A fast APRIORI implementation

Ferenc Bodon*
Informatics Laboratory, Computer and Automation Research Institute,
Hungarian Academy of Sciences
H-1111 Budapest, Lágymányosi u. 11, Hungary

Abstract

The efficiency of frequent itemset mining algorithms is determined mainly by three factors: the way candidates are generated, the data structure that is used and the implementation details. Most papers focus on the first factor, some describe the underlying data structures, but implementation details are almost always neglected. In this paper we show that the effect of implementation can be more important than the selection of the algorithm. Ideas that seem to be quite promising, may turn out to be ineffective if we descend to the implementation level.

We theoretically and experimentally analyze APRIORI which is the most established algorithm for frequent itemset mining. Several implementations of the algorithm have been put forward in the last decade. Although they are implementations of the very same algorithm, they display large differences in running time and memory need. In this paper we describe an implementation of APRIORI that outperforms all implementations known to us. We analyze, theoretically and experimentally, the principal data structure of our solution. This data structure is the main factor in the efficiency of our implementation. Moreover, we present a simple modification of APRIORI that appears to be faster than the original algorithm.

1 Introduction

Finding frequent itemsets is one of the most investigated fields of data mining. The problem was first presented in [1]. The subsequent paper [3] is considered as one of the most important contributions to the subject. Its main algorithm, APRIORI, not only influenced the association rule mining community, but it affected other data mining fields as well.

Association rule and frequent itemset mining became a widely researched area, and hence faster and faster algorithms have been presented. Numerous of them are APRIORI based algorithms or APRIORI modifications. Those who adapted APRIORI as a basic search strategy, tended to adapt the whole set of procedures and data structures as well [20][8][21][26]. Since the scheme of this important algorithm was not only used in basic association rules mining, but also in other data mining fields (hierarchical association rules [22][16][11], association rules maintenance [9][10][24][5], sequential pattern mining [4][23], episode mining [18] and functional dependency discovery [14] [15]), it seems appropriate to critically examine the algorithm and clarify its implementation details.

A central data structure of the algorithm is trie or hash-tree. Concerning speed, memory need and sensitivity of parameters, tries were proven to outperform hash-trees [7]. In this paper we will show a version of trie that gives the best result in frequent itemset mining. In addition to description, theoretical and experimental analysis, we provide implementation details as well.

The rest of the paper is organized as follows. In Section 1 the problem is presented, in Section 2 tries are described. Section 3 shows how the original trie can be modified to obtain a much faster algorithm. Implementation is detailed in Section 4. Experimental details are given in Section 5. In Section 7 we mention further improvement possibilities.

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. In the rest of the paper we make the (realistic) assumption that the items are from an ordered set, and transactions are stored as sorted itemsets. The support of an itemset X in $\mathcal{T}$, denoted as $supp_{\mathcal{T}}(X)$, is the number of those transactions that contain 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

*Research supported in part by OTKA grants T42706, T42481 and the EU-COE Grant of MTA SZTAKI.

itemset mining problem is to find all frequent itemset in a given transaction database.

The first, and maybe the most important solution for finding frequent itemsets, is the APRIORI algorithm [3]. Later faster and more sophisticated algorithms have been suggested, most of them being modifications of APRIORI [20][8][21][26]. Therefore if we improve the APRIORI algorithm then we improve a whole family of algorithms. We assume that the reader is familiar with APRIORI [2] and we turn our attention to its central data structure.

3. Determining Support with a Trie

The data structure *trie* was originally introduced to store and efficiently retrieve words of a dictionary (see for example [17]). A trie is a rooted, (downward) directed tree like a hash-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 a letter. There exists a special letter * which represents an "end" character. If node u points to node v, then we call u the parent of v, and v is a child node of u.

Every leaf ℓ represents a word which is the concatenation 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.

Tries are suitable to store and retrieve not only words, but any finite ordered sets. In this setting a link is labeled by an element of the set, and the trie contains a set if there exists a path where the links are labeled by the elements of the set, in increasing order.

In our data mining context the alphabet is the (ordered) set of all items $\mathcal{I}$. A candidate k-itemset

$$C = \{i_1 < i_2 < \ldots < i_k\}$$

can be viewed as the word $i_1 i_2 \ldots i_k$ composed of letters from $\mathcal{I}$. We do not need the * symbol, because every inner node represent an important itemset (i.e. a meaningful word).

Figure 1 presents a trie that stores the candidates {A,C,D}, {A,E,G}, {A,E,L}, {A,E,M}, {K,M,N}. Numbers in the nodes serve as identifiers and will be used in the implementation of the trie. Building a trie is straightforward, we omit the details, which can be found in [17].

In support count method we take the transactions one-by-one. For a transaction t we take all ordered k-subsets X of t and search for them in the trie structure. If X is found (as a candidate), then we increase the support count of this candidate by one. Here, we do not generate all k-subsets of t, rather we perform early quits if possible. More precisely, if we are at a node at depth d by following the j^{th} item link, then we move forward on those links that have the labels $i \in t$ with index greater than j, but less than $|t| - k + d + 1$.

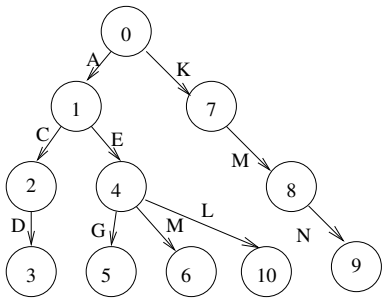

Figure 1. A trie containing 5 candidates

In our approach, tries store not only candidates, but frequent itemsets as well. This has the following advantages:

1. Candidate generation becomes easy and fast. We can generate candidates from pairs of nodes that have the same parents (which means, that except for the last item, the two sets are the same).

2. Association rules are produced much faster, since retrieving a support of an itemset is quicker (remember the trie was originally developed to quickly decide if a word is included in a dictionary).

3. Just one data structure has to be implemented, hence the code is simpler and easier to maintain.

4. We can immediately generate the so called *negative border*, which plays an important role in many APRIORI-based algorithm (online association rules [25], sampling based algorithms [26], etc.).

3.1 Support Count Methods with Trie

Support count is done, by reading transactions one-by-one and determine which candidates are contained in the actual transaction (denoted by t). Finding candidates in a given transaction determines the overall running time primarily. There are two simple recursive methods to solve this task, both starts from the root of the trie. The recursive step is the following (let us denote the number of edges of the actual node by m).

1. For each item in the transaction we determine whether there exists an edge whose label corresponds to the item. Since the edges are ordered according to the labels this means a search in an ordered set.

2. We maintain two indices, one for the items in the transaction, one for the edges of the node. Each index is initialized to the first element. Next we check whether the element pointed by the two indices equals. If they do,

we call our procedure recursively. Otherwise we increase the index that points to the smaller item. These steps are repeated until the end of the transaction or the last edge is reached.

In both methods if item i of the transaction leads to a new node, then item j is considered in the next step only if $j > i$ (more precisely item j is considered, if $j < |t| + \text{actual\_depth} - m + 1$).

Let us compare the running time of the methods. Since both methods are correct, the same branches of the trie will be visited. Running time difference is determined by the recursive step. The first method calls the subroutine that decides whether there exist an edge with a given label $|t|$ times. If binary search is evoked then it requires $\log_2 m$ steps. Also note that subroutine calling needs as many value assignments as many parameters the subroutine has. We can easily improve the first approach. If the number of edges is small (i.e. if $|t|^m < m^{|t|}$) we can do the inverse procedure, i.e. for all labels we check whether there exists a corresponding item. This way the overall running time is proportional to $\min\{|t| \log_2 m, m \log_2 |t|\}$.

The second method needs at least $min\{m, |t|\}$ and at most $m + |t|$ steps and there is no subroutine call.

Theoretically it can happen that the first method is the better solution (for example if $|t|$=1, m is large, and the label of the last edge corresponds to the only item in the transaction), however in general the second method is faster. Experiments showed that the second method finished 50% faster on the average.

Running time of support count can be further reduced if we modify the trie a little. These small modifications, tricks are described in the next subsections.

3.2 Storing the Length of Maximal Paths

Here we show how the time of finding supported candidates in a transaction can be significantly reduced by storing a little extra information. The point is that we often perform superfluous moves in trie search in the sense that there are no candidates in the direction we are about to explore. To illustrate this, consider the following example. Assume that after determining frequent 4-itemsets only candidate $\{A, B, C, D, E\}$ was generated, and Figure 2 shows the resulting trie.

If we search for 5-itemset candidates supported by the transaction $\{A, B, C, D, E, F, G, H, I\}$, then we must visit every node of the trie. This appears to be unnecessary since only one path leads to a node at depth 5, which means that only one path represents a 5-itemset candidate. Instead of visiting merely 6 nodes, we visit 32 of them. At each node, we also have to decide which link to follow, which can greatly affect running time if a node has many links.

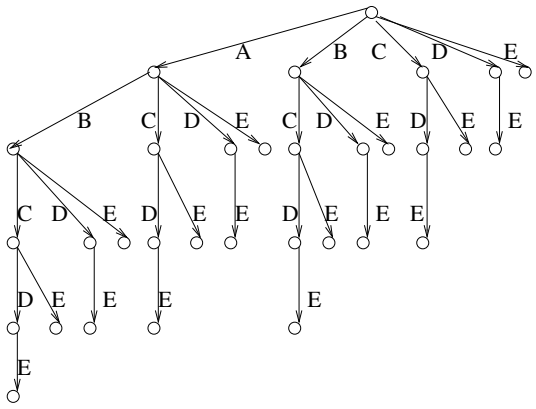

Figure 2. A trie with a single 5-itemset candidate

To avoid this superfluous traveling, at every node we store the length of the longest directed path that starts from there. When searching for k-itemset candidates at depth d, we move downward only if the maximal path length at this node is at least $k - d$. Storing counters needs memory, but as experiments proved, it can seriously reduce search time for large itemsets.

3.3 Frequency Codes

It often happens that we have to find the node that represents a given itemset. For example, during candidate generation, when the subsets of the generated candidate have to be checked, or if we want to obtain the association rules. Starting from the root we have to travel, and at depth d we have to find the edge whose label is the same as the d^{th} element of the itemset.

Theoretically, binary search would be the fastest way to find an item in an ordered list. But if we go down to the implementation level, we can easily see that if the list is small the linear search is faster than the binary (because an iteration step is faster). Hence the fastest solution is to apply linear search under a threshold and binary above it. The threshold does not depend on the characteristic of the data, but on the ratio of the elementary operations (value assignment, increase, division, ...)

In linear search, we read the first item, compare with the searched item. If it is smaller, then there is no edge with this item, if greater, we step forward, if they equal then the search is finished. If we have bad luck, the most frequent item has the highest order, and we have to march to the end of the line whenever this item is searched for.

On the whole, the search will be faster if the order of items corresponds to the frequency order of them. We know exactly the frequency order after the first read of the whole

database, thus everything is at hand to build the trie with the frequency codes instead of the original codes. The *frequency code* of an item i is $fc[i]$, if i is the $fc[i]^{th}$ most frequent item. Storing frequency codes and their inverses increases the memory need slightly, in return it increases the speed of retrieving the occurrence of the itemsets. A theoretical analysis of the improvement can be read in the Appendix.

Frequency codes also affect the structure of the trie, and consequently the running time of support count. To illustrate this, let us suppose that 2 candidates of size 3 are generated: $\{A, B, C\}, \{A, B, D\}$. Different tries are generated if the items have different code. The next figure present the tries generated by two different coding.

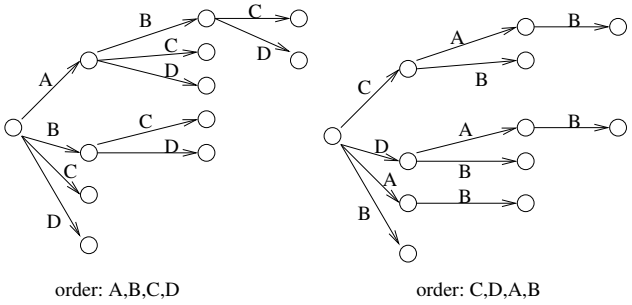

Figure 3. Different coding results in different tries

If we want to find which candidates are stored in a basket $\{A, B, C, D\}$ then 5 nodes are visited in the first case and 7 in the second case. That does not mean that we will find the candidates faster in the first case, because nodes are not so "fat" in the second case, which means, that they have fewer edges. Processing a node is faster in the second case, but more nodes have to be visited.

In general, if the codes of the item correspond to the frequency code, then the resulted trie will be unbalanced while in the other case the trie will be rather balanced. Neither is clearly more advantageous than the other. We choose to build unbalanced trie, because it travels through fewer nodes, which means fewer recursive steps which is a slow operation (subroutine call with at least five parameters in our implementation) compared to finding proper edges at a node.

In [12] it was showed that it is advantageous to recode frequent items according to ascending order of their frequencies (i.e.: the inverse of the frequency codes) because candidate generation will be faster. The first step of candidate generation is to find siblings and take the union of the itemset represented by them. It is easy to prove that there are less sibling relations in a balanced trie, therefore less unions are generated and the second step of the candidate generation is evoked fewer times. For example in our figure one union would be generated and then deleted in the first case and none would be generated in the second.

Altogether frequency codes have advantages and disadvantages. They accelerate retrieving the support of an itemset which can be useful in association rule generation or in on-line frequent itemset mining, but slows down candidate generation. Since candidate generation is by many order of magnitude faster that support count, the speed decrease is not noticeable.

3.4 Determining the support of 1- and 2-itemset candidates

We already mentioned that the support count of 1-element candidates can easily be done by a single vector. The same easy solution can be applied to 2-itemset candidates. Here we can use an array [19]. The next figure illustrates the solutions.

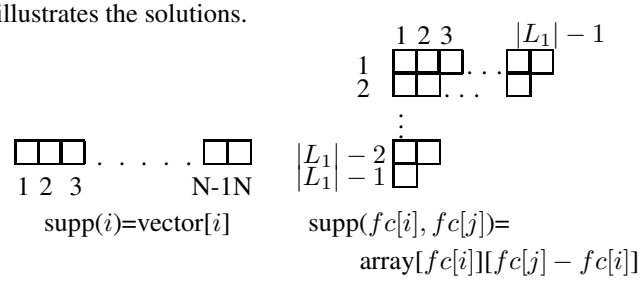

Figure 4. Data structure to determine the support of the items and candidate itempairs

Note that this solution is faster than the trie-based solution, since increasing a support takes one step. Its second advantage is that it needs less memory.

Memory need can be further reduced by applying on-the-fly candidate generation [12]. If the number of frequent items is $|L_1|$ then the number of 2-itemset candidates is $\binom{|L_1|}{2}$, out of which a lot will never occur. Thus instead of the array, we can use trie. A 2-itemset candidate is inserted only if it occurs in a basket.

3.5 Applying hashing techniques

Determining the support with a trie is relatively slow when we have to step forward from a node that has many edges. We can accelerate the search if hash-tables are employed. This way the cost of a step down is calculating one hash value, thus a recursive step takes exactly $|t|$ steps. We want to keep the property that a leaf represents exactly one itemset, hence we have to use perfect hashing. Frequency codes suit our needs again, because a trie stores only frequent items. Please note, that applying hashing techniques at tries does not result in a hash-tree proposed in [3].

It is wasteful to change all inner nodes to hash-table, since a hash-table needs much more memory than an ordered list of edges. We propose to alter only those inner nodes into a hash-table, which have more edges than a reasonable threshold (denoted by $leaf\_max\_size$). During trie construction when a new leaf is added, we have to check whether the number of its parent's edges exceeds $leaf\_max\_size$. If it does, the node has to be altered to hash-table. The inverse of this transaction may be needed when infrequent itemsets are deleted.

If the frequent itemsets are stored in the trie, then the number of edges can not grow as we go down the trie. In practice nodes, at higher level have many edges, and nodes at low level have only a few. The structure of the trie will be the following: nodes over a (not necessarily a horizontal) line will be hash-tables, while the others will be original nodes. Consequently, where it was slow, search will be faster, and where it was fast –because of the small number of edges– it will remain fast.

3.6 Brave Candidate Generation

It is typical, that in the last phases of APRIORI there are a small number of candidates. However, to determine their support the whole database is scanned, which is wasteful. This problem was also mentioned in the original paper of APRIORI [3], where algorithm APRIORI-HYBRID was proposed. If a certain heuristic holds, then APRIORI switches to APRIORI-TID, where for each candidate we store the transactions that contain it (and support is immediately obtained). This results in a much faster execution of the latter phases of APRIORI.

The hard point of this approach is to tell when to switch from APRIORI to APRIORI-TID. If the heuristics fails the algorithm may need too much memory and becomes very slow. Here we choose another approach that we call the brave candidate generation and the algorithm is denoted by APRIORI-BRAVE.

APRIORI-BRAVE keeps track of memory need, and stores the amount of the maximal memory need. After determining frequent k-itemsets it generates $(k+1)$-itemset candidates as APRIORI does. However, it does not carry on with the support count, but checks if the memory need exceeds the maximal memory need. If not, $(k+2)$-itemset candidates are generated, otherwise support count is evoked and maximal memory need counter is updated. We carry on with memory check and candidate generation till memory need does not reach maximal memory need.

This procedure will collect together the candidates of the latter phases and determine their support in one database read. For example, candidates in database T40I10D100K with $min\_supp = 0.01$ will be processed the following way: 1, 2, 3, 4-10, 11-14, which means 5 database scan instead of 14.

One may say that APRIORI-BRAVE does not consume more memory than APRIORI and it can be faster, because there is a possibility that some candidates at different sizes are collected and their support is determined in one read. However, accelerating in speed is not necessarily true. APRIORI-BRAVE may generate $(k+2)$-itemset candidates from frequent k-itemset, which can lead to more false candidates. Determining support of false candidates needs time. Consequently, we cannot guarantee that APRIORI-BRAVE is faster than APRIORI, however, test results prove that this heuristics works well in real life.

3.7 A Deadend Idea- Support Lower Bound

The false candidates problem in APRIORI-BRAVE can be avoided if only those candidates are generated that are surely frequent. Fortunately, we can give a lower bound to the support of a $(k+j)$-itemset candidate based on the support of k-itemsets ($j \geq 0$) [6]. Let $X = X' \cup Y \cup Z$. The following inequality holds:

$$supp(X) \geq supp(X' \cup Y) + supp(X' \cup Z) - supp(X').$$

And hence

$$supp(X) \geq \max_{Y,Z \in X} \{supp(X \setminus Y) + supp(X \setminus Z) - supp(X \setminus Y \setminus Z)\}.$$

If we want to give a lower bound to a support of $(k+j)$-itemset base on support of k-itemset, we can use the generalization of the above inequality ($X = X' \cup x_1 \cup \ldots \cup x_j$):

$$supp(X) \geq supp(X' \cup x_1) + \ldots + supp(X' \cup x_j) - (j-1)supp(X').$$

To avoid false candidate generation we generate only those candidates that are surely frequent. This way, we could say that neither memory need nor running time is worse than APRIORI's. Unfortunately, this is not true!

Test results proved that this method not only slower than original APRIORI-BRAVE, but also APRIORI outperforms it. The reason is simple. Determining the support threshold is a slow operation (we have to find the support of $\binom{k+j}{k-1}j$ itemsets) and has to be executed many times. It loses more time with determining support thresholds than we win by generating sooner some candidates.

The failure of the support-threshold candidate generation is a nice example when a promising idea turns out to be useless at the implementation level.

3.8 Storing input

Many papers in the frequent itemset mining subject focus on the number of the whole database scan. They say

that reading data from disc is much slower than operating in memory, thus the speed is mainly determined by this factor. However, in most cases the database is not so big and it fits into the memory. Behind the scenery the operating system swaps it in the memory and the algorithms read the disc only once. For example, a database that stores 10^7 transaction, and in each transaction there are 6 items on the average needs approximately 120Mbytes, which is a fraction of today's average memory capacity. Consequently, if we explicitly store the simple input data, the algorithm will not speed up, but will consume more memory (because of the double storing), which may result in swapping and slowing down. Again, if we descend to the elementary operation of an algorithm, we may conclude the opposite result.

Storing the input data is profitable if the same transactions are gathered up. If a transaction occurs ℓ times, the support count method is evoked once with counter increment ℓ, instead of calling the procedure ℓ times with counter increment 1. In Borgelt algorithm input is stored in a prefix tree. This is dangerous, because data file can be too large.

We've chosen to store only *reduced transactions*. A reduced transaction stores only the frequent items of the transaction. Storing reduced transactions have all information needed to discover frequent itemsets of larger sizes, but it is expected to need less memory (obviously it depends on min_supp). Reduced transactions are stored in a tree for the fast insertion (if reduced transactions are recode with frequency codes then we almost get an FP-tree [13]). Optionally, when the support of candidates of size k is determined we can delete those reduced transactions that do not contain candidates of size k.

4. Implementation details

APRIORI is implemented in an object-oriented manner in language C++. STL possibilities (`vector`, `set`, `map`) are heavily used. The algorithm (class `Apriori`) and the data structure (class `Trie`) are separated. We can change the data structure (for example to a hash-tree) without modifying the source code of the algorithm.

The baskets in a file are first stored in a `vector<...>`. If we choose to store input –which is the default– the reduced baskets are stored in a `map<vector<...>,unsigned long>`, where the second parameter is the number of times that reduced basket occurred. A better solution would be to apply trie, because `map` does not make use of the fact that two baskets can have the same prefixes. Hence insertion of a basket would be faster, and the memory need would be smaller, since the same prefixes would be stored just once. Because of the lack of time trie-based basket storing was not implemented and we do not delete a reduced basket from the `map` if it did not contain any candidate during some scan.

The class `Trie` can be programmed in many ways. We've chosen to implement it with vectors and arrays. It is simple, fast and minimal with respect to memory need. Each node is described by the same element of the vectors (or row of the arrays). The root belongs to the 0^{th} element of each vector. The following figure shows the way the trie is represented by vectors and arrays.

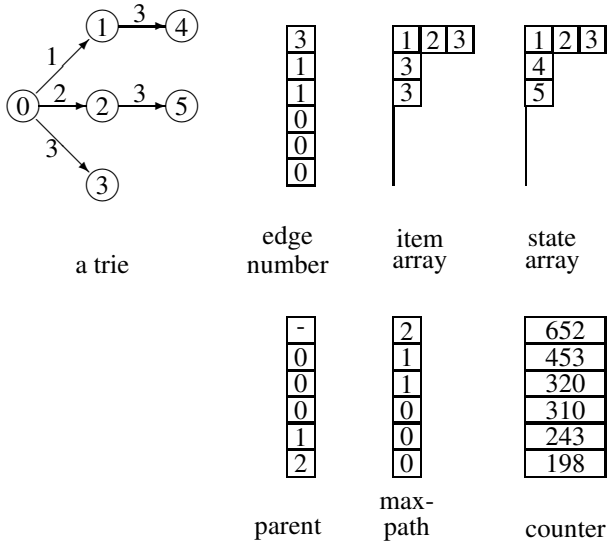

Figure 5. Implementation of a trie

The vector `edge_number` stores the number of edges of the nodes. The `itemarray[i]` stores the label of the edges, `statearray[i]` stores the end node of the edges of node i. Vectors `parent[i]` and `maxpath[i]` store the parents and the length of the longest path respectively. The occurrences of itemset represented by the nodes can be found in vector `counter`.

For vectors we use `vector` class offered by the STL, but arrays are stored in a traditional C way. Obviously, it is not a fixed-size array (which caused the ugly `calloc`, `realloc` commands in the code). Each row is as long as many edges the node has, and new rows are inserted as the trie grows (during candidate generation). A more elegant way would be if the arrays were implemented as a `vector` of `vectors`. The code would be easier to understand and shorter, because the algorithms of STL could also be used. However, the algorithm would be slower because determining a value of the array takes more time. Tests showed that sacrificing a bit from readability leads to 10-15% speed up.

In our implementation we do not adapt on-line 2-itemset candidate generation (see Section 3.4) but use a vector and an array (`temp_counter_array`) for determining the support of 1- and 2-itemset candidates efficiently.

The vector and array description of a trie makes it pos-

sible to give a fast implementation of the basic functions (like candidate generation, support count, ...). For example, deleting infrequent nodes and pulling the vectors together is achieved by a single scan of the vectors. For more details readers are referred to the html-based documentation.

5. Experimental results

Here we present the experimental results of our implementation of APRIORI and APRIORI-BRAVE compared to the two famous APRIORI implementations by Christian Borgelt (version 4.08)[1] and Bart Goethals (release date: 01/06/2003)[2]. 3 databases were used: the well-known T40I10D100K and T10I4D100K and a coded log of a click-stream data of a Hungarian on-line news portal (denoted by *kosarak*). This database contains 990002 transactions of size 8.1 on the average.

Test were run on a PC with 2.8 GHz dual Xeon processors and 3Gbyte RAM. The operating system was Debian Linux, running times were obtained by the `/usr/bin/time` command. The following 3 tables present the test results of the 3 different implementations of APRIORI and the APRIORI_BRAVE on the 3 databases. Each test was carried out 3 times; the tables contain the averages of the results. The two well-known implementations are denoted by the last name of the coders.

min_ supp	Bodon impl.	Borgelt impl.	Goethals impl.	APRIORI_ BRAVE
0.05	8.57	10.53	25.1	8.3
0.030	10.73	11.5	41.07	10.6
0.020	15.3	13.9	53.63	14.0
0.010	95.17	155.83	207.67	100.27
0.009	254.33	408.33	458.7	178.93
0.0085	309.6	573.4	521.0	188.0

Running time (sec.)

Table 1. *T40I10D100K* **database**

Tables 4-6 show result of Bodon's APRIORI implementation with hash techniques. The notation *leaf_max_size* stands for the threshold above a node applies perfect hashing technique.

Our APRIORI implementation beats Goethals implementation almost all the times, and beats Borgelt's implementation many times. It performs best at low support threshold. We can also see that in the case of these 3 databases APRIORI_BRAVE outperforms APRIORI at most support threshold.

Strange, but hashing technique not always resulted in faster execution. The reason for this might be that small vectors are cached in, where linear search is very fast. If we enlarge the size of the vector by altering it into a hashtable, then the vector may be moved into the memory, where read is a slower operation. Applying hashing technique is the other example when an accelerating technique does not result in improvement.

[1] http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html#assoc

[2] http://www.cs.helsinki.fi/u/goethals/software/index.html

min_ supp	Bodon impl.	Borgelt impl.	Goethals impl.	APRIORI_ BRAVE
0.0500	4.23	5.2	11.73	2.87
0.0100	10.27	14.03	30.5	6.6
0.0050	17.87	16.13	40.77	12.7
0.0030	34.07	18.8	53.43	23.97
0.0020	70.3	21.9	69.73	46.5
0.0015	85.23	25.1	86.37	87.63

Running time (sec.)

Table 2. *T10I4D100K* **database**

min_ supp	Bodon impl.	Borgelt impl.	Goethals impl.	APRIORI_ BRAVE
0.050	14.43	32.8	28.27	14.1
0.010	17.9	41.87	44.3	17.5
0.005	24.0	52.4	58.4	21.67
0.003	35.9	67.03	76.77	28.5
0.002	81.83	199.07	182.53	72.13
0.001	612.67	1488.0	1101.0	563.33

Running time (sec.)

Table 3. *kosarak* **database**

min_ supp	leaf_max_size 1	2	5	7	10	25	60	100
0.0500	8.1	8.1	8.1	8.1	8.1	8.1	8.1	8.4
0.0300	9.6	9.8	9.7	9.8	9.8	9.8	9.8	9.9
0.0200	13.8	14.4	13.6	13.9	13.6	13.9	13.9	14.1
0.0100	114.0	96.3	83.8	82.5	78.9	79.2	80.4	83.0
0.0090	469.8	339.6	271.8	258.1	253.0	253.0	251.0	253.8
0.0085	539.0	373.0	340.0	310.0	306.0	306.0	306.0	309.0

Runing time (sec.)

Table 4. T40I10D100K database

min_ supp	leaf_max_size 1	2	5	7	10	25	60	100
0.0500	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8
0.0100	7.8	7.3	6.9	6.9	6.9	6.9	7.0	7.1
0.0050	24.2	14.9	13.4	13.1	13	12.8	13.0	13.2
0.0030	55.3	34.6	30.3	25.9	25.2	25.2	25.3	25.8
0.0020	137.3	100.2	76.2	77.0	75.2	78.0	69.5	64.5
0.0015	235.0	176.0	125.0	130.0	125.0	132.0	115.0	103.0

Running time (sec.)

Table 5. T10I4D100K database

min_ supp	leaf_max_size 1	2	5	7	10	25	60	100
0.0500	14.6	14.3	14.3	14.2	14.2	14.2	14.2	14.2
0.0100	17.5	17.5	17.5	17.6	17.6	17.6	18	18.1
0.0050	21.0	21.0	22.0	21.0	22.0	22.0	22.8	22.8
0.0030	26.3	26.1	26.3	26.5	27.2	27.4	28.5	29.6
0.0020	98.8	77.5	62.3	60.0	59.7	61.0	61.0	63.4
0.0010	1630	1023.0	640.0	597.0	574.0	577.0	572.0	573.0

Running time (sec.)

Table 6. kosarak database

6. Further Improvement and Research Possibilities

Our APRIORI implementation can be further improved if trie is used to store reduced basket, and a reduced basket is removed if it does not contain any candidate.

We mentioned that there are two basic ways of finding the contained candidates in a given transaction. Further theoretical and experimental analysis may lead to the conclusion that a mixture of the two approaches would lead to the fastest execution.

Theoretically, hashing technique accelerates support count. However, experiments did not support this claim. Further investigations are needed to clear the possibilities of this technique.

7. Conclusion

Determining frequent objects (itemsets, episodes, sequential patterns) is one of the most important fields of data mining. It is well known that the way candidates are defined has great effect on running time and memory need, and this is the reason for the large number of algorithms. It is also clear that the applied data structure also influences efficiency parameters. However, the same algorithm that uses a certain data structure has a wide variety of implementation. In this paper, we showed that different implementation results in different running time, and the differences can exceed differences between algorithms. We presented an implementation that solved frequent itemset mining problem in most cases faster than other well-known implementations.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *In Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, 1993.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328, 1996.

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

[4] 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.

[5] N. F. Ayan, A. U. Tansel, and M. E. Arkun. An efficient algorithm to update large itemsets with early pruning. In *Knowledge Discovery and Data Mining*, pages 287–291, 1999.

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

[7] F. Bodon and L. Rónyai. Trie: an alternative data structure for data mining algorithms. *to appear in Computers and Mathematics with Applications*, 2003.

[8] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Record (ACM Special Interest Group on Management of Data),26(2):255*, 1997.

[9] D. W.-L. Cheung, J. Han, V. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *ICDE*, pages 106–114, 1996.

[10] D. W.-L. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Database Systems for Advanced Applications*, pages 185–194, 1997.

[11] Y. Fu. Discovery of multiple-level rules from large databases, 1996.

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

[13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, 05 2000.

[14] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional

and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[15] Y. Huhtala, J. Kinen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *ICDE*, pages 392–401, 1998.

[16] Y. F. Jiawei Han. Discovery of multiple-level association rules from large databases. *In Proc. of the 21st International Conference on Very Large Databases (VLDB), Zurich, Switzerland*, 1995.

[17] D. E. Knuth. *The Art of Computer Programming Vol. 3.* Addison-Wesley, 1968.

[18] 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.

[19] B. Ozden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *ICDE*, pages 412–421, 1998.

[20] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 175–186, San Jose, California, 22–25 1995.

[21] A. Sarasere, E. Omiecinsky, and S. Navathe. An efficient algorithm for mining association rules in large databases. *In Proc. 21st International Conference on Very Large Databases (VLDB), Zurich, Switzerland, Also Gatech Technical Report No. GIT-CC-95-04.*, 1995.

[22] R. Srikant and R. Agrawal. Mining generalized association rules. *In Proc. of the 21st International Conference on Very Large Databases (VLDB), Zurich, Switzerland*, 1995.

[23] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. Technical report, IBM Almaden Research Center, San Jose, California, 1995.

[24] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. page 263.

[25] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Knowledge Discovery and Data Mining*, pages 263–266, 1997.

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

8. Appendix

Let us analyze formally how frequency code accelerate the search. Suppose that the number of frequent items is m, and the j^{th} most frequent has to be searched for m_j times ($n_1 \geq n_2 \geq \ldots \geq n_m$) and $n = \sum_{i=1}^{m} n_i$. If an item is in the position j, then the cost of finding it is $c \cdot j$, where c is a constant. For the sake of simplicity c is omitted. The total cost of search based on frequency codes is $\sum_{j=1}^{m} j \cdot n_j$.

How much is the cost if the list is not ordered by frequencies? We cannot determine this precisely, because we don't know which item is in the first position, which item is in the second, etc. We can calculate the expected time of the total cost if we assume that each order occurs with the same probability. Then the probability of each permutation is $\frac{1}{m!}$. Thus

$$E[\text{total cost}] = \sum_{\pi} \frac{1}{m!} \cdot (\text{cost of } \pi)$$
$$= \frac{1}{m!} \sum_{\pi} \sum_{j=1}^{m} \pi(j) n_{\pi(j)}.$$

Here π runs through the permutations of $1, 2, \ldots, m$, and the j^{th} item of π is denoted by $\pi(j)$. Since each item gets to each position $(m-1)!$ times, we obtain that

$$E[\text{total cost}] = \frac{1}{m!}(m-1)! \sum_{j=1}^{m} n_j \sum_{k=1}^{m} k$$
$$= \frac{1}{m} \frac{(m+1)m}{2} \sum_{j=1}^{m} n_j = \frac{m+1}{2} n.$$

It is easy to prove that $E[\text{total cost}]$ is greater than or equal to the total cost of the search based on frequency codes (because of the condition $n_1 \geq n_2 \geq \ldots \geq n_m$). We want to know more, namely how small the ratio

$$\frac{\sum_{j=1}^{m} j \cdot n_j}{n \frac{m+1}{2}} \tag{1}$$

can be. In the worst case ($n_1 = n_2 = \ldots = n_m$) it is 1, in best case ($n_1 = n - m + 1, n_2 = n_3 = \ldots = n_m = 1$) it converges to 0, if $n \to \infty$.

We can not say anything more unless the probability distribution of frequent items is known. In many applications, there are some very frequent items, and the probability of rare items differs slightly. This is why we voted for an exponential decrease. In our model the probability of occurrence of the j^{th} most frequent item is $p_j = ae^{bj}$, where $a > 0, b < 0$ are two parameters, such that $ae^b \leq 1$ holds. Parameter b can be regarded as the gradient of the distribution, and parameter a determines the starting point[3].

[3] Note that $\sum p_j$ does not have to be equal with 1, since more than one item can occur in a basket.

We suppose, that the ratio of occurrences is the same as the ratio of the probabilities, hence $n_1 : n_2 : \ldots : n_m = p_1 : p_2 : \ldots : p_m$. From this and the condition $n = \sum_{j=1}^{m} n_j$, we infer that $n_j = n\frac{p_j}{\sum_{k=1}^{m} p_k}$. Using the formula for geometric series, and using the notation $x = e^b$ we obtain

$$n_j = n\frac{x^j(x-1)}{x^{m+1}-1} = n\frac{x-1}{x^{m+1}-1} \cdot x^j.$$

The total cost can be determined:

$$\sum_{j=1}^{m} j \cdot n_j = \frac{n(x-1)}{x^{m+1}-1} \sum_{j=1}^{m} j \cdot x^j.$$

Let us calculate $\sum_{j=1}^{m} j \cdot x^j$:

$$\sum_{j=1}^{m} j \cdot x^j = \sum_{j=1}^{m} (j+1) \cdot x^j - \sum_{j=1}^{m} x^j = \Big(\sum_{j=1}^{m} x^{j+1}\Big)' - \sum_{j=1}^{m} x^j$$
$$= \frac{mx^{m+2} - (m+1)x^{m+1} + x}{(x-1)^2}.$$

The ratio of total costs can be expressed in a closed formula:

$$\text{cost ratio} = \frac{2x(mx^{m+1} - (m+1)x^m + 1)}{(x^{m+1}-1)(x-1)(m+1)}, \qquad (2)$$

where $x = e^b$. We can see, that the speedup is independent of a. In Figure 6 3 different distributions can be seen. The first is gently sloping, the second has larger graduation and the last distribution is quite steep.

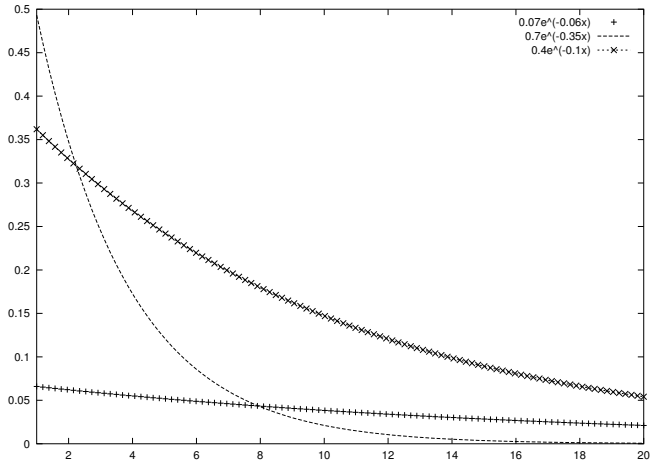

Figure 6. 3 different probability distribution of the items

If we substitute the parameters of the probability distribution to the formula (2) (with $m = 10$), then the result will be 0.39, 0.73 and 0.8, which meets our expectations: by adapting frequency codes the search time will drop sharply if the probabilities differ greatly from each other. We have to remember that frequency codes do not have any effect on nodes where binary search is applied.