The Relation of Closed Itemset Mining, Complete Pruning Strategies and Item Ordering in Apriori-Based FIM Algorithms

Ferenc Bodon[1,*] and Lars Schmidt-Thieme[2]

[1] Department of Computer Science and Information Theory,
Budapest University of Technology and Economics
bodon@cs.bme.hu
[2] Computer Based New Media Group (CGNM),
Albert-Ludwigs-Universität Freiburg
lst@informatik.uni-freiburg.de

Abstract. In this paper we investigate the relationship between closed itemset mining, the complete pruning technique and item ordering in the Apriori algorithm. We claim, that when proper item order is used, complete pruning does not necessarily speed up Apriori, and in databases with certain characteristics, pruning increases run time significantly. We also show that if complete pruning is applied, then an intersection-based technique not only results in a faster algorithm, but we get free closed-itemset selection concerning both memory consumption and run-time.

1 Introduction

Frequent itemset mining (FIM) is a popular and practical research field of data mining. Techniques and algorithms developed here are used in the discovery of association rules, sequential patterns, episode rules, frequent trees and subgraphs, and classification rules. The set of *frequent closed itemsets* (FC) is an important subset of the frequent itemsets (F) because it offers a compact representation of F. This means that FC contains fewer elements, and from FC we can completely determine the frequent itemsets [1].

Over 170 FIM and FCIM algorithms have been proposed in the last decade, each claiming to outperform its existing rivals [2]. Thanks to some comparisons from independent authors (the FIMI competitions [2] are regarded to be the most important), the chaos seems to be settling. The most successful algorithms are Apriori [3], ECLAT [4][5], FP-growth [6] and variants of these. Adaptations of these algorithms used to extract closed itemsets are also the most popular and most efficient FCIM algorithms.

Apriori is regarded to be the first FIM algorithm that can cope with large datasets. One of the most important surprises of the FIM competition was that

* 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).

A. Jorge et al. (Eds.): PKDD 2005, LNAI 3721, pp. 437–444, 2005.
© Springer-Verlag Berlin Heidelberg 2005

this algorithm is competitive regarding run time (particularly at high support thresholds), and its memory need was outstandingly low in many cases. Moreover, the resulting closed extension, Apriori-Close [1], is the best algorithm for certain sets of test. An inherent feature of Apriori is *complete pruning*, which only allows the generation of candidates that possess only frequent subsets. Due to complete pruning Apriori never generated more candidates than those algorithms which traverse the itemset space in a depth-first manner (DFS algorithms), as do Eclat and FP-growth. Complete pruning in Apriori is considered to be so essential, that the frequent pattern mining community has accepted it as a rule of thumb.

In this paper, we investigate the efficiency of complete pruning, and draw the surprising conclusion, that this technique is not as necessary as once believed. If the database has a certain characteristic, then pruning may even slow down Apriori. We also show, that the efficiency of pruning depends on the item ordering used during the algorithm.

We also investigate the connection between pruning and closed-itemset selection. By presenting a novel pruning strategy, we will show that closed-itemset mining comes for free. In Apriori-Close, this does not hold because closed itemset selection is merged into the phase where infrequent candidates are removed, and requires many scans of the data structure which stores the frequent itemsets. This can be saved by applying our new pruning strategy.

2 Problem Statement

Frequent itemset mining came from efforts to discover useful patterns in customers' transaction databases. A customers' transaction database is a sequence of transactions ($\mathcal{T} = \langle t_1, \ldots, t_n \rangle$), where each transaction is an itemset ($t_i \subseteq \mathcal{I}$). An itemset with k elements is called a k-itemset. The *support* of an itemset X in $\mathcal{T}$, denoted as $supp_{\mathcal{T}}(X)$, is the number of transactions containing X, i.e. $supp_{\mathcal{T}}(X) = |\{t_j : X \subseteq t_j\}|$. An itemset is *frequent* if its support is greater than a *support threshold*, originally denoted by *min_supp*. The frequent itemset mining problem is to discover all frequent itemsets in a given transaction database.

Itemset I is *closed* if no proper superset of I exists that has the same support as I. The set of closed itemsets is a compact representation of the frequent itemsets. All frequent itemsets together with their supports can be generated if only the closed itemsets and their supports are known. In some databases the number of closed itemsets is much smaller than the number of frequent sets, thus it is an important data mining task to determine FCI.

The concepts of *negative border* and *order-based negative border* play an important role in our contributions. Let F be the set of frequent itemsets, and $\prec$ a total order on the elements of $2^{\mathcal{I}}$. The negative border of F is the set of itemsets, whose elements are infrequent, but all their proper subsets are frequent (formally: $NB(F) = \{I | I \notin F, \forall I' \subset I, I' \in F\}$). The order-based negative border (denoted by $NB^{\prec}(F)$) is a superset of $NB(F)$. 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 a fixed ordering of items. For example, if $\mathcal{I} = \{A, B, C\}$ and $F = \{\emptyset, A, B, C, AB, AC\}$ then $NB(F) = \{BC\}$ and $NB^{\prec}(F) = \{BC, ABC\}$ if $\prec$ is the alphabetic order.

In the rest of the paper the ascending and descending order according to supports of the items are denoted by $\prec_D$ and $\prec_A$ respectively.

The Apriori algorithm plays a central role in frequent itemset mining. Although it is one of the oldest algorithms, the intensive researches that polished its data structure and implementation specific issues [7] [8] have raised it to a competitive algorithm which outperforms the newest DFS algorithms in some cases [2]. We assume that the reader is familiar with the Apriori algorithm. In this paper we concentrate on its candidate generation method.

3 Candidate Generation of Apriori

To understand our claims we also need to understand the main data structure of Apriori, i.e. the *trie* (also called *prefix-tree*). The data structure trie was originally introduced by de la Briandais [9] and Fredkin to store and efficiently retrieve words (i.e. sequence of letters) of a dictionary. In the FIM setting the alphabet is the set of items, and the itemsets are converted to sequences by a predefined order. A trie is a rooted, (downward) directed tree. 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 called *edge* or *link*, which is labeled by an item. If node u points to node v, then we call u the *parent* of v, and v is a *child node* of u. Nodes with no child are called *leaves*.

Every leaf ℓ represents an itemset which is the union of the letters in the path from the root to ℓ. Note that if the first k letters are the same in two words, then the first k steps on their paths are the same as well. In the rest of the paper the node that represents itemset I is referred to node I. For more details about the usage of the trie data structure in Apriori the reader is referred to [7][8].

In Apriori's candidate generation phase we generate $(\ell + 1)$-itemset candidates. Itemset I becomes a candidate if all proper subsets of I are frequent. The trie that stores the frequent items, supports this method. Each itemset that fulfills the complete pruning requirement can be obtained by taking the union of the representations of two sibling nodes. In the so called *simple pruning* we go through all nodes at depth $\ell - 1$, take the pairwise union of the children and then check all subsets of the union if they are frequent. Two straightforward modifications can be applied to reduce unnecessary work. On one hand, we do not check those subsets that are obtained by removing the last and the one before the last elements of the union. On the other hand, the prune check is terminated as soon as a subset is infrequent, i.e. not contained in the trie.

3.1 Pruning by Intersection

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 the four frequent 4-itemsets. When we check the subsets 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 subset of ABC 6 times.

To save these superfluous traversals we propose an *intersection-based pruning* method. 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 $(\ell-2)$-element subsets 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 the 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, hence, in generating of the children of siblings, we can use the same $v_1, v_2, \ldots, v_{\ell-1}$ nodes. It is enough to find their children with the proper label (the new v'_j nodes) and to make the intersection of the labels of edges that starts from the prefix and the new $v'_1, v'_2, \ldots, v'_{\ell-1}$. This is the real advantage of this method. The $(\ell-2)$-subset nodes of the prefix are reused, hence the paths representing the subsets are traversed only once, instead of $\binom{n}{2}$, where n is the number of the children of the prefix.

As an illustrative example let us assume that the trie that is obtained after removing infrequent itemsets of size 4 is depicted in Fig. 1.

To get the children of node $ABCD$ that fulfill complete pruning requirement (all subsets are frequent), we find the nodes that represent the 2-subsets of the prefix (ABC). These nodes are denoted by v_1, v_2, v_3. Next we find their children that are reached by edges with label D. These children are denoted by v'_1, v'_2

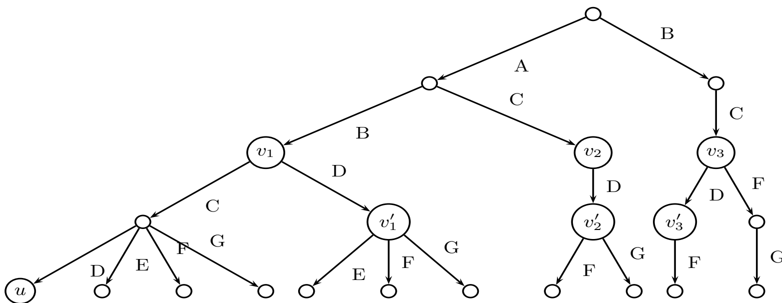

Fig. 1. Example: intersection-based pruning

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.

3.2 Closed Itemset Selection

Closed itemsets can be retrieved from frequent itemsets by post-processing, but it results in a faster solution, if the selection is pushed into the FIM algorithm. In Apriori-Close the infrequent candidate deletion is extended by a step, where the subsets of the frequent candidate are checked. By default all subsets are marked as closed, which is changed if the subsets' support equals to the candidate's actually examined. Consequently, in Apriori-Close all subsets of the frequent candidates are generated, which mean many travels in the trie.

These superfluous travels are avoided if the closed itemset filtering is done in the candidate generation phase and intersection-based pruning is applied. In this method the subsets are already determined, hence checking support equivalence does not require any extra travels.

4 Item Ordering and the Pruning Efficiency

Our previous work has [8] shown that the order of items used in the trie to convert itemsets to sequences greatly affects both run-time and memory need of the Apriori. Next we show that the efficacy of complete pruning is also influenced by this factor.

The advantage of the pruning is to reduce the number of candidates. The number of candidates in Apriori equals to the number of frequent itemsets plus the number of infrequent candidates, i.e. the negative border of the frequent itemsets. If pruning is not used then the number of infrequent candidates becomes the size of the order-based negative border, where the order corresponds to the order used in the trie. It follows, that if we want to decrease the redundant work (i.e determining a support of the infrequent candidates) then we have to use the order that results in the smallest order-based negative border. This comes into play in all DFS algorithms, so we already know the answer: the ascending order according to supports achieves in most cases the best result. This is again a rule of thumb, that works well on real and synthetic datasets. The statement cannot be proven unless the distribution of the items is known and the independence of the items is assumed.

The disadvantage of the pruning strategy is simple: we have to traverse some part of the trie to decide if all subsets are frequent or not. Obviously this needs some time.

Here we state that pruning is not necessarily an important part of Apriori. This statement is supported by the following observation, that applies in most cases:

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

The left-hand side of the inequality gives the number of infrequent itemsets that are not candidates in the original Apriori, but are candidates in Apriori-NOPRUNE. So the left-hand side is proportional to the extra work to be done by omitting pruning. On the other hand, $|F|$ is proportional to the extra work done with pruning. Candidate generation with pruning checks all the subsets of each element of F, while Apriori-NOPRUNE does not. The outcomes of the two approaches are the same for frequent itemsets, but the pruning-based solution determines the outcome with much more work (i.e traverses the trie many times).

Although the above inequality holds for most cases, this does not imply that pruning is unnecessary, and slows down Apriori. The extra work is just proportional to the formulas above. Extra work caused by omitting pruning means determining the support of some candidates, which is affected by many factors, such as the size of these candidates, the number of transactions, the number of elements in the transactions, and the length of matching prefixes in the transaction. The extra work caused by pruning comes in a form of redundant traversals of the tree during checking the subsets.

As soon as pruning strategy is omitted, Apriori can be further tuned by merging the candidate generation and the infrequent node deletion phases. After removing the infrequent children of a node, we extend each child the same way as we would do in candidate generation. This way we spare an entire traversal of the trie.

5 Experiments

All tests were carried out on 16 public "benchmark" databases, which can be downloaded from the FIMI repository[1]. Results would require too much space, hence only the most typical ones are shown below. All results, all programs as well as the test scripts can be downloaded from http://www.cs.bme.hu/~bodon/en/fim/test.html. For Apriori implementation we have used an improved version of our code, that took part in the FIMI'04 competition, and reached many times outstanding results concerning memory requirement. Due to the improvements it is now a true rival of the best Apriori implementation [7] and outperforms it in many cases. The code can be downloaded from http://fim.informatik.uni-freiburg.de.

Comparing just pruning techniques, Apriori-IBP (Apriori that uses intersection-based pruning) was always faster than Apriori-SP (simple pruning), however, the differences were insignificant in many cases. The intersection-based pruning was 25% - 100% faster than the original solution at databases `BMS-WebView-1`, `BMS-WebView-2`, `T10I5N1KP5KC0.25D200K`.

It is not so easy to declare a winner in the competition of Apriori-IBP and Apriori-NOPRUNE. Apriori-NOPRUNE was faster in 85% of the tests, however in most cases the difference was under 10%. Using low support threshold six measurements showed significant differences. In the case of `BMS-WebView-1` and

[1] `http://fimi.cs.helsinki.fi/data/`

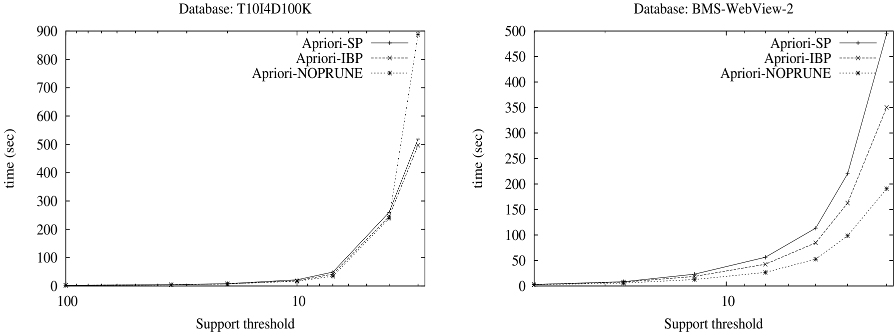

Fig. 2. Candidate generation with different pruning strategies

`BMS-WebView-2` Apriori-NOPRUNE was fast twofold faster than Apriori-IBP, but in T10I4D100K the contrary was true. Figure 2 shows the run-times of these cases, and the run-time of a typical result (Apriori-IBP slightly faster than Apriori-SP; Apriori-NOPRUNE is 10%-20% faster than Apriori-IBP).

To understand why Apriori-IBP was the faster in the first case and why Apriori-NOPRUNE in the second, we have to examine the number of candidates generated by the two algorithms and the number of frequent itemsets. These data are summarized in the next table (for the sake of better readability the numbers of itemsets are divided by 1000).

Table 1. Number of frequent itemsets and number of candidates

database	*min_supp*	$\lvert F\rvert$	$\lvert NB(F)\rvert$	$\lvert NB^{\prec_A}\rvert$	$\lvert NB^{\prec_D}\rvert$	$\frac{\lvert NB^{\prec_A}\rvert - \lvert NB\rvert}{\lvert F\rvert}$
T10I4D100K	3	5 947	39 404	92 636	166 461	8.95
BMS-WebView-2	4	60 083	3 341	9 789	197 576	0.11

The data support our hypothesis. The ratio of number of frequent itemset to the difference of the two negative borders greatly determines pruning efficiency. Obviously, if the number of extra candidates is insignificant compared to the number of frequent itemsets, then pruning slows down Apriori. The table also shows the importance of the proper order. In the case of `T10I4D100K` the $\lvert NB^{\prec_D}\rvert$ is so large that the algorithm did not fit in the 2GB of main memory. This does not occur at `BMS-WebView-2`, but the number of infrequent candidates was 20-fold more compared to the ascending order based solution.

6 Conclusions

In this paper, we have proposed an intersection-based pruning strategy that outperforms the classic candidate-generation method. The other advantage of the

method is that closed-itemset selection comes for free. Since the new candidate-generation method does not affect any other part of the algorithm, it can also be applied in Apriori-Close to obtain an improved version.

The major contribution of the paper is the investigation of the pruning efficiency in Apriori. We claim that, if ascending order is used, then pruning does not necessarily speed-up the algorithm, and if $(|NB^{\prec_A}(F)| - |NB(F)|)/|F|$ is small, then the run-time increases in most cases. Note that this conclusion does not only affect Apriori and its variants, but also all those Apriori modifications that discover other type of frequent patterns, like sequences, episodes, boolean formulas, trees or graphs. Since in such cases subpattern inclusion check is more complicated (for example in the case of labeled graphs this requires a graph isomorphism test) the difference can be more significant, and thus needs to be investigated.

References

1. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Pruning closed itemset lattices for association rules. In: Proceedings of the 14th BDA French Conference on Advanced Databases, Hammamet, Tunisie (1998) 177–196
2. Goethals, B., Zaki, M.J.: Advances in frequent itemset mining implementations: Introduction to fimi03. In: Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03). Volume 90 of CEUR Workshop Proceedings., Melbourne, Florida, USA (2003)
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In Bocca, J.B., Jarke, M., Zaniolo, C., eds.: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), Chile, Morgan Kaufmann (1994) 487–499
4. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. In Heckerman, D., Mannila, H., Pregibon, D., Uthurusamy, R., Park, M., eds.: Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining, California, USA, AAAI Press (1997) 283–296
5. Schmidt-Thieme, L.: Algorithmic features of eclat. In: Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04). Volume 126 of CEUR Workshop Proceedings., Brighton, UK (2004)
6. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proceedings of the 2000 ACM SIGMOD international conference on Management of data, Dallas, Texas, United States, ACM Press (2000) 1–12
7. Borgelt, C.: Efficient implementations of apriori and eclat. In: Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03). Volume 90 of CEUR Workshop Proceedings., Melbourne, Florida, USA (2003)
8. Bodon, F.: Surprising results of trie-based fim algorithms. In: Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'04). Volume 126 of CEUR Workshop Proceedings., Brighton, UK (2004)
9. de la Briandais, R.: File searching using variable-length keys. In: Proceedings of the Western Joint Computer Conference. (1959) 295–298