nodes on the IC93 data.

classes: about 400 belong to class CA (confirmed active), about 1000 to CM (confirmed medium active) and the rest belongs to CI (confirmed inactive). In the experiments we report here, however, we neglected these class assignments and tried to find common substructures of all molecules at minimum support thresholds ranging from 0.7 to 3%. In addition, we tested on the IC93 data set [11]. For all experiments we used a Pentium 4C 2.6GHz system with 1GB of main memory running S.u.S.E. Linux 9.3 and Java 1.5.0.

Results on the NIC's HIV data are shown in Figures 15 and 16. These results were obtained with activated ring mining for rings of size 5 and 6. Results on the IC93 data are shown in Figures 17 and 18 and were obtained without ring mining. The former figure of each pair (i.e., 15 and 17) shows the execution time in seconds, the latter (i.e., 16 and 18) the number of search tree nodes (in thousands) over the minimum support in percent. Solid grey lines refer to the basic algorithm with neither of the two advanced pruning strategies. The dashed grey line refers to the results for equivalent sibling pruning only, the dotted grey line to the results with perfect extension pruning only. Finally, the black line shows the results obtained with both pruning methods activated.

As can be seen from Figures 15 and 16, equivalent sibling pruning yields the highest gains for the HIV data set, while perfect extension pruning yields only fairly small gains. On the IC93 data the picture is reversed: perfect extension pruning is more effective. However, both methods yield considerable gains, which add nicely when both pruning strategies are activated. It is interesting to note that the effects of perfect extension pruning are more pronounced for the number of nodes (see Figure 18), which indicates that part of the gains resulting from this pruning type are eaten up by the somewhat costly test for a perfect extension.

6. CONCLUSIONS

In this paper we presented a Java implementation of an efficient graph substructure mining algorithm that finds closed frequent fragments of molecules. The algorithm is based on a depth first search in a tree of substructures, in which lists of embeddings into the set of molecules are processed and extended. The core ingredients of the approach, which make it efficient, are a structural pruning scheme, which tries to minimize the number of times a fragment is considered in the search, and two advanced pruning techniques, which speed up the search considerably. In addition, the implementation offers other extensions, like ring and chain mining, which make it very useful for biochemical applications.

7. PROGRAM

The implementation of the molecular substructure mining algorithm described in this paper (executable Java archive as well as the source code, distributed under the LGPL) can be downloaded free of charge at

http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html

Explanations of how to apply the program can be found at

http://www.inf.uni-konstanz.de/bioml/projects/mofa/

It takes a simple set of 17 steroids (also used above) as an example, which can also be retrieved from the above URLs.

8. REFERENCES

[1] R. Agrawal, T. Imielienski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proc. Conf. on Management of Data*, 207–216. ACM Press, New York, NY, USA 1993

[2] C. Borgelt and M.R. Berthold. Mining Molecular Fragments: Finding Relevant Substructures of Molecules. *Proc. IEEE Int. Conf. on Data Mining (ICDM 2002, Maebashi, Japan)*, 51–58. IEEE Press, Piscataway, NJ, USA 2002

[3] C. Borgelt. Efficient Implementations of Apriori and Eclat. *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003. http://www.ceur-ws.org/Vol-90/

[4] C. Borgelt, T. Meinl, and M.R. Berthold. Advanced Pruning Strategies to Speed Up Mining Closed Molecular Fragments. *Proc. IEEE Conf. on Systems, Man and Cybernetics (SMC 2004, The Hague, Netherlands)*, CD-ROM. IEEE Press, Piscataway, NJ, USA 2004

[5] D.J. Cook and L.B. Holder. Graph-Based Data Mining. IEEE Trans. on Intelligent Systems 15(2):32–41. IEEE Press, Piscataway, NJ, USA 2000

[6] P.W. Finn, S. Muggleton, D. Page, and A. Srinivasan. Pharmacore Discovery Using the Inductive Logic Programming System PROGOL. *Machine Learning*, 30(2-3):241–270. Kluwer, Amsterdam, Netherlands 1998

[7] H. Hofer, C. Borgelt, and M.R. Berthold. Large Scale Mining of Molecular Fragments with Wildcards. *Proc. 5th Int. Symposium on Intelligent Data Analysis (IDA 2003, Berlin, Germany)*, LNCS 2810:380–389. Springer-Verlag, Heidelberg, Germany 2003

[8] H. Hofer, C. Borgelt, and M.R. Berthold. Large Scale Mining of Molecular Fragments with Wildcards. *Intelligent Data Analysis* 8:495–504. IOS Press, Amsterdam, Netherlands 2004

[9] J. Huan, W. Wang, and J. Prins. Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. *Proc. 3rd IEEE Int. Conf. on Data Mining (ICDM 2003, Melbourne, FL)*, 549–552. IEEE Press, Piscataway, NJ, USA 2003

[10] *HIV Antiviral Screen*. National Cancer Institute. http://dtp.nci.nih.gov/docs/aids/aids_data.html

[11] *Index Chemicus — Subset from 1993*. Institute of Scientific Information, Inc. (ISI). Thomson Scientific, Philadelphia, PA, USA 1993

[12] S. Kramer, L. de Raedt, and C. Helma. Molecular Feature Mining in HIV Data. *Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2001, San Francisco, CA)*, 136–143. ACM Press, New York, NY, USA 2001

[13] M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. *Proc. 1st IEEE Int. Conf. on Data Mining (ICDM 2001, San Jose, CA)*, 313–320. IEEE Press, Piscataway, NJ, USA 2001

[14] T. Meinl, C. Borgelt, and M.R. Berthold. Discriminative Closed Fragment Mining and Perfect Extensions in MoFa. *Proc. 2nd Starting AI Researchers' Symposium (STAIRS 2004, Valencia, Spain)*, 3–14. IOS Press, Amsterdam, Netherlands 2004

[15] T. Meinl, C. Borgelt, and M.R. Berthold. Mining Fragments with Fuzzy Chains in Molecular Databases. *Proc. 2nd Int. Workshop on Mining Graphs, Trees and Sequences (MGTS 2004, Pisa, Italy)*, 49–60. University of Pisa, Pisa, Italy 2004

[16] S. Nijssen and J.N. Kok. A Quickstart in Frequent Structure Mining can Make a Difference. *Proc. 10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD2004, Seattle, WA)*, 647–652. ACM Press, New York, NY, USA 2004

[17] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering Frequent Closed Itemsets for Association Rules. *Proc. 7th Int. Conf. on Database Theory (ICDT'99, Jerusalem, Israel)*, LNCS 1540:398–416. Springer-Verlag, Heidelberg, Germany 1999

[18] T. Washio and H. Motoda. State of the Art of Graph-Based Data Mining. *SIGKDD Explorations Newsletter* 5(1):59–68. ACM Press, New York, NY, USA 2003

[19] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. *Proc. 2nd IEEE Int. Conf. on Data Mining (ICDM 2003, Maebashi, Japan)*, 721–724. IEEE Press, Piscataway, NJ, USA 2002

[20] X. Yan and J. Han. Closegraph: Mining Closed Frequent Graph Patterns. *Proc. 9th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2003, Washington, DC)*, 286–295. ACM Press, New York, NY, USA 2003

[21] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97, Newport Beach, CA)*, 283–296. AAAI Press, Menlo Park, CA, USA 1997

Keeping Things Simple: Finding Frequent Item Sets by Recursive Elimination

Christian Borgelt

Department of Knowledge Processing and Language Engineering
School of Computer Science, Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany

borgelt@iws.cs.uni-magdeburg.de

ABSTRACT

Recursive elimination is an algorithm for finding frequent item sets, which is strongly inspired by the FP-growth algorithm and very similar to the H-mine algorithm. It does its work without prefix trees or any other complicated data structures, processing the transactions directly. Its main strength is not its speed (although it is not slow, even outperforms Apriori and Eclat on some data sets), but the simplicity of its structure. Basically all the work is done in one simple recursive function, which can be written with relatively few lines of code.

1. INTRODUCTION

One of the currently fastest and most popular algorithms for frequent item set mining is the FP-growth algorithm [7]. It is based on a prefix tree representation of the given database of transactions (called an FP-tree), which can save considerable amounts of memory for storing the transactions. A close relative of this approach is the H-mine algorithm [9], which uses a somewhat simpler data structure called an H-struct, which is faster to build than an FP-tree.

The basic idea of both algorithms can be described as a *recursive elimination* scheme: in a preprocessing step delete all items from the transactions that are not frequent individually, i.e., do not appear in a user-specified minimum number of transactions. Then select all transactions that contain the least frequent item (least frequent among those that are frequent), delete this item from them, and recurse to process the obtained reduced database, remembering that the item sets found in the recursion share the item as a prefix. On return, remove the processed item also from the database of all transactions and start over, i.e., process the second frequent item etc. In these processing steps the prefix tree (or the H-struct), which is enhanced by links between the branches, is exploited to quickly find the transactions containing a given item and also to remove this item from the transactions after it has been processed.

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

In this paper I study an algorithm that is based on a very similar scheme, but does its work without a prefix tree or an H-struct representation. It rather processes the transactions directly, organizing them merely into singly linked lists. The main advantage of such an approach is that the needed data structures are very simple and that no re-representation of the transactions is necessary, which saves memory in the recursion. In addition, processing the transactions is almost trivial and can be coded in a single recursive function with relatively few lines of code. Surprisingly enough, the price one has to pay for this simplicity is relatively small: my implementation of this recursive elimination scheme yields competitive execution times compared to my implementations [4, 5] of the Apriori [1, 2] and Eclat [10] algorithms.

2. RECURSIVE ELIMINATION

As already indicated in the introduction, recursive elimination is based on a step by step elimination of items from the transaction database together with a recursive processing of transaction subsets. This section describes the details of this scheme as well as some implementation issues.

2.1 Preprocessing

Similar to several other algorithms for frequent item set mining, like, for example, Apriori or FP-growth, recursive elimination preprocesses the transaction database as follows: in an initial scan the frequencies of the items (support of single element item sets) are determined. All infrequent items—that is, all items that appear in fewer transactions than a user-specified minimum number—are discarded from the transactions, since, obviously, they can never be part of a frequent item set.

In addition, the items in each transaction are sorted, so that they are in *ascending* order w.r.t. their frequency in the database. Although the algorithm does not depend on this specific order, experiments showed that it leads to much shorter execution times than a random order. A *descending* order led to a particularly slow operation in my experiments, performing even worse than a random order.

This preprocessing is demonstrated in Table 1, which shows an example transaction database on the left. The frequencies of the items in this database, sorted ascendingly, are shown in the table in the middle. If we are given a user specified minimal support of 3 transactions, items f and g can be discarded. After doing so and sorting the items in

a d f		a d
c d e		e c d
b d		b d
a b c d		a c b d
b c		c b
a b d		a b d
b d e		e b d
b c e g		e c b
c d f		c d
a b d		a b d

g	1
f	2
e	3
a	4
c	5
b	7
d	8

Table 1: Transaction database (left), item frequencies (middle), and reduced transaction database with items in transactions sorted ascendingly w.r.t. their frequency (right).

each transaction ascendingly w.r.t. their frequencies we obtain the reduced database shown in Table 1 on the right.

2.2 Transaction Representation

Each transaction is represented as a simple arrays of item identifiers (which are integer numbers). The initial transaction database is turned into a set of transaction lists, with one list for each item. These lists are stored in a simple array, each element of which contains a support counter and a pointer to the head of the list. The list elements themselves consist only of a successor pointer and a pointer to (or rather into, see below) the transaction. The transactions are inserted one by one into this structure by simply using their leading item as an index. However, the leading item is removed from the transaction, that is, the pointer in the transaction list element points to the second item. Note that this does not lose any information as the first item is implicitely represented by the list the transaction is in.

To illustrate this, Figure 1 shows, at the very top, the representation of the reduced database shown in Table 1 on the right. The first list, corresponding to the item e, contains the second, seventh and eight transaction, with the item e removed. The counter in the array element states the number of transactions containing the corresponding item. It should be noted, as will become clear later, that this counter is not always equal to the length of the associated list, although this is the case for this initial representation of the database. Differences result from (shrunk) transactions that contain no other items and are thus not represented in the list.

For implementations it is important to note that the described scheme, with a pointer into the transaction so that the leading item is skipped, can only be applied in languages that allow for pointer arithmetic. In languages in which this is impossible (like, for instance, Java) the items in the transactions may be sorted the other way round and an element counter, stored in the list elements, may used to specify the subset of the items that is to be considered.

2.3 Recursive Processing

Recursive elimination works as follows: The array of lists that represents a (reduced) transaction database is "disassembled" by traversing it from left to right, processing the transactions in a list in a recursive call to find all frequent

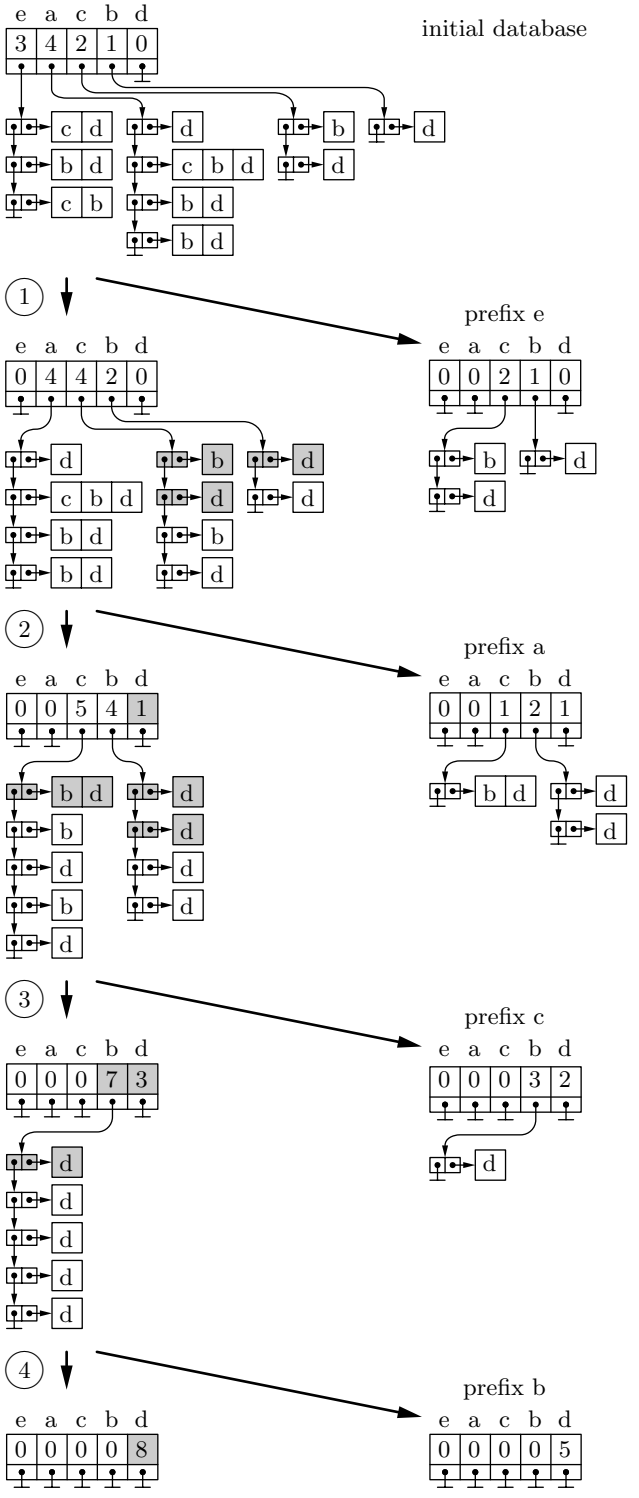


Figure 1: Procedure of the recursive elimination with the modification of the transaction lists (left) as well as the construction of the transaction lists for the recursion (right).

item sets that contain the item the list corresponds to. After a list has been processed recursively, its elements are either reassigned to the remaining lists or discarded (depending on the transactions they represent), and the next list is worked on. Since all reassignments are made to lists that lie to the right of the currently processed one, the list array will finally be empty (will contain only empty lists).

Before a transaction list is processed, however, its support counter is checked, and if it exceeds the user-specified minimum support, a frequent item set is reported, consisting of the item associated with the list and a possible prefix associated with the whole list array (see below).

One transaction list is processed as follows: for each list element the leading item of its (shrunk) transaction is retrieved and used as an index into the list array; then the element is added at the head of the corresponding list. In such a reassignment, the leading item is also removed from the transaction, which can be implemented as a simple pointer increment (or as a counter decrement, see above). In addition, a copy of the list element (with the leading item of the transaction already removed by the pointer increment) is inserted in the same way into an initially empty second array of transaction lists. (Note that only the list element is copied, *not* the transaction. Both list elements, the reassigned one and the copy refer to the same transaction.)

Since the elements of a transaction list all share an item (given by the list index), this second array collects the subset of transactions that contain a specific item and represents them as a set of transaction lists. This set of transaction lists is then processed recursively, noting the item associated with the list it was generated from as a common prefix of all frequent item sets found in the recursion. After the recursion the next transaction list is reassigned, copied, and processed in a recursive call and so on.

The process is illustrated for the root level of the recursion in Figure 1, which shows the transaction list representation of the initial database at the very top. In the first step all item sets containing the item e are found by processing the leftmost list. The elements of this list are reassigned to the lists to the right (grey list elements) and copies are inserted into a second list array (shown on the right). This second list array is then processed recursively, before proceeding to the next list, i.e., the one for item a.

Note that a list element representing a (shrunk) transaction that contains only one item is neither reassigned nor copied, because the transaction would be empty after the removal of the leading item. Instead only the counter in the lists array element is incremented as an indicator of such list elements. Such a situation occurs when the list corresponding to the item a is processed. The first list element refers to a (shrunk) transaction that contains only item d and thus only the counter for item d (grey) is incremented. For the same reason only one of the five elements in the list for item c is reassigned/copied in step 3.

After four steps all transaction lists have been processed and the lists array has become empty. Note that the list for the last element (referring to item d) is always empty, because there are no items left that could be in a transaction and thus all transactions are represented in the counter.

2.4 Optimization Issues

Obviously the allocation of the elements of the transaction lists is a critical issue. This can be done, for example, with a specialized memory management for equally sized small objects, which allocates them in large arrays and then distributes them individually as needed. A related alternative to solve the problem works as follows: in the first place, since by the number of transactions we know the maximum number of list elements we will ever have to create on a recursion level, we can allocate the list elements as an array, in one block of memory. Secondly, we can store the allocated memory blocks in a globally accessible place, so that we only have to allocate them the first time we reach a particular recursion depth and simply reuse them on the second time. This wastes some memory, since one has to allocate on each recursion level as many list elements as there are transactions in the original database in order to be sure that they suffice in all recursion branches. However, since each list element is fairly small (8 bytes on a 32-bit machine) the total amount of needed memory is acceptable. Furthermore, the gains in computation time are substantial and definitely justify this small waste of memory.

Of course, the same scheme of storing allocated blocks of memory in a globally accessible place and resuing them in the recursion is also used for the list arrays. Furthermore, it is clear that copies of the list elements are created only if the support counter corresponding to the currently processed list exceeds the user-specified minimum support, because otherwise no frequent item sets can be found in the recursion and hence building the lists is not necessary.

A final optimization issue concerns the traversal of the lists array. Since in the second lists array only the entries beyond the entry for the item corresponding to the currently processed list can be filled, the traversal can be started at this point in the recursion by passing its index.

2.5 Extensions

An idea that suggests itself when considering how the described recursive elimination scheme may be improved is to use prefix trees instead of simple arrays of item identifiers to represent the transactions. The transaction lists will then be lists of transaction trees and thus can be expected to be shorter. Howver, the disadvantage of such an approach is that processing a list gets much more complex, since reassigning a transaction prefix tree while removing the first item from the represented transactions involves splitting it into its branches, assigning each branch to a different list. In addition, the prefix tree itself is already a much more complicated data structure.

Nevertheless I implemented it, too, in order to check its merits in experiments. Unfortunately the hopes I placed in it were not fulfilled as the experiments reported in the following section show: most of the time using prefix trees actually degrades performance. The cause of this behavior presumably are the more complex operations involved in reassigning prefix trees compared to simple transactions.

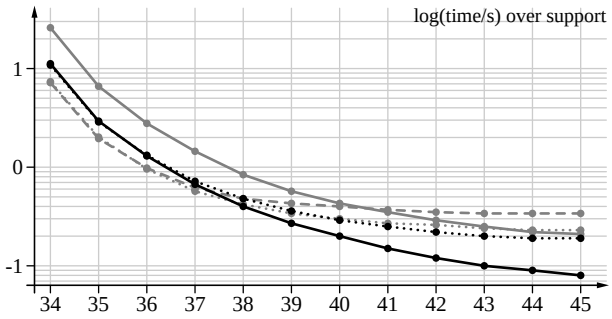


Figure 2: Results on BMS-Webview-1

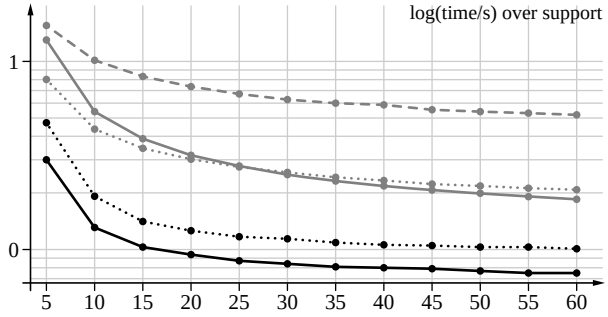


Figure 3: Results on T10I4D100K

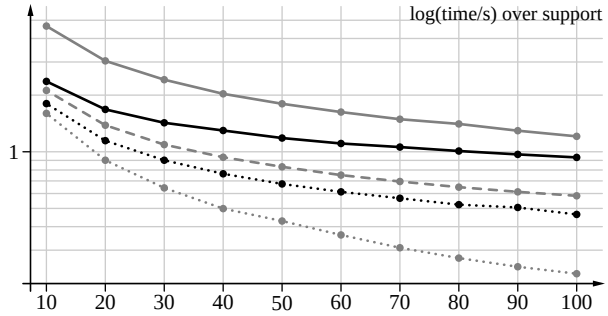


Figure 4: Results on census

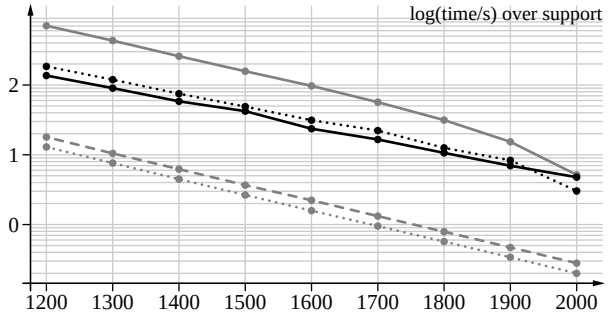


Figure 5: Results on chess

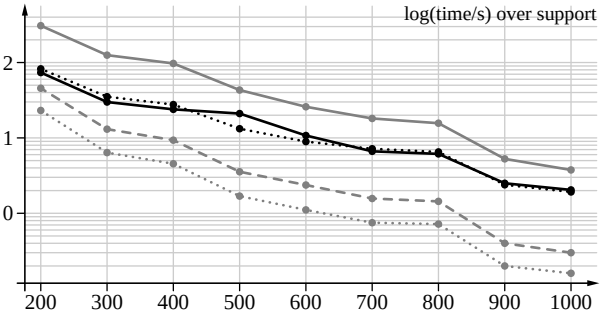


Figure 6: Results on mushroom

3. EXPERIMENTAL RESULTS

I ran experiments on the same five data sets that I already used in [4, 5], namely BMS-Webview-1 [8], T10I4D100K [11], census, chess, and mushroom [3]. However, I used a different machine and an updated operating system, namely a Pentium 4C 2.6GHz system with 1 GB of main memory running S.u.S.E. Linux 9.3 and gcc version 3.3.5). The results were compared to experiments with my implementations of Apriori, Eclat, and FPgrowth. All experiments were rerun to ensure that the results are comparable.

Figures 2 to 6 show, each for one of the five data sets, the decimal logarithm of the execution time over different minimum support values. The solid black line refers to the recursive elimination algorithm studied here, the dotted black line to the version that uses transaction prefix trees. The grey lines represent the corresponding results for Apriori (solid line), Eclat (dashed line), and FPgrowth (dotted line).

In general the recursive elimination algorithm seems to perform in a similar way as Apriori, as can be seen from the fairly similar shapes of the solid black and solid grey lines in all diagrams. However, the recursive elimination algorithm always outperforms Apriori by a considerable margin, on all data sets, particularly pronounced on T10I4D100K. In addition, its superiority usually increases with lower values of the minimum support, an effect that is clearly visible on census, chess, and mushroom.

In a comparison with Eclat recursive elimination also fares pretty well. Although it is bet clearly on chess and, for lower support, on BMS-Webview-1, it almost reaches Eclat's performance on census and mushroom for lower support. FPgrowth, however, fairly clearly outperforms recursive elimination on census, chess, and mushroom.

Although using prefix trees may seem like a good idea at first sight, it almost always degrades performance. The only exception is census, on which prefix trees lead to a substantial gain, even though its relative superiority shrinks with higher support values. Separating the execution times for the tree construction and the recursive elimination shows that it is not the tree construction, but the more complex later processing that is the cause. The gains resulting from the reduced list lengths (which, after some reassignments of a transaction tree, cannot be expected to be so substantial anyway) seem to be too small to outweigh this effect.

4. CONCLUSIONS

Even though its underlying scheme—which is based on deleting items, recursive processing, and reassigning transactions—is very simple and works without complicated data structures, recursive elimination performs surprisingly well, as can be seen from the experiments reported in the preceding section. If a quick and straightforward implementation is desired, it could be the method of choice.

5. PROGRAM

The implementation of the recursive elimination algorithm described in this paper (Windows™ and Linux™ executables as well as the source code, distributed under the LGPL) can be downloaded free of charge at

http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html

At this URL my implementations of Apriori, Eclat, and FP-growth are also available as well as a graphical user interface (written in Java) for finding association rules with Apriori.

6. REFERENCES

[1] R. Agrawal, T. Imielienski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proc. Conf. on Management of Data*, 207–216. ACM Press, New York, NY, USA 1993

[2] A. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast Discovery of Association Rules. In: [6], 307–328

[3] C.L. Blake and C.J. Merz. *UCI Repository of Machine Learning Databases*. Dept. of Information and Computer Science, University of California at Irvine, CA, USA 1998 http://www.ics.uci.edu/~mlearn/MLRepository.html

[4] C. Borgelt. Efficient Implementations of Apriori and Eclat. *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003. http://www.ceur-ws.org/Vol-90/

[5] C. Borgelt. Recursion Pruning for the Apriori Algorithm. *Proc. 2nd IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Brighton, United Kingdom)*. CEUR Workshop Proceedings 126, Aachen, Germany 2004. http://www.ceur-ws.org/Vol-126/

[6] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*. AAAI Press / MIT Press, Cambridge, CA, USA 1996

[7] 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)*. ACM Press, New York, NY, USA 2000

[8] R. Kohavi, C.E. Bradley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 Organizers' Report: Peeling the Onion. SIGKDD Exploration 2(2):86–93. 2000.

[9] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. *IEEE Conf. on Data Mining (ICDM'01, San Jose, CA)*, 441–448. IEEE Press, Piscataway, NJ, USA 2001

[10] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97)*, 283–296. AAAI Press, Menlo Park, CA, USA 1997

[11] Synthetic Data Generation Code for Associations and Sequential Patterns. Intelligent Information Systems, IBM Almaden Research Center http://www.almaden.ibm.com/software/quest/Resources/index.shtml

Subdue: Compression-Based Frequent Pattern Discovery in Graph Data

Nikhil S. Ketkar
University of Texas at Arlington
ketkar@cse.uta.edu

Lawrence B. Holder
University of Texas at Arlington
holder@cse.uta.edu

Diane J. Cook
University of Texas at Arlington
cook@cse.uta.edu

ABSTRACT

A majority of the existing algorithms which mine graph datasets target complete, frequent sub-graph discovery. We describe the graph-based data mining system Subdue which focuses on the discovery of sub-graphs which are not only frequent but also compress the graph dataset, using a heuristic algorithm. The rationale behind the use of a compression-based methodology for frequent pattern discovery is to produce a fewer number of highly interesting patterns than to generate a large number of patterns from which interesting patterns need to be identified. We perform an experimental comparison of Subdue with the graph mining systems gSpan and FSG on the Chemical Toxicity and the Chemical Compounds datasets that are provided with gSpan. We present results on the performance on the Subdue system on the Mutagenesis and the KDD 2003 Citation Graph dataset. An analysis of the results indicates that Subdue can efficiently discover best-compressing frequent patterns which are fewer in number but can be of higher interest.

1. INTRODUCTION

Recently, an increasing body of research has focused on developing algorithms to mine graph datasets. A graph representation provides a natural way to express relationships within data. Graph-based data mining expresses data in the form of graphs, and focuses on the the discovery of interesting sub-graph patterns.

Graph-based data mining has been successfully applied to various application domains including protein analysis[19], chemical compound analysis[1], link analysis[13] and web searching[16]. A number of varied techniques and methodologies have been applied to mining interesting subgraph patterns from graph datasets. These include mathematical graph theory based approaches like FSG[10] and gSpan[20], greedy search based approaches like Subdue [2] or GBI[12], inductive logic programming (ILP) approaches like WARMR[3], inductive database approaches like MolFea[15] and kernel function based approaches[8].

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

Mathematical graph theory based approaches mine a complete set of subgraphs mainly using a support or frequency measure. The initial work in this area was the AGM[6] system which uses the Apriori level-wise approach. FSG takes a similar approach and further optimizes the algorithm for improved running times. gFSG [9] is a variant of FSG which enumerates all geometric subgraphs from the database. gSpan uses DFS codes for canonical labeling and is much more memory and computationally efficient than previous approaches. Instead of mining all subgraphs, CloseGraph[21] only mines closed subgraphs. A graph G is closed in a dataset if there exists no supergraph of G that has the same support as G. Gaston [14] efficiently mines graph datasets by first considering frequent paths which are transformed to trees which are further transformed to graphs. FFSM [5] is a graph mining system which uses an algebric graph framework to address the underlying problem of subgraph isomorphism. In comparison to mathematical graph theory based approaches which are complete, greedy search based approaches use heuristics to evaluate the solution. The two pioneering works in the field are Subdue and GBI. Subdue uses MDL-based compression heuristics, and GBI uses an empirical graph size-based heuristic. The empirical graph size definition depends on the size of the extracted patterns and the size of the compressed graph. Another methodology in this field is that of inductive logic programming which has the advantage of the extensive descriptive power of first-order logic. The first graph-based system to combine the ILP method with Apriori-like level-wise search was WARMR. The major advantage of these approaches is their high representation power. WARMR was used on carcinogenesis prediction of chemical compounds [7].

Another promising direction in the field of graph-based data mining is that of inductive databases which are a new generation of databases that are not only capable of dealing with data but also with patterns or regularities within the data. Data mining in such a framework is an interactive querying process. The inductive database framework is especially interesting for bioinformatics and chemoinformatics, because of the large and complex databases that exist in these domains, and the lack of methods to gain scientific knowledge from them. The pioneer work in this field was the MolFea system, which is based on the level-wise version space algorithm. MolFea is the Molecular Feature miner that mines for linear fragments in chemical compounds. Lastly, the kernel function based approaches have been used to a certain extent for mining graph datasets. The kernel function

defines a similarity between two graphs. When high dimensional data is represented in linear space, the function to learn is difficult in that space. We can map the linear data to nonlinear space and the problem of learning in that high dimensional space becomes learning scalar products. Kernel functions make computation of such scalar products very efficient. Learning in this high dimensional space becomes difficult. The key is finding efficient mapping functions and good feature vectors. The pioneering approach that applied kernel functions to graph structures is the diffusion kernel [8].

In this paper we present the graph-based data mining system Subdue which approaches the problem of mining interesting substructures in graph datasets by finding subgraphs which are not only frequent but also compress the graph dataset. The driving principle of Subdue's approach is the use of the minimum description length (MDL) principle to evaluate the interestingness of a substructure. The use of MDL rather than frequency to evaluate a substructure distinguishes Subdue from all other frequent subgraph discovery systems. Subdue typically produces a fewer number of substructures which best compress the graph dataset. These few substructures which compress the input dataset can provide important insights about the domain than a large number of substructures, each of which is frequent, in the graph dataset. The rest of the paper is organized as follows. In Section 2 we discuss the Subdue system. In Section 3 we discuss the suitability of a compression-based methodology to frequent subgraph discovery. In Section 4 we perform a comparison of Subdue with the graph-based data mining systems FSG and gSpan on the Chemical Toxicity and the Chemical Compounds datasets that are provided with gSpan. In Section 4 we present Subdue's results on Mutagenesis and the KDD 2003 Citation Graph dataset. Conclusions and future work are presented in Section 6.

2. SUBDUE

Subdue is a graph-based relational learning system. The work on Subdue is one of the pioneering works in the field of graph-based data mining. Inputs to the Subdue system can be a single graph or a set of graphs. The graphs can be labeled or unlabeled. Subdue outputs substructures that best compress the input dataset according to the Minimum Description Length (MDL) [17] principle. Subdue performs a computationally-constrained beam search which begins from substructures consisting of all vertices with unique labels. The substructures are extended by one vertex and one edge or one edge in all possible ways, as guided by the example graphs, to generate candidate substructures. Subdue maintains the instances of substructures in the examples and uses graph isomorphism to determine the instances of the candidate substructure in the examples. Substructures are then evaluated according to how well they compresses the Description Length (DL) of the dataset. The DL of the input dataset G using substructure S can be calculated using the following formula,

$$I(S) + I(G|S)$$

where S is the substructure used to compress the dataset G. I(S) and I(G|S) represent the number of bits required to encode S and dataset G after S compresses G. The length of the search beam defines the number of substructures retained for further expansion. This procedure repeats until all substructures are considered or user-imposed computational constraints are exceeded. At the end of the procedure Subdue reports the best compressing substructures. The following is the algorithm for Subdue's discovery process,

```
SUBDUE(Graph, BeamWidth, MaxBest, MaxSubSize, Limit)
 1  ParentList = NULL;
 2  ChildList = NULL;
 3  BestList = NULL;
 4  ProcessedSubs = 0;
 5  Create a substructure from each unique vertex label
    and its single-vertex instances;
 6  Insert the resulting substructures in ParentList;
 7  while ProcessedSubs less than or equal to Limit
    and ParentList not empty
 8        do
 9           while ParentList is not empty
10                 do
11                    Parent = RemoveHead(ParentList);
12                    Extend each instance of Parent
                      in all possible ways;
13                    Group the extended instances
                      into Child substructures;
14           for each Child
15                 do
16                    if SizeOf(Child) less
                      than MaxSubSize
17                       then
18                             Evaluate the Child;
19                             Insert Child in ChildList in
                               order by value;
20                             if BeamWidth less than
                               Length(ChildList)
21                                then
22                                      Destroy
                                        substructure
                                        at end of
                                        ChildList;
23           Increment ProcessedSubs;
24           Insert Parent in BestList
             in order by value;
25           if MaxBest less than Length(BestList)
26              then
27                    Destroy substructure
                      at end of BestList;
28           Switch ParentList and ChildList;
29  return BestList;
```

The Subdue system provides a user-defined option of performing an inexact graph match in the discovery procedure described above. This option is useful in the case where substructure instances can appear in different forms throughout the graph dataset. In this approach, the user can also assign a distortion cost to each graph distortion. A distortion consists of basic transformations such as insertion, deletion and substitution of vertices and edges. By determining the distortion costs, the user can bias the match for or against particular types of distortions.

The Subdue system also provides a user-defined option of using the best substructure found in a discovery step to compress the input graph by replacing those instances of the substructure by a single vertex and performing the discovery process on the compressed graph. The process of finding substructures and compressing the graph can continue until the whole graph is represented by one vertex or the user imposed constraints are exceeded. This feature generates a hierarchical description of the graph dataset at various levels of abstraction in terms of the discovered substructures.

The Subdue knowledge discovery system comes with several auxiliary programs. Some important programs are Inexact graph matcher, Minimum Description Length, distributed MPI Subdue[4] and Subgraph Isomorphism.

3. COMPRESSION-BASED METHODOLOGY

A common characteristic of a majority of the graph-based data mining methodologies described above is that they focus on complete, frequent sub-graph discovery. Complete, frequent sub-graph discovery algorithms such as FSG and gSpan are guaranteed to find all subgraphs that satisfy the user specified constraints. Although completeness is a fundamental and desirable property, one cannot ignore the fact that these systems typically generate a large number of substructures, which by themselves provide relatively less insight about the domain. Typically, interesting substructures have to be identified from the large set of substructures either by domain experts or by other automated methods so as to achieve insights into this domain. Subdue typically produces a smaller number of substructures which best compress the graph dataset. These few substructures which compress the input dataset can provide important insights about the domain.

We demonstrate this advantage of Subdue's compression-based methodology by performing an experimental comparison of Subdue with the graph-based data mining systems FSG and gSpan on a number of artificial datasets. We generate graph datasets with 500, 1000, 1500 and 2000 transactions by embedding one of the six substructures shown in Figure 1. Each of the generated twenty-four datasets (six different embedded substructures, each with 500, 1000, 1500 and 2000 transactions) have the following properties,

1. 60% of the transactions have the embedded substructure, rest of the transactions are generated randomly.
2. For all the transactions that contain the embedded substructure, 60% of the transaction is the embedded substructure, that is, coverage of the embedded substructure is 60%.

Since each of the twenty-four datasets have the properties 1 and 2 discussed above, it is clear that the embedded substructure is the most interesting substructure in each of the datasets. Using these datasets we now compare the performance of Subdue, FSG and gSpan. For each of the twenty-four datasets, Subdue was run with the default parameters and FSG and gSpan were run at a 10% support level. Table

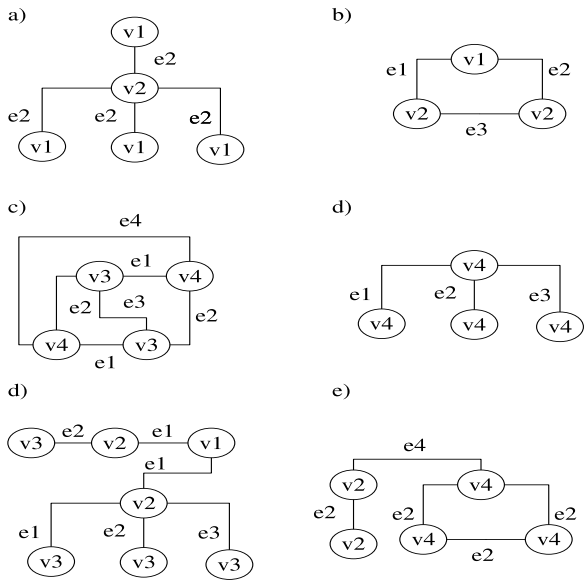


Figure 1: Substructures embedded in the artificial graph datasets. Datasets are generated by embedding one of the six substructures.

Table 1: Results (Average) on 6 artificial datasets each with a different embedded substructure.

Number of transactions	Percent times embedded substructure reported by Subdue as best	Number of substructures generated by FSG/gSpan
500	66%	233495
1000	83%	224174
1500	83%	217477
2000	78%	217479
Average	79%	223156

Table 2: Runtimes (secs.) on 6 artificial datasets each with a different embedded substructure.

Number of transactions	FSG	Subdue	gSpan
500	734	51	61
1000	815	169	107
1500	882	139	182
2000	1112	696	249
Average	885	328	150

1 summarizes the results of the experiments and Table 2 the runtimes of Subdue, FSG and gSpan.

The results indicate that Subdue discovers the embedded substructure and reports it as the best substructure about 80% of the time. Both FSG and gSpan generate approximately 200,000 substructures among which there exists the embedded substructure. The runtime of Subdue is intermediate between FSG and gSpan. Subdue clearly discovers and reports fewer but more interesting substructures. Although it could be argued that setting a higher support value for FSG and gSpan can lead to fewer generated substructures, it should be noted that this can cause FSG and gSpan to miss the interesting pattern. We observed that the increase in run time for Subdue is non-linear when we increase the size of the dataset. Increase for FSG and gSpan was observed to be linear (largely because of various optimizations). The primary reason for this behavior is the less efficient implementation of graph isomorphism in Subdue than in FSG and gSpan. A more efficient approach for graph isomorphism, possibly based on canonical labeling, needs to be developed for Subdue.

4. COMPARISON OF SUBDUE WITH FSG AND GSPAN ON REAL-WORLD DATASETS

We performed an experimental comparison of Subdue with the graph mining systems gSpan and FSG on the chemical toxicity and the chemical compounds datasets that are provided with gSpan. We use these datasets rather than one of the several datasets studied by Subdue researchers in order to perform a fair comparison.

The chemical toxicity dataset has a total of 344 transactions. There are 66 different edge and vertex labels in the dataset. FSG and gSpan results were recorded at 5% support. We found that, if support is set to a lesser value, large numbers of random and insignificant patterns are generated. Table 3 summarizes the results of the experiment. Figure 2 shows the best compressing substructure discovered by Subdue. Subdue discovered best-compressing frequent patterns which were missed by FSG and gSpan. The best substructure by Subdue compressed 8% more than any of the substructures discovered by FSG and gSpan. It is also observed that the runtime of Subdue is much larger than that of FSG and gSpan.

The chemical compounds dataset has a total of 422 transactions. There are 21 different edge and vertex labels in the dataset. The results for FSG and gSpan were recorded at

Table 3: Results on the Chemical Toxicity Dataset.

Compression achieved by best substructure reported by Subdue	16%
Best compression achieved by any of the substructures reported by FSG/gSpan	8%
Number of substructures reported by FSG/gSpan	884
Runtime Subdue (secs.)	115
Runtime FSG (secs.)	8
Runtime gSpan (secs.)	7

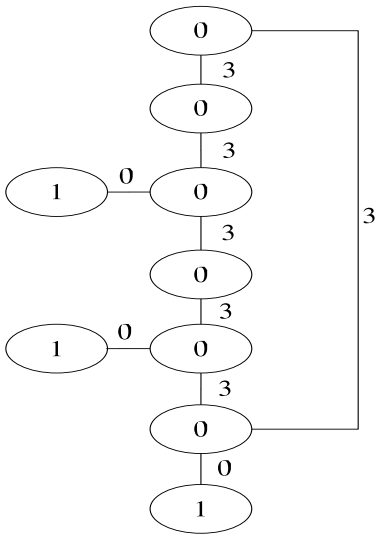


Figure 2: Best Compressing Substructure Discovered by Subdue on the Chemical Toxicity Dataset. The lables on the vertices and edges represent the atom names and bond names. Numbers have been used to represent atom names and bond names as gSpan only accepts numerical lables.

Table 4: Results on the Chemical Compounds Dataset

Compression achieved by best substructure reported by Subdue	19%
Best compression achieved by any of the substructures reported by FSG/gSpan	7%
Number of substructures reported by FSG/gSpan	15966
Runtime Subdue (secs.)	142
Runtime FSG (secs.)	21
Runtime gSpan (secs.)	4

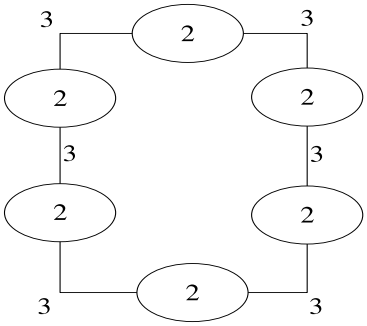


Figure 3: Best Compressing Substructure Discovered by Subdue on the Chemical Compounds Dataset. The lables on the vertices and edges represent the atom names and bond names. Numbers have been used to represent atom names and bond names as gSpan only accepts numerical lables.

10% support. We found that, if support is set to a lesser value, large numbers of random and insignificant patterns are generated. Table 4 summarizes the results of the experiment. Figure 3 shows the best compressing substructures discovered by Subdue. It is observed that Subdue discovered best-compressing frequent patterns which were missed by FSG and gSpan. The best substructure found by Subdue compressed 12% more than any of the substructures discovered by FSG and gSpan. It is also observed that the runtime of Subdue is much larger than that of FSG and gSpan.

5. EXPERIMENTS ON THE MUTAGENESIS AND KDD 2003 CITATION DATASET

The Mutagenesis dataset[18] has been collected to identify mutagenic activity in a compound based on its molecular structure and is considered to be a benchmark dataset for multi-relational data mining. The Mutagenesis dataset consists of the molecular structure of 230 compounds, of which 138 are labeled as mutagenic and 92 as non-mutagenic. The mutagenicity of the compounds has been determined by the Ames Test. The task is to distinguish mutagenic compounds from non-mutagenic ones based on their molecular structure. The Mutagenesis dataset basically consists of atoms, bonds, atom types, bond types and partial charges on atoms. We ran Subdue on the 138 positive examples in the Mutagenesis dataset. Figure 4 shows the best compressing substructure discovered by Subdue which compresses the input data by

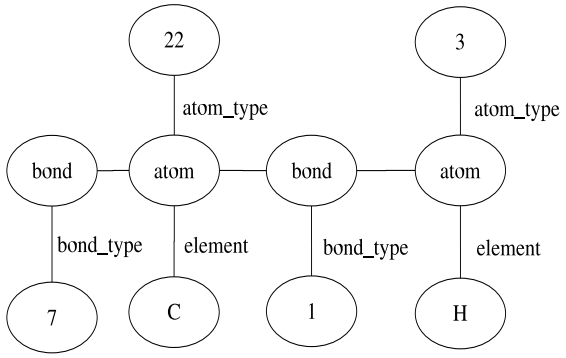


Figure 4: Best Compressing Substructure Discovered by Subdue on the Mutagenesis Dataset

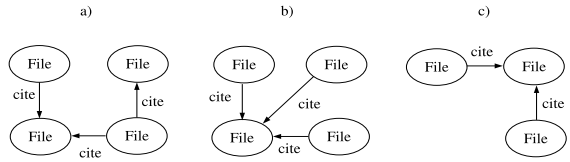


Figure 5: Best Compressing Substructures Discovered by Subdue on the KDD 2003 Citation Dataset

40%. The KDD 2003 Citation dataset consists of citation information from the e-print arXiv, which is the primary mode of research communication in physics. We constructed a citation graph from this data in which each paper was denoted by a single vertex labeled File and a citation between two papers was denoted by a directed edge. The citation graph comprised of a total of 29,014 vertices and 342,437 edges. We ran Subdue on this citation graph. Figure 5 shows the best compressing substructures discovered by Subdue each of which compresses the input data by 9%.

6. CONCLUSIONS AND FUTURE WORK

We presented the graph-based data mining system Subdue which uses a compression-based methodology to frequent subgraph discovery. Subdue can efficiently discover best-compressing frequent patterns which are fewer in number but can be of higher interest. In fact, Subdue may find high-compressing patterns that are less frequent, and therefore missed by the purely frequency-based approach. As a part of our future work we plan to develop better algorithms for graph isomorphism which will improve the run time of Subdue. We also plan to implement canonical labels which would allow implementation of several optimization as in FSG and gSpan. The use of pre-processing schemes (like elliminating vertices and edges with single instances) will also improve the run time of Subdue. Subdue's run time and quality of results is sensitive to the parameters beam and limit. Currently the default values for limit and beam are set based on the total size of the dataset. In certain cases this leads to an unsatisfactory performance as in [11]. Determining optimal values for limit and beam parameters by looking at the graph datasets would be an important direction of work. We also plan to compare Subdue with

recent graph mining systems such as Gaston an FFSM.

The Subdue source code and related materials can be downloaded from http://ailab.uta.edu/subdue.

7. ACKNOWLEDGEMENTS

This research is sponsored by the Air Force Research Laboratory (AFRL) under contract F30602-01-2-0570. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of AFRL or the United States Government.

8. ADDITIONAL AUTHORS

Additional authors: Rohan Shah, University of Texas at Arlington, email: shah@cse.uta.edu and Jeffrey Coble, University of Texas at Arlington, email: coble@cse.uta.edu

9. REFERENCES

[1] R. N. Chittimoori, L. B. Holder, and D. J. Cook. Applying the subdue substructure discovery system to the chemical toxicity domain. In *FLAIRS Conference*, pages 90–94, 1999.

[2] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *J. Artif. Intell. Res. (JAIR)*, 1:231–255, 1994.

[3] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Min. Knowl. Discov.*, 3(1):7–36, 1999.

[4] G. Galal, D. J. Cook, and L. B. Holder. Exploiting parallelism in a structural scientific discovery system to improve scalability. *JASIS*, 50(1):65–73, 1999.

[5] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, pages 549–552, 2003.

[6] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 2000.

[7] R. D. King, A. Srinivasan, and L. Dehaspe. Warmr: a data mining tool for chemical data. *Journal of Computer-Aided Molecular Design*, 15(2):173–181, 2001.

[8] R. I. Kondor and J. D. Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *ICML*, pages 315–322, 2002.

[9] M. Kuramochi and G. Karypis. Discovering frequent geometric subgraphs. In *ICDM*, pages 258–265, 2002.

[10] M. Kuramochi and G. Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Trans. Knowl. Data Eng.*, 16(9):1038–1051, 2004.

[11] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In *SDM*, 2004.

[12] T. Matsuda, T. Horiuchi, H. Motoda, and T. Washio. Extension of graph-based induction for general graph structured data. In *PAKDD*, pages 420–431, 2000.

[13] M. Mukherjee and L. B. Holder. Graph-based data mining for social network analysis. In *Proceedings of the ACM KDD Workshop on Link Analysis and Group Detection*, 2004.

[14] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.

[15] L. D. Raedt and S. Kramer. The levelwise version space algorithm and its application to molecular fragment finding. In *IJCAI*, pages 853–862, 2001.

[16] A. Rakhshan, L. B. Holder, and D. J. Cook. Structural web search engine. In *FLAIRS Conference*, pages 319–324, 2003.

[17] J. Rissanen. *Sochastic Complexity in Statistical Inquiry*. Singapore: World Scientific Publishing, 1989.

[18] A. Srinivasan, S. Muggleton, M. J. E. Sternberg, and R. D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artif. Intell.*, 85(1-2):277–299, 1996.

[19] S. Su, D. J. Cook, and L. B. Holder. Application of knowledge discovery to molecular biology: Identifying structural regularities in proteins. In *Pacific Symposium on Biocomputing*, pages 190–201, 1999.

[20] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[21] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *KDD*, pages 286–295, 2003.

LCM ver.3: Collaboration of Array, Bitmap and Prefix Tree for Frequent Itemset Mining

Takeaki Uno
National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda-ku
Tokyo, JAPAN, 101-8430
uno@nii.jp

Masashi Kiyomi
National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda-ku
Tokyo, JAPAN, 101-8430
masashi@grad.nii.ac.jp

Hiroki Arimura
Information Science and Technology, Hokkaido University
Kita 14-jo Nishi 9-chome
060-0814 Sapporo, JAPAN
arim@ist.hokudai.ac.jp

ABSTRACT

For a transaction database, a frequent itemset is an itemset included in at least a specified number of transactions. To find all the frequent itemsets, the heaviest task is the computation of frequency of each candidate itemset. In the previous studies, there are roughly three data structures and algorithms for the computation: bitmap, prefix tree, and array lists. Each of these has its own advantage and disadvantage with respect to the density of the input database. In this paper, we propose an efficient way to combine these three data structures so that in any case the combination gives the best performance.

1. INTRODUCTION

Frequent item set mining is one of the fundamental problems in data mining and has many applications such as association rule mining, inductive databases, and query expansion. For these applications, fast implementations of frequent itemset mining problems are needed. In this paper we propose a new data structure for decreasing the computational cost, and implemented it to obtain the third versions of LCM, LCMfreq, for enumerating all frequent closed itemsets and all frequent itemsets, respectively. LCM is an abbreviation of *Linear time Closed itemset Miner*.

According to the computational experiments in FIMI03 and FIMI04[6], the heaviest task in the process of frequent itemset mining is the frequency counting, which is to compute the number of transactions including the candidate itemsets. Thus, many techniques and data structures were proposed for frequency counting. Among these, the bitmap [4, 5, 10, 11], the prefix tree [1, 2, 7, 8, 9, 13], and "occurrence deliver" with array lists [14, 16] are popular (see Figure 1).

The bitmap stores the transaction database in a 01 matrix, such that the ij element of the matrix is 1 if and only if item i is included in jth transaction. Each cell can be represented by 1 bit, thus we can save memory, especially in the case that the database is dense. The itemset is also represented by the bitmap, so that the intersection and the union of two itemsets or transactions can be done in short time, since a 32bit CPU operates 32 bits at once. Roughly speaking, the bitmap is efficient if the input database is dense, and the minimum support is not small, i.e., larger than 5% of the number of transactions.

The prefix tree is a popular data structure to store strings or sequences. It is a rooted tree such that any string (or sequence) is represented as a path from a leaf to the root, and any common prefix of two strings is the common subpath of the representative paths of them (see Figure 1). Thus, the common prefixes save memory. The prefix tree is strong if the data is structured, and the minimum support is not too small, hence it is efficient in practice. We can also save computations for frequency counting with respect to the common prefixes. The disadvantage of the prefix tree is the high cost for its reconstruction in the recursive calls.

Occurrence deliver is a technique for efficiently computing the frequencies of many itemsets at once in short time. In an iteration of mining algorithms, some candidate itemsets are generated, and their frequencies are computed. The occurrence deliver stores the transaction database by array lists, and compute the frequencies by scanning the lists once. The occurrence deliver is efficient especially for sparse databases. With a database reduction, it is also efficient for dense databases, but still weak for quite dense databases.

These three techniques have advantages, but also disadvantages. To avoid the disadvantages, we propose a new data structure of a combination of these three techniques. The main part of the data structure is the array list, but for constant number of items, we use a bitmap and a complete prefix tree (see an example in Figure 5). The complete prefix tree is a prefix tree including all the possible itemsets. Mining algorithms are generally recursive, and reduce the database recursively. Thus, the reduced databases usually include a constant number of items in the bottom levels of the recursion. For such small databases, we can use the efficiency of the bitmap and the prefix tree, for frequency counting. Since the computation time in these bottom levels dominates the total computation time, the increase of

OSDM'05, August 21, 2005, Chicago, Illinois, USA.

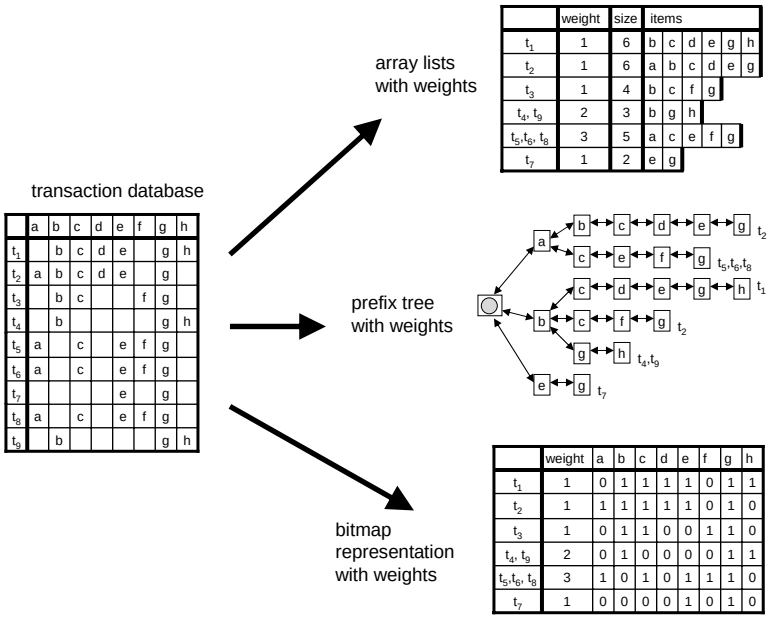


Figure 1: Popular data structures to store transaction databases in memory

the speed in the bottom levels has a drastic effect. The disadvantage of the prefix tree is avoided by using the complete prefix tree and by reusing the complete prefix tree in every iteration.

Moreover, many large databases are in the power law in practice, thus only few items are included in many transactions. Our data structure compresses the part of the databases with respect to such items by the bitmap.

We here list up the advantages of our data structure.

· fast construction
· fast frequency counting for high frequency items
· fast database reduction by using the bitmap
· no need of reconstruction of prefix trees
· applicable to many existing algorithms

However, when the database is very sparse and the minimum support is very small, i.e., less than 10, the computation time is sometime slow. In these cases, the computation time is 3 or 4 times longer than the fastest implementation in FIMI repository. This is because of the cost for changing the strategy. Note that in these cases the fastest implementation is the previous versions of LCM.

Our data structure can be applied to backtrack algorithms (depth-first algorithm) and also apriori type algorithms. It can also be applied to mining algorithms for any problem; frequent itemset mining, maximal frequent itemsets mining, and frequent closed itemset mining. We implemented codes for both frequent itemset mining and frequent closed itemset mining. The computational experiments shows that in almost cases our implementation is the fastest in the implementations proposed in FIMI03 and FIMI04. A part of the results are shown in section 4.

2. PRELIMINARIES

Let $\mathcal{E} = \{1, ..., n\}$ be the set of *items*. A *transaction database* on $\mathcal{E}$ is a set $\mathcal{T} = \{T_1, \ldots, T_m\}$ such that each T_i is included in $\mathcal{E}$. Each T_i is called a *transaction*. We denote by $||\mathcal{T}||$ the sum of sizes of all transactions in $\mathcal{T}$, that is, the size of database $\mathcal{T}$. A set $P \subseteq \mathcal{E}$ is called an *itemset*. The maximum element of P is called the *tail* of P, and denoted by $tail(P)$. An itemset Q is a *tail extension* of P if and only if both $Q \setminus P = \{e\}$ and $e > tail(P)$ hold for an item e. An itemset $P \neq \emptyset$ is a tail extension of Q if and only if $Q = P \setminus tail(P)$, hence Q is unique, i.e., any non-empty itemset is a tail extension of a unique itemset.

For itemset P, a transaction including P is called an *occurrence* of P. The *denotation* of P, denoted by $Occ(P)$ is the set of the occurrences of P. $|Occ(P)|$ is called the *frequency* of P, and denoted by $frq(P)$. In particular, for an item e, $frq(\{e\})$ is called the frequency of e. For given constant θ, called a *minimum support*, itemset P is *frequent* if $frq(P) \geq \theta$. If a frequent itemset P is included in no other frequent itemset, P is called *maximal*. We define the *closure* of itemset P in $\mathcal{T}$, denoted by $clo(P)$, by $\bigcap_{T \in Occ(P)} T$. An itemset is called *closed itemset* if $P = clo(P)$. For any two itemsets P and Q, the following properties hold.

(1) $Occ(P \cup Q) = Occ(P) \cap Occ(Q)$
(2) If Q is a tail extension of P,
$Occ(Q) = Occ(P \cup \{e\})$ for an item e
(3) If $P \subseteq Q$, $frq(Q) \leq frq(P)$
(4) If Q is a tail extension of P, $frq(Q) \leq frq(P)$.

Through this paper, let c be a given constant, which we can choose by looking the property of the input database. For transaction T, we call the set of items in T greater than $n-c$ *constant suffix* of T, and denote it by T^s. Similarly, we call $T \setminus T^p$ *constant prefix* of T, and denote it by T^p. We recall that n is the largest item.

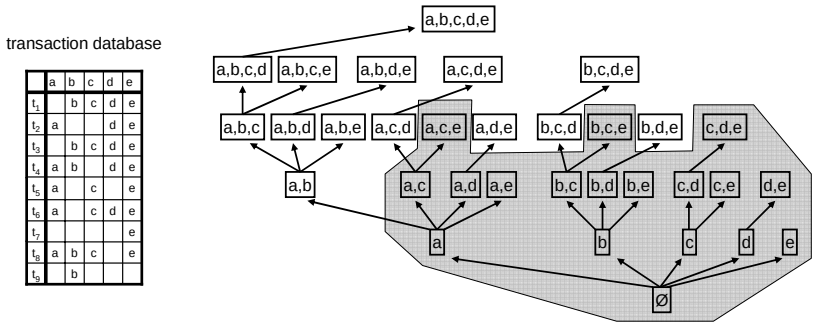


Figure 2: A tree induced by tail extensions: backtrack algorithm traverses the tree in a depth-first way

3. EXISTING ALGORITHMS AND DATA STRUCTURES

In the following subsections, we explain the popular techniques used in frequent itemset mining.

3.1 Backtrack Algorithm

Any itemset included in a frequent itemset is itself frequent. Thereby, the property "frequent" is anti-monotone. In particular, any frequent itemset is a tail extension of a frequent itemset. From this, we can see that any frequent itemset can be constructed from the empty set by generating tail extensions recursively. A backtrack algorithm is a depth-first version of this generation[3, 8, 13, 14, 15, 16].

Backtrack algorithm is based on recursive calls. An iteration of a backtrack algorithm inputs a frequent itemset P, and generates all tail extensions of P. Then, for each extension being frequent, the iteration generates a recursive call with respect to it. We describe the framework of backtrack algorithms below. Figure 2 shows an example of the execution of a backtrack algorithm.

ALGORITHM Backtrack (P:itemset)
1. **Output** P
2. **For each** $e \in \mathcal{E}, e > tail(P)$ **do**
3. **If** $P \cup \{e\}$ is frequent **then call** Backtrack ($P \cup \{e\}$)

We note that here an *iteration* of the algorithm is the computation from step 1 to 3 in a recursive call, except for the computation done in the recursive calls generated in step 3. The advantage of the algorithm is that it needs no memory for storing the frequent itemsets previously obtained, even if the number of the frequent itemsets is huge. Since backtrack algorithms generate a number of recursive calls in an iteration, the number of iterations at the kth level of the recursion is small if k is closed to 1, and increases exponentially as the increase of k. We call this property *bottom-wideness*. From the bottom-wideness, we can see that the total computation time of a backtrack algorithm is dominated by the computation in the bottom levels of the recursion.

3.2 Frequency Counting

As we can see, the heaviest part of the computation in a backtrack algorithm is in step 3, the computation of $frq(P \cup \{e\})$ to check whether $P \cup \{e\}$ is frequent or not. We call this computation *frequency counting*. Frequency counting can be done by taking the intersection of $\mathcal{T}(P)$ and $\mathcal{T}(\{e\})$, however it takes long time in a straightforward way. To reduce the computation time, there are many approaches; bitmap, prefix tree, occurrence deliver, and conditional database. The conditional database is orthogonal to the others, hence we can combine it to one of the others.

The approach of the bitmap is to use a bitmap for representing $\mathcal{T}(P)$ and $\mathcal{T}(\{e\})$. Then, we can take intersection by binary operation "and", thus 32 or 64 operations can be done in one step. However, if the database is sparse and the minimum support is small, then the bitmap representation includes so many 0's in it, which can be omitted in the other ways.

In the following, we explain the conditional database, the prefix tree, and the occurrence deliver.

3.3 Conditional Database

A *conditional database* of an itemset P is a database used in an iteration inputting P, and is obtained by removing items and transactions which are not necessary in the iteration and its recursive calls. Using conditional databases we can reduce the tail extensions to be checked, and the computation time for frequency counting.

For an itemset P, its conditional database, denoted by $\mathcal{T}_P$ is obtained from $\mathcal{T}$ by the following way (see Figure 3):

1. remove transactions not including P
   (database becomes equal to $Occ(P)$)
2. remove items no larger than $tail(P)$
3. remove items included in less than θ transactions of $Occ(P)$
4. remove items e included in all transactions of $Occ(P)$,
   and record that "e is included in all transactions"
5. after removing items as 2, 3, and 4, remove duplicated transactions, i.e., if there are k same transactions, remove $k-1$ of them, and set the weight of T to k for keeping that there were $k-1$ more transactions equal to it.

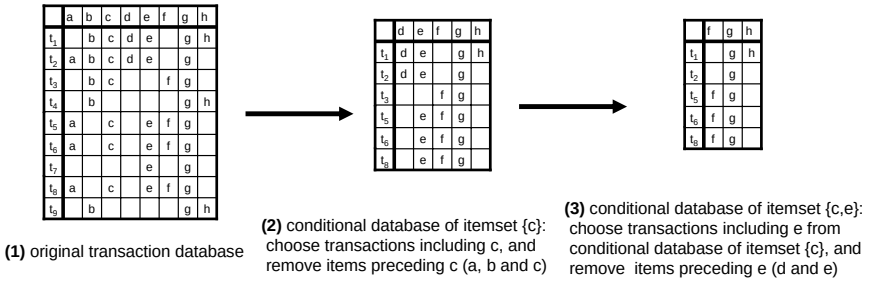


Figure 3: Conditional Database: example of construction

For any tail extension $P \cup \{e\}$, $Occ(P \cup \{e\})$ is obtained by choosing all transactions in $\mathcal{T}_P$ including e. The frequency of $P \cup \{e\}$ can be computed in a similar way. Thus we can do all operations in the recursive call with respect to P only with the conditional database of P. As the increase of the size of the itemset P, the size of its conditional database decreases exponentially. Thus the computation time for frequency counting is reduced in the bottom level. From the bottom-wideness, the total computation time for mining can also be drastically shortened. If a transaction has no weight, we consider the weight of the transaction is 1. Thus, through this paper, we assume that any transaction has a weight.

Actually, the term "conditional database" is used for many kinds of restricted databases in other papers, thus its definition is different from those in the other papers.

3.4 Prefix Tree

A *Prefix tree* is a tree shaped data structure for storing strings. Itemsets and transactions can be considered as strings composed of items, thus we can use prefix trees to store them. A vertex of the tree has an alphabet, thus a path of the tree gives a string. Each string is represented as the path from a representative vertex to the root. A vertex has no two children having the same alphabet, thus the position of the representative vertex of any string is unique. For any two strings having a common prefix, the paths representing them share the path corresponding to the prefix. See an example in Figure 1. The prefix tree is sometime called *frequent pattern tree*, or FP-tree in short, since it is also used in some algorithms to store the obtained frequent itemsets. Note that we sort items in each transaction to be stored in the order of their frequencies, so that many transactions share prefixes with others.

To compute the occurrences and the frequency of a tail extension $P \cup \{e\}$, we look up all vertices having e on them, and go down (opposite direction to the root) the tree from the vertices to find all the vertices assigned a transaction including P. The prefix tree can be also used for storing the conditional database. In the case, any occurrence of $P \cup \{e\}$ is assigned to a descendant of a vertex having e, and vice versa. Thus, the computation can be efficiently done, and computation time becomes short. This technique is popular, and many recent implementations use this technique[1, 2, 7, 8, 9, 13].

3.5 Occurrence Deliver

For an itemset P, the occurrence deliver computes the frequency and occurrences of all tail extensions of P at once. The occurrence deliver stores the transaction database by array lists. An array list is a list represented by an array such that the elements of the list is stored in the array. In an array list representation, we store each transaction of the input database in an array list so that the items in the array is sorted. See an example of array lists in Figure 1. For the initialization, the occurrence deliver assign an empty bucket for each item. Then, the occurrence deliver scans each transaction T_i in $Occ(P)$, and insert T_i to the bucket of each item included in T_i. After scanning all transactions in $Occ(P)$, the bucket of item e is equal to $Occ(P \cup \{e\})$.

The occurrence deliver works in the conditional database. The process of the occurrence deliver is equivalent to the transpose of the matrix of the conditional database, represented by array lists. We write the pseudo code of the conditional database version of the occurrence deliver below. We can also see an example of its execution in Figure 4.

ALGORITHM OccurrenceDeliver (P:itemset, $\mathcal{T}_P$:conditional database)
1. Set $Bucket[e] := \emptyset$ for each item e in $\mathcal{T}_P$
2. **For each** transaction $T_i \in \mathcal{T}_P$ **do**
3. **For each** item $e \in T_i$ **do**
4. Insert T_i to $Bucket[e]$
5. **End for**
6. **End for**
7. **Output** $Bucket[e]$ for all items e

LEMMA 1. *We can get the conditional database $\mathcal{T}_{P\cup\{e\}}$ by merging the transactions of $Bucket[e]$ including exactly the same items into one, and put the weights.*

Since array list is a compact form for sparse databases, the occurrence deliver is efficient for sparse databases[14, 15, 16]. With the use of conditional databases, it is also efficient for some dense databases, however, it takes much cost for quite dense databases[16].

4. NEW DATA STRUCTURE

Although the existing algorithms and data structures are efficient, they have their own disadvantage coming from the density and the structure of databases. The motivation of our new data structure is that we would have a good data

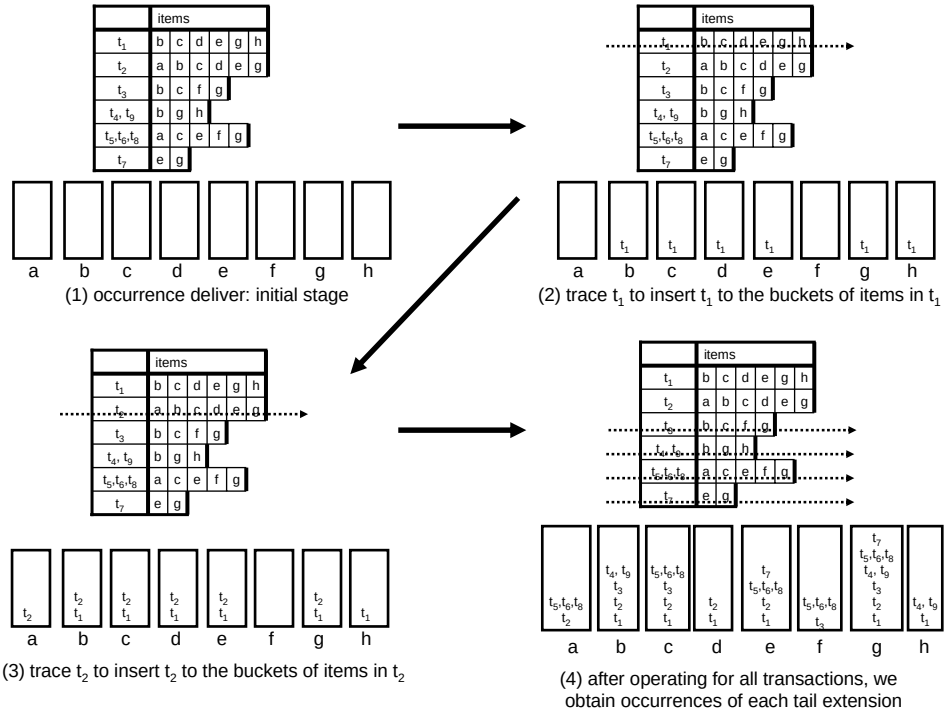


Figure 4: Occurrence deliver: example of its process

structure and a fast frequency counting algorithm by carefully combining them.

Our new data structure is very simple. When we input a transaction database, we choose a constant c. We can choose c so that the memory usage is minimal. Then, we use one integer $bit(T)$ and an integer array $ary(T)$ of size $|T^p|$. The ith bit of $bit(T)$ is 1 if and only if T^s includes item $n-i$, thus $bit(T)$ is a bitmap representation of T^s. Each item of the constant prefix T^p of T is stored in $ary(T)$ in the increasing order of items, thus $ary(T)$ is an array list representation of T^p.

For efficient frequency counting, we have a complete prefix tree for items greater than c. The complete prefix tree is composed of vertices i for $0 \leq i < 2^{n-c}$, so that vertex i corresponds to the transactions T such that $bit(T) = i$. For each vertex i, we have two integers $w(i)$ and $h(i)$. $w(i)$ is the *weight* of i, and used to keep the number (the sum of the weights) of transactions T such that $bit(T) = i$. $h(i)$ is the highest bit in i set to 1, that is, the smallest item in the constant suffix represented by i. The parent vertex of vertex i is the vertex $i - 2^{h(i)}$, where $i - 2^{h(i)}$ is the number obtained by setting $h(i)$th bit of i to zero. Similarly, any child vertex j of vertex i is obtained by setting hth bit of i to 1 for $h > h(i)$. In contrast to the usual prefix trees, we need neither pointers indicating the parent vertex and the child vertices.

Further, we have a bucket for each item, which is used by the occurrence deliver. The size of the bucket of item i is 2^{n-i} if $i > c$, and $|Occ(\{i\})|$ otherwise. We reuse the complete prefix trees and the buckets in every iteration, thus we have to allocate memory for them only once at when we input the database.

For the memory and time efficiency, we renumber the items of the input database so that the items of higher frequencies have larger indices. We can see an example of how to construct our data structure from an array list represented transaction database in Figure 5. Note that the items are sorted in the order of their frequencies so that items with high frequencies are left (it corresponds to renumbering). We can also see the complete prefix tree for the itemset $\{a, b, c, d\}$ in Figure 6.

The size of the complete prefix tree increases exponentially as the increase of c, and the memory space needed by array lists and the bitmap can decrease by the increase of c. Thus, we choose c so that the memory use is optimal, and larger than a specified minimum threshold such as 12, for speeding up. The optimal c can be found in $O(||\mathcal{T}||)$ time since the memory use can be computed from the frequencies of items. If $c = 12$, the complete prefix tree has 4096 vertices. It is not so large. However, if $c = 32$, the number of vertices grows up to 4,000 million.

We have some motivations and ideas on the new data structure. We briefly explain them below.

1. Many datasets in the real world are in the power law, thus the bitmap is efficient for and only for a small number of items.
Roughly speaking, the bitmap is efficient if items are included in at least 1% of transactions. Under the power law, databases include few, constant number of, such items. The part with respect to such items are dense part of the database, and the bitmap can compress it. For the sparse

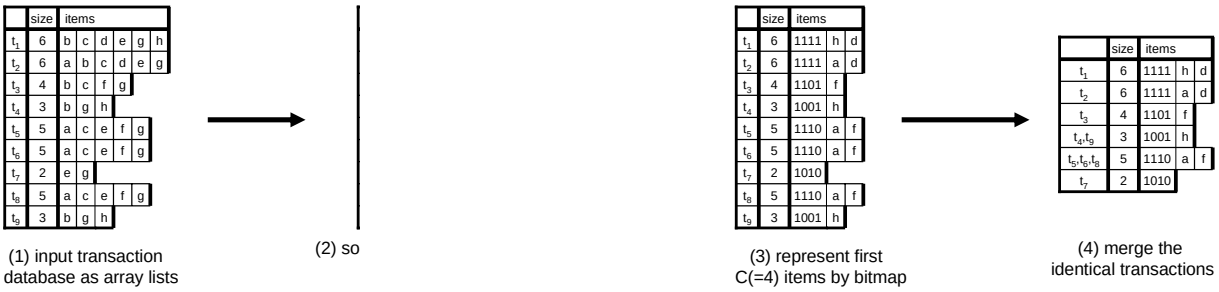


Figure 5: Example of the construction of our database

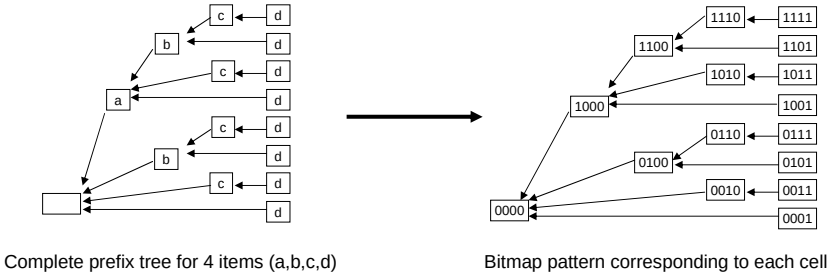


Figure 6: The complete prefix tree: each vertex of the tree on the right-side has the bitmap pattern corresponding to the string which the vertex representing.

(non-dense) part of databases, array lists are efficient. The prefix tree does not compress sparse databases as well as dense databases, but takes much cost. Hence, we do not use it for storing the transactions.

2. For the constant number of items, the size of the complete prefix tree is also constant.
Under the power law, the database has few items, usually bounded by a constant, with high frequencies. The use of the prefix tree for such items is efficient for frequency counting. Although the number of the items in the complete prefix tree is small, the computation time is drastically reduced because of bottom-wideness.

3. The complete prefix tree can be re-used.
A disadvantage of the prefix tree is that they need pointer operations and reconstructions. The complete prefix tree includes a vertex corresponding to any pattern composed of items larger than c, thus it needs no reconstruction, but only initialization of accessed vertices in the previous iterations. Thus, by using the complete prefix tree and reused it again and again in every iterations, we can avoid this disadvantage. It shortens the computation time of the iterations in the bottom level of the recursion, especially if the input database is dense.

4. In many iterations databases are small and dense
By sorting and re-numbering the items in the increasing order of their frequency, the computation time for frequency counting decreases, since the sizes of the conditional databases become smaller, more dense, and more structured on average. According to bottom-wideness, in many iterations, the conditional database includes at most c items. Hence, we can use the bitmap and the complete prefix tree, so that the computation time is reduced especially in the case that the input database is dense.

5. Constructing conditional databases becomes easy
The heaviest task in the constructing conditional databases for array lists is to find the duplicated transactions. Actually, it can be done by applying radix sort to the transactions. By using the bitmap, we can omit the computation of the radix sort for items with high frequencies. Thus, the computation time is reduced.

4.1 Frequency Counting with Complete Prefix Tree

By using the complete prefix tree, we can get both the frequency and the conditional databases for all extensions of the current itemset P by a sweep on the complete prefix tree.

Suppose that we have an itemset P and want to compute the frequencies and the conditional databases for all extensions of P. First, we initialize the complete prefix tree and the buckets, so that the weight of any vertex is zero and any bucket is empty. It can be done by initializing the vertices v with $h(v) > tail(P)$ of non-zero weights, and non-empty buckets of items $e > tail(P)$, hence we do not need initialize all the vertices and buckets.

After the initialization, for each transaction T in the conditional database $\mathcal{T}_P$, we add the weight of T to vertex $bit(T)$. Then, set the weight of vertex to the sum of weights of its

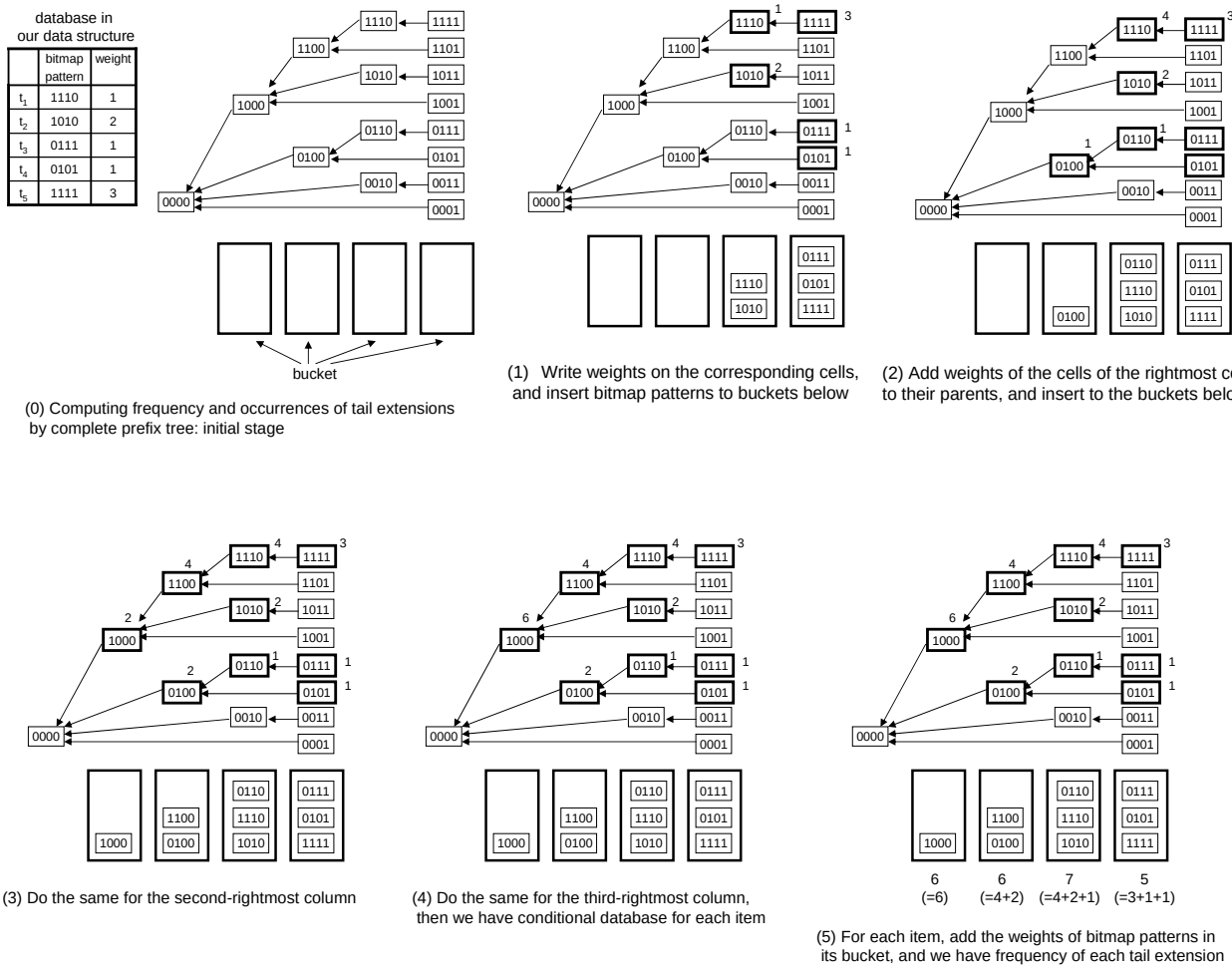


Figure 7: Example of complete prefix tree: re-use and frequency counting

descendants, and insert each vertex i of non-zero weight to the $(n-h(i))$th bucket. The former operation can be done in linear time of $|\mathcal{T}_P|$. In a bottom up way, or sweep the buckets in the increasing order, the latter computation can be done in time linear to the number of vertices with non-zero weights.

LEMMA 2. *After this operation, for any $i > c$, the ith bucket is the bitmap representation of the conditional database $\mathcal{T}_{P\cup\{i\}}$, i.e., each vertex and its weight in ith bucket are the bitmap representation of a transaction in $\mathcal{T}_{P\cup\{i\}}$ and its weight.*

From the lemma, we can see that the sum of the weights of the vertices in the ith bucket is the frequency of $P\cup\{i\}$ for any $i > c, i \notin P$. See an example in Figure 7.

4.2 Reuse of Complete Prefix Tree

To use prefix trees and the conditional database for frequency counting, we have to construct a prefix tree for the conditional database. We avoid this construction by reusing the complete prefix tree in every iteration, with the use of the rightmost sweep proposed in the first version of LCM[14].

Suppose that we have an itemset P, and completed the sweep described in the previous subsection. Then, we make recursive calls with respect to each tail extension $P\cup\{e\}$, in the decreasing order of items. In the recursive call with respect to $P\cup\{e\}$, the buckets of items no greater than e never be accessed. The vertices i with $n-h(i) \leq e$ never be also accessed. This shows that during the recursive call, the buckets and weights of vertices in the buckets are preserved for any item smaller than e. Therefore, we can reuse the complete prefix trees and buckets without disturbing the later recursive calls. This saves the memory space and computation time to allocate new memory space.

4.3 Prefix Maintenance for Closure and Maximality Checking

To apply our data structure to closed itemset mining algorithms and maximal frequent itemset mining algorithms, we have to modify the data structure so that we can compute the closure and checking the maximality in short time. In LCM ver.2[16], we propose a technique for efficiently

maintaining the prefixes of the transactions in conditional databases. Here the prefix of a transaction T in a conditional database $\mathcal{T}_P$ is the set of items in T no larger than $tail(P)$.

Suppose that transactions $T_1, \cdots, T_k$ are merged into one transaction T in $\mathcal{T}_P$. For taking closure, we put to T the intersection of the prefixes of $T_1, \cdots, T_k$.

For the check of the maximality, we take union of $T_1, ..., T_k$, and put it to T. We further put a weight to each item i of the prefix, where the weight is the number of transactions in $T_1, ..., T_k$ containing i. If the transactions are already merged in the operations previously done, the weight of i is given by the sum of the weights of i over $T_1, ..., T_k$.

These operations can be easily treated by our new data structure, just by doing them when we compute the frequencies and conditional databases of tail extensions in the way described in the previous subsection. Thus, our new data structures are efficiently applicable to closed itemset mining and maximal frequent itemset mining.

5. COMPUTATIONAL EXPERIMENTS

In this section, we show the results of our computational experiments. We implemented our new data structure and apply it for the second version of LCM and LCMfreq, which are for frequent itemset mining and frequent closed itemset mining. They are coded by ANSI C, and complied by gcc, and the machine was a PC with Pentium4 3.20E GHz CPU and 2GB memory. Due to the time limitation, we have no implementation for LCMmax for maximal frequent itemset mining. The performance of LCM algorithms are compared with the algorithms which marked good score on FIMI 03 or FIMI 04: fpgrowth[7], afopt[9], aim2[1], MAFIA[4, 5], kDCI and DCI-closed[10, 11], nonodrfp[2], and PATRICIA-AMINE[13]. We note that aim2, nonodrfp and PATRICIA-AMINE are only for all frequent itemset mining.

From the performances of implementations, the instances were classified into four groups, in each of which the results are similar. Due to the space limitation, we show one instance as a representative for each group.

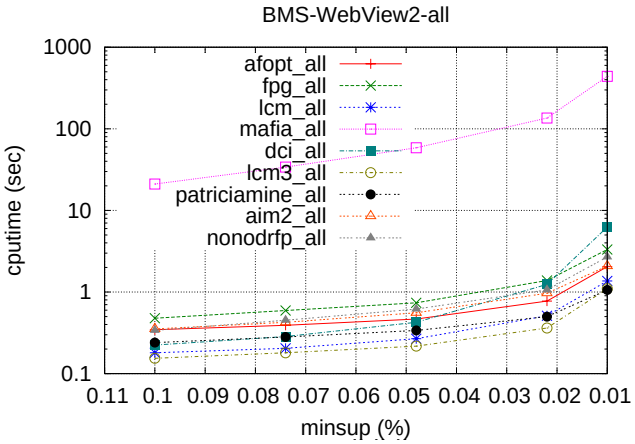
The first group is composed of BMS-WebView1, BMS-WebView2, T10I4D100K, and retail. These datasets have many items and transactions but are sparse, and even if the minimum support is very small, such as 10, the number of frequent itemsets is not so huge, i.e., frequent itemsets are enumeratable. We call these datasets *very sparse datasets*. We chose BMS-WebView2 as the representative.

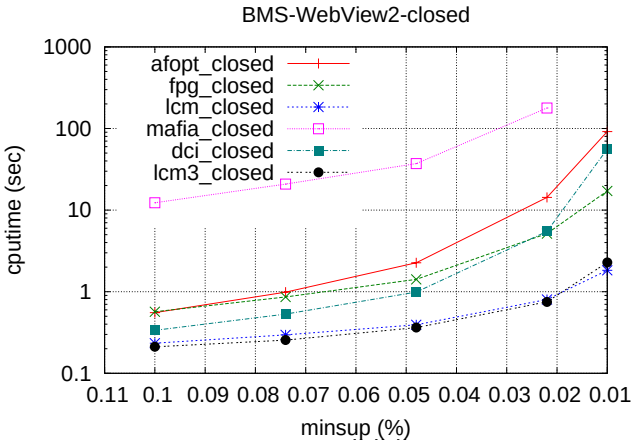
The second group is of mushrooms, BMS-POS, kosarak, Webdoc, and T40I10D100K. They are also sparse, but the number of frequent itemsets is quite huge when the minimum support is very small. We call them *sparse datasets*. We chose kosarak as the representative.

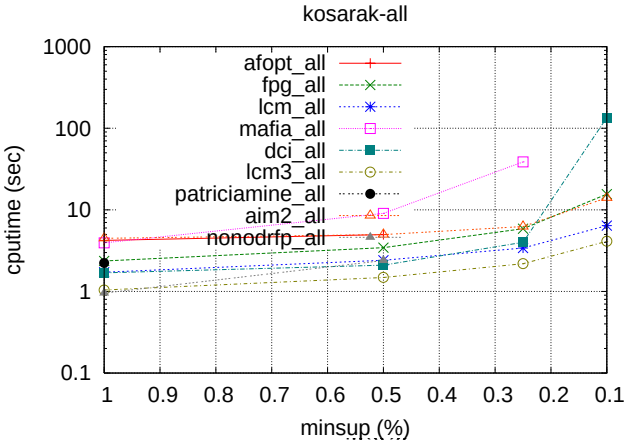
The third group is composed of connect, chess, pumsb, and pumsb-star. These datasets are generated in mathematical ways but are not natural data, thus they are structured. They have many transactions but few items. Transactions have many items, so the dataset is dense. Moreover, the number of frequent closed itemsets is very much smaller than the number of frequent itemsets for smaller minimum supports. We call these datasets *structured dense datasets*. As a representative, we show the result of chess.

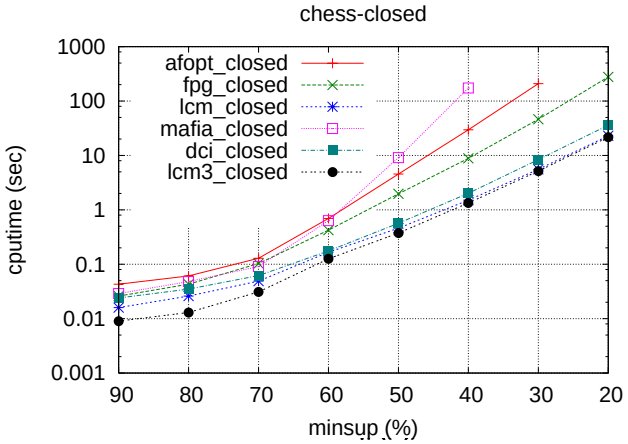
The fourth group is composed only of accidents, since its density is different from any other dataset. It has huge number of transactions, but few items. Each transaction includes many items, so the dataset is very dense, and is not structured so that the number of frequent itemsets and the number of frequent closed itemsets are almost equal for any enumeratable minimum support. We call this dataset *dense dataset*.

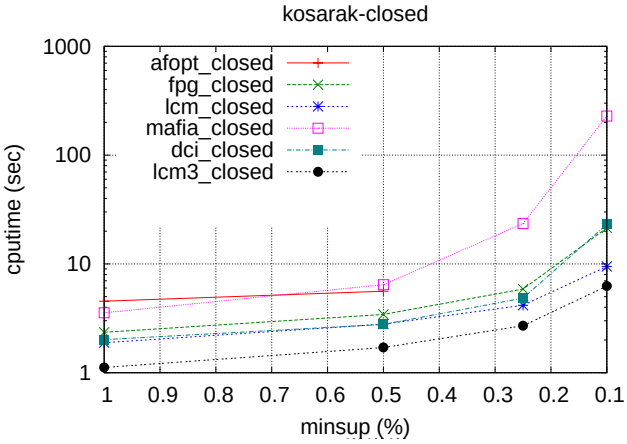
In the following, we show the results for representatives. The new implementation in this paper is written as "LCM3". To reduce the time for experiments, we stop the execution when an implementation takes more than 10 minutes. We do not plot if the computation time is over 10 minutes, or abnormal termination.

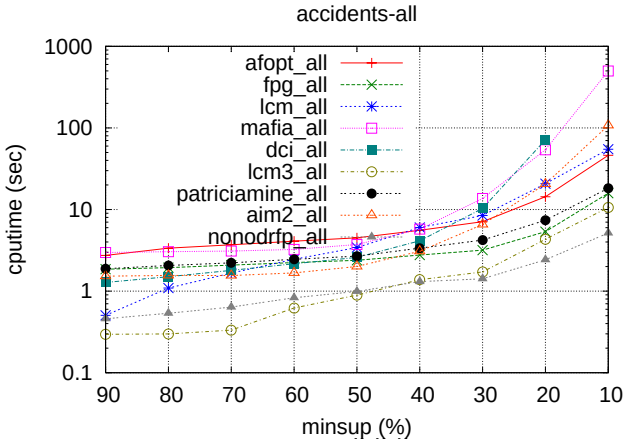


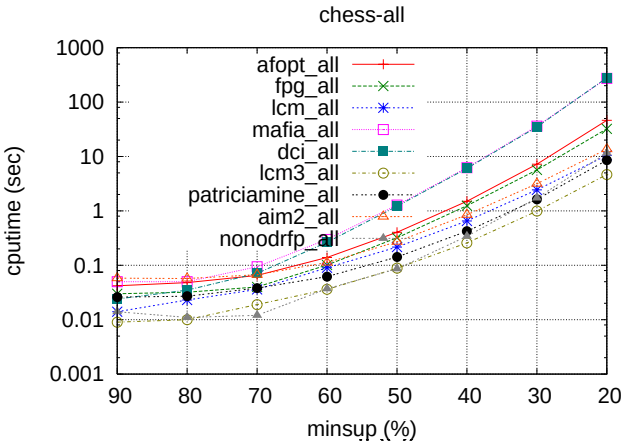


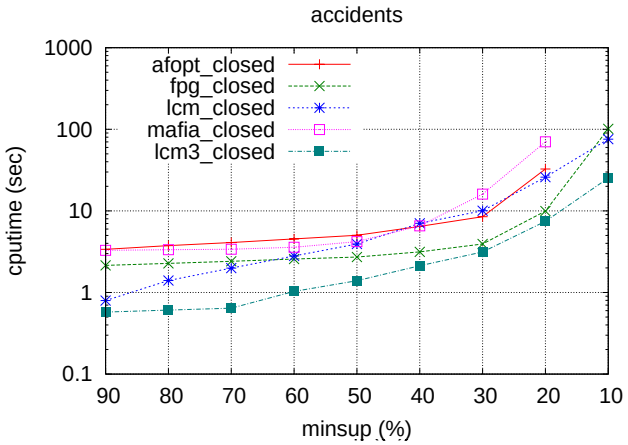












In almost instances and minimum supports, LCM3 performs the best. For unstructured dense datasets, and dense datasets with large minimum supports, nonodrfp sometime outperforms LCM3. For quite small minimum supports, LCM3 algorithms are fast but not the fastest. This is because of the cost for changing the strategy, which is from array lists to the prefix tree. If the cost per a frequent itemset is large, i.e., iterations generates few recursive calls on average, then LCM3 is slow. Moreover, in some sparse datasets, only few transactions of conditional databases share prefixes on average. In such cases, frequency counting with a prefix tree

takes longer time than that with array lists, hence the first version and the second version of LCM are fast.

For very sparse datasets, the memory usage is almost the same as array list, which is used in LCM ver.2. However, for the other datasets, the memory usage decreases to half in average.

6. CONCLUSION

In this paper, we proposed a new data structure for frequent itemset mining algorithms. We applied our data structure to the second version of LCM algorithms, and gave implementations of them. We show by computational experiments that our implementations perform above the other implementations for almost almost datasets in any minimum support, except for very small minimum support. For these very small minimum support, the second version of LCM is the fastest, thus we conclude that we can get a fast implementation by a combination of the second version and the new version of LCM algorithms.

For very huge datasets such that 10 or 100 times larger than the fimi datasets, optimizing the memory usage is quite important. In this sense, our data structure is not the best: perhaps the combination of array list and prefix trees (patricia trees[13]) might be the best. In particular, if the database is quite structured so that many transactions share prefixes, but includes no item of a high frequency. For maximal frequent itemsets mining and frequent closed itemset mining, if the number of output itemsets is small, then the checking the maximality and taking closure take long time rather than the way with storing all the itemsets obtained. Our data structure is not weak in these cases, but the next goal is how to perform better than others for such cases.

Acknowledgment

We sincerely thank to the organizers of FIMI03 and FIMI04, and all the authors gave implementations to FIMI, especially for Prof. Bart Goethals for his working for constructing and maintaining FIMI repository. This research is supported by Grant-in-Aid for Scientific Research of Japan and joint-research funds of National Institute of Informatics.

7. REFERENCES

[1] A. Fiat and S. Shporer, "AIM2: Improved implementation of AIM" In *Proc. IEEE ICDM'04 Workshop FIMI'04*, 2004.

[2] B. Racz, "nonordfp: An FP-growth variation without rebuilding the FP-tree," In *Proc. IEEE ICDM'04 Workshop FIMI'04*, 2004.

[3] R. J. Bayardo Jr., "Efficiently Mining Long Patterns from Databases", In *Proc. SIGMOD'98*, pp. 85–93, 1998.

[4] D. Burdick, M. Calimlim, J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," In *Proc. ICDE 2001*, pp. 443-452, 2001.

[5] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, "MAFIA: A Performance Study of Mining Maximal Frequent Itemsets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[6] B. Goethals, *the FIMI repository*, `http://fimi.cs.helsinki.fi/`, 2003.

[7] G. Grahne and J. Zhu, "Efficiently Using Prefix-trees in Mining Frequent Itemsets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[8] J. Han, J. Pei, Y. Yin, "Mining Frequent Patterns without Candidate Generation," *SIGMOD Conference 2000*, pp. 1-12, 2000

[9] Guimei Liu, Hongjun Lu, Jeffrey Xu Yu, Wang Wei, and Xiangye Xiao, "AFOPT: An Efficient Implementation of Pattern Growth Approach," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[10] S. Orlando, C. Lucchese, P. Palmerini, R. Perego and F. Silvestri, "kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[11] C. Lucchese, S. Orlando and R. Perego, "DCI Closed: A Fast and Memory Efficient Algorithm to Mine Frequent Closed Itemsets," In *Proc. IEEE ICDM'04 Workshop FIMI'04*, 2004.

[12] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, *Efficient Mining of Association Rules Using Closed Itemset Lattices*, *Inform. Syst.*, 24(1), 25–46, 1999.

[13] A. Pietracaprina and D. Zandolin, "Mining Frequent Itemsets using Patricia Tries," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[14] T. Uno, T. Asai, Y. Uchida, H. Arimura, "LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003.

[15] T. Uno, T. Asai, Y. Uchida, H. Arimura, "An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases," *Lecture Notes in Artificial Intelligence* 3245, pp. 16–31, 2004.

[16] T. Uno, M. Kiyomi, H. Arimura, "LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets", In *Proc. IEEE ICDM'04 Workshop FIMI'04*, 2004.