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.

*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).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OSDM'05, August 21, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-210-0/05/08 ...$5.00.

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 $z \npreceq x$.

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

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 ℓth 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 ℓth 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.

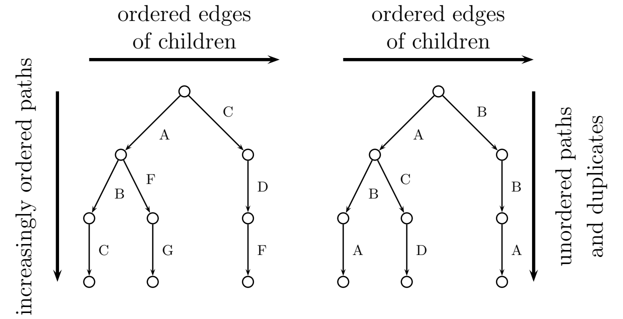

Figure 1: Tries of itemsets and item sequences

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 jth 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.

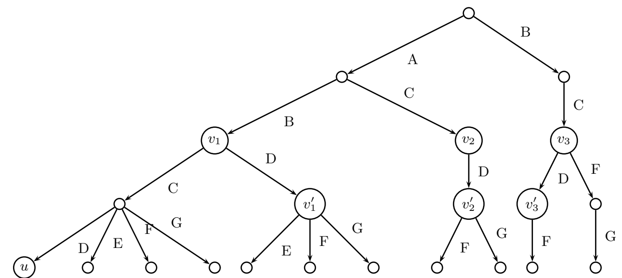

Figure 2: Example: intersection-based pruning

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.

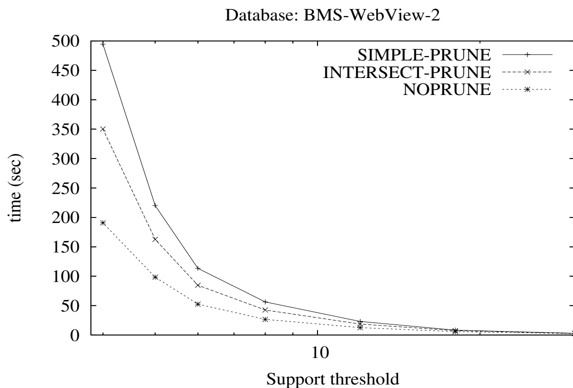

Figure 3: Candidate generation of itemsets with different pruning strategies

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.

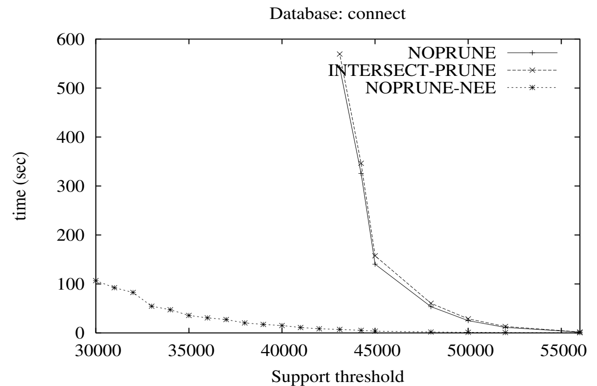

Figure 4: Omitting equisupport extensions (itemset case)

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

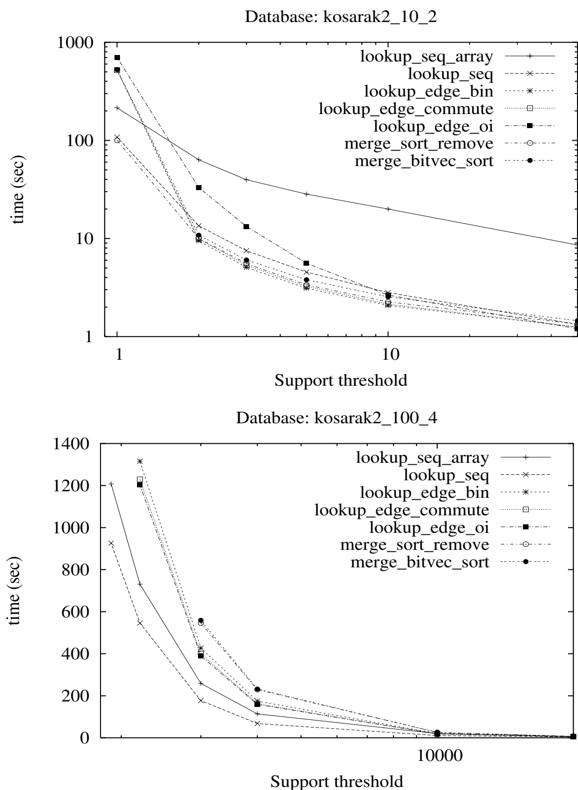

Figure 5: Routing strategies

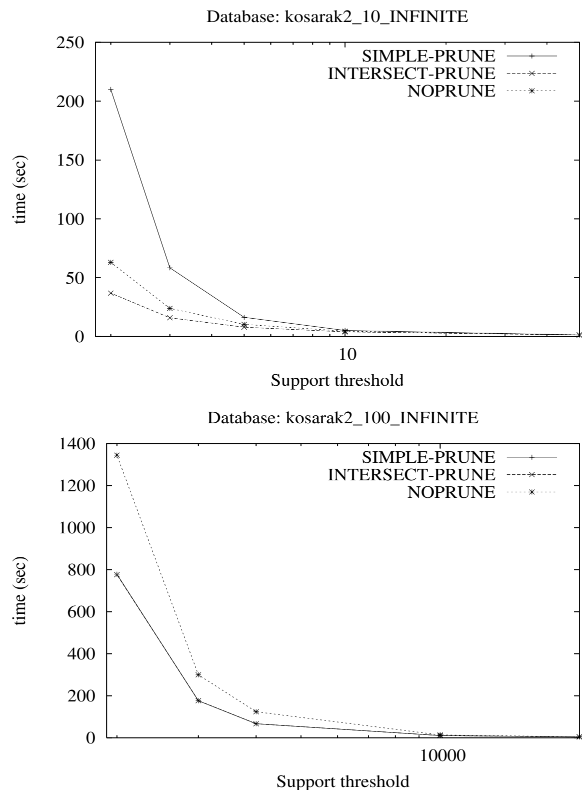

Figure 6: Candidate generation with different pruning strategies

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

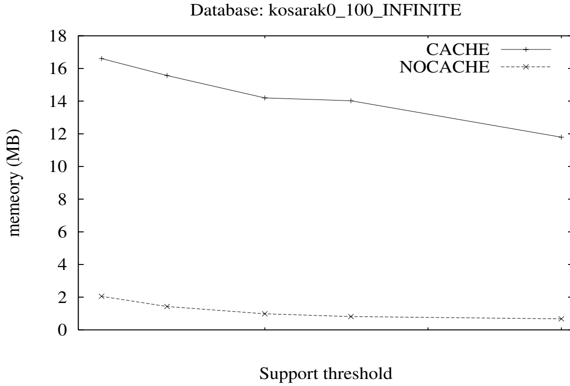

Figure 7: Transaction caching: effect on memory need

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	∞

$kosarak_100_\infty$

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

$kosarak2_10_2$

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

$kosarak2_100_\infty$

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	

$kosarak_100_\infty$

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

$kosarak2_10_2$

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

$kosarak2_100_\infty$

- 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	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{I}\rvert$	$\lvert t\rvert$
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 yth 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.