Alcom-FT Technical Report Series
ALCOMFT-TR-03-198

A Best-First Strategy For Finding Frequent Sets

J. Baixeries — G. Casas-Garriga — J.L. Balcázar

Departament de LSI. Campus Nord
Universitat Politècnica de Catalunya
Jordi Girona Salgado, 1-3
08034 Barcelona

{jbaixer,gcasas,balqui}@lsi.upc.es

ABSTRACT. *The association rule discovery problem consists in identifying frequent itemsets in a database and, then, forming conditional implication rules among them. The algorithmically most difficult part of this task is finding all frequent sets. There exists a wealth of algorithms both for the problem as such and for variations, particular cases, and generalizations. Except for some recent, fully different approaches, most algorithms can be seen either as a breadth-first search or a depth-first search of the lattice of itemsets. In this paper, we propose a way of developing best-first search strategies.*

RÉSUMÉ. *Le problème de la quête de règles d'association consiste en l'identification d'itemsets fréquents dans une base de données, et puis en produir des règles. La difficulté algorithmique la plus importante de cette tâche est celle de chercher tous les itemsets fréquents. Il y a beaucoup d'algorithmes pour ce même problème et pour les plusieurs variations. Sauf récents solutions très différentes, la majorité des algorithmes peuvent être classifiés en largeur d'abord (breadth-first) ou profondeur d'abord (depth-first). Nous proposons une façon de développer la quête pour les théchniques du meilleur d'abord (best-first).*

KEYWORDS: *data mining, frequent itemsets, association rules, breadth-first search, depth-first search.*

MOTS-CLÉS : *data mining, itemsets fréquents, règles d'association , quête en largeur d'abord, quête en profondeur d'abord.*

NOTE. — This work is supported in part by EU ESPRIT IST-1999-14186 (ALCOM-FT), EU EP27150 (Neurocolt II), Spanish Government PB98-0937-C04 (FRESCO) and matching funds TIC2000-1970-CE, and CIRIT 1997SGR-00366.

1. Introduction

The research field of Knowledge Discovery in Databases (KDD) aims at developing methods, algorithms, and criteria for finding useful information in large masses of data. One of the most relevant tasks in KDD is the discovery of association rules, first formulated in [**?**]. This task is often referred to as the *"market-basket"* problem where we are given a (large) set of items and a (very large) set of transactions, each of which is a set of items. The "association rule" discovery task identifies the groups of items that most often appear along with other groups of items.

For this frequent itemsets problem, there is a rich variety of algorithmic proposals. Most of them, with the only exception of some recent alternative approaches [**?**], [**?**], [**?**] (which are based on computing projections of the input database), are based on traversing the lattice of itemsets in search of either all frequent sets or at least all maximal frequent sets. As analyzed in [**?**], the principal lattice-traversal algorithms can be roughly classified into two high-level frameworks: either they focus on counting the support of each itemset, usually through a breadth-first search, like Apriori [**?**] and its variants, or on finding frequent itemsets by intersecting lists of transactions containing each itemset, usually in a depth-first-like search, like Eclat [**?**] and its variants. However, there are other alternatives, and we propose here to study one of them, a best-first strategy in a well-defined sense. In our best-first strategy, the exploration deepens early on those itemsets that are easy to certify as candidates: a sort of adaptive search resembling the way efficient graph search algorithms (like branch-and-bound or shortest paths) work.

Currently our strategy does not seem to be truly competitive with fully different approaches like FPGrowth [**?**] or the TreeProjection algorithm of [**?**], which are based on projections of the input database; however, we continue to study these algorithms from our perspective.

2. Problem Formulation

The standard input to a frequent sets algorithm is a collection of *itemsets*, or *transactions*, each labeled by a unique identifier. Each transaction is a subset of a given, fixed set of items $\mathcal{I} = \{i_1, i_2, \ldots, i_N\}$; thus, if we forget about the identifiers, the input would be just a multiset of transactions that we identify as our input *database*. A k-itemset is an itemset consisting of k items.

The *support* of an itemset is the fraction of transactions in the input database that contain the itemset. An itemset is called *frequent* if its support exceeds a given threshold, σ. The frequent sets problem is: given the database and the threshold σ, find all frequent sets of items.

2.1. *Breadth-First Algorithms*

Many frequent sets algorithms traverse the lattice of all itemsets according to inclusion, by seeing it as a graph. A basic, well-known breadth-first algorithm for finding frequent itemsets is the Apriori algorithm [**?**]. It devotes one pass through the database to each itemset size k. Along each pass, it determines the support of all candidate itemsets of that size, and at the end of each pass, a candidate generation phase is performed.

A variation of Apriori is the Dynamic Itemset Counting (DIC) algorithm [**?**], which partially retains the breadth-first strategy but tries to jump one level up every M transactions, for a user-specified parameter M. Another important variant of Apriori is the Direct Hashing and Pruning algorithm DHP ([**?**], and it is not difficult to see that the strategies used in DHP can be easily applied to other algorithms as well.

2.2. *Depth-First Algorithms*

The itemset lattice can be explored also in a form close to depth-first search. Some algorithms of this sort were proposed in [**?**] and clear examples appear in [**?**]. It has been argued that counting supports is inefficient in depth-first search, and indeed most depth-first proposals construct instead, for each frequent itemset, the whole list of transactions in which it appears, walking up through the lattice by intersecting them, and complementing the procedure with various heuristics like fast intersections [**?**]. See also [**?**] for precise descriptions and analyses.

3. A Best-First Strategy

Our proposal is based on the idea of advancing as much as possible the start point for counting the support of a candidate itemset; so, frequent itemsets detected early will lead sooner to the exploration of larger itemsets. To count the support of a candidate, one needs to fully scan the database, but the order is irrelevant; thus, if we see the input database as "cyclic", we can start counting the support of an itemset at any arbitrary transaction, wrap up to the beginning upon reaching the end, and stop one line earlier than the start. The use of this property leads to the Dynamic Itemset Counting, which allows one to generate new candidates at specific moments of the database scan: one profits from the fact that we may know that an itemset is already frequent before finishing counting its support. Thus: our concept of "best" is precisely defined by how early the algorithm can realize that an itemset is, in fact, potentially frequent.

Besides, we fully avoid the time-consuming process of generating a new round of candidates from the supports found up to the point; instead, we keep track of how many immediate subitemsets of a given itemset have already been proved frequent. At the moment of receiving the last such confirmation, the itemset is declared potentially frequent, and its support is started to be counted right there without delay. The earlier we know that all subitemsets of an itemset are frequent, the earlier we start counting its support.

Note that small itemsets included in big frequent itemsets may easily tend to have a support substantially higher than the threshold; so this threshold can be reached with just a fraction of the input database. We take this as an indication that it is worth to try to climb up soon on those itemsets, aiming at handling soon a substantial part of the path towards big frequent itemsets.

To this end, whenever each itemset is proved frequent, we record this fact appropriately, in such a way that, for itemsets having all their subitemsets frequent, we know it literally as soon as possible, so as to start counting them immediately.

3.1. *Best-First Exploration Based On Apriori*

Let us describe more precisely this specific case of best-first: the algorithm, which we call Ready and Go (R&G), obtained by applying our strategy to the Apriori approach. Along the exploration, our algorithm maintains two collections of itemsets:

– The **candidate itemsets**, whose support is being counted in the database.

– The **borderline itemsets**, immediate supersets of some candidate itemset that we know is frequent, and might become frequent themselves.

An itemset enters the collection of borderline itemsets as soon as one of its immediate subsets has been found frequent. Each borderline itemset evolves in one of two forms: either the algorithm finishes counting an infrequent subitemset, and then it gets pruned as with the Apriori heuristic; or the algorithm eventually finds that all its subitemsets are frequent (possibly before finishing counting them!) and then the borderline itemset becomes a candidate and its support starts being counted.

The algorithm cycles through the database, reading a transaction at a time and processing it, and wrapping up to the beginning upon reaching the last transaction; for each transaction read, the data structure that maintains candidate itemsets is carefully updated.

A boderline itemset becomes an actual candidate whenever the algorithm is sure that it will not be pruned by the Apriori antimonotonicity property. The algorithm maintains a counter for each borderline itemset. Each frequent immediate subitemset will increment it, so that we always know how many of its immediate subitemsets have been declared already frequent. Hence, as soon as a candidate itemset of size $k-1$ becomes frequent, it increments the counter of its supersets of size k, and all such counters reaching the value k at that point indicate borderline itemsets that must immediately join the set of candidates. Support counting for them starts at that very same transaction: in this way, each individual candidate is generated as soon as possible, instead of waiting to a later candidate generation batch.

Of course, the algorithm must keep track of the transaction where counting started for each candidate, in order to know when the corresponding database pass is complete.

A more precise formal description of the algorithm is as follows:

1/ Inputs: database, support threshold σ.

2/ Initialize the candidate itemset collection with all 1-itemsets, their supports to zero, and the borderline itemsets collection with an empty set.

3/ Loop on the following:

4/ Read a new transaction from the database and increment the support of all candidates that appear in the transaction.

5/ For each candidate becoming frequent (its support reaches σ) do 6 through 8:

6/ Increment the subitemset counter of those borderline itemsets that are its immediate supersets.

7/ Add, as borderline itemsets, with subitemset counter initialized to 1, all immediate supersets that are not found among the current borderline itemsets.

8/ If the counter of a borderline itemset α of size k reaches value k, then remove α from the borderline itemsets collection and add α to the candidate itemset collection, initializing its support to 0 or 1 according to whether it is present in the current transaction, and storing the transaction identifier with it.

9/ If a candidate has been counted through all the transactions then:

10/ remove it from the candidate itemset collection, and

11/ if its support is at least σ, output it together with its support, else remove all its supersets from the set of borderline itemsets[1].

12/ If we are at the end of the transaction file, rewind to the beginning.

13/ Exit loop in case there are no candidates left.

For the comparisons, we tried to be fair in the sense that the same basic data structures were being used whenever possible. Thus, for comparing with DIC, both DIC and R&G maintained candidates in a prefix tree [**?**]; we avoided the design of sophisticated data structures for the borderline sets to be sure that any improvement was only due to the algorithm, and simply stored them in a hash table together with the relevant counters and start transactions.

1. Occassionally, it may happen that a removed borderline itemset is inserted again later on, but it will never become a candidate anyway; so this fact is harmless.

3.2. *Experiments and Comparisons*

It is interesting to compare our algorithm with DIC, specifically with respect to the parameter M that controls the generation of candidates. Setting it to the length of the database results in Apriori again. Setting it very small the candidate generation function gets called too often, and this creates an overhead that makes DIC less efficient than standard Apriori. In this context, Ready and Go can be seen as an intent of running DIC in the extreme case, that is, when the size of the interval M is 1, but, of course, without incurring in the overhead.

Testing was made on a PentiumII PC at 233 MHz with 256Mb of RAM, running Linux; the implementation was made in C++ and compiled by `gcc`. In order to clearly identify the effect of the best-first strategy, the data structures implemented are the ones suggested by [**?**]. We used data of an average size of 10 items chosen from 100 items from a syntethic database generator [2], and compared the differences in running time and transactions visited as a function of the variations of *mininum support* and *length of the database*.

For the first case, we used a 100000-transaction synthetic database. We tried R&G and DIC for supports varying from 0.3% to 2%[3]. DIC was run with a value of M equal a 10% of the database size. In absolutely all cases Ready and Go was faster by a modest but very stable factor close to 20 percent.

As a function of the length of the database, we ran DIC and R&G for fixed support thresholds on databases from 250000 to 1M transactions, at the same value of M. We give here the results for a threshold of 0.6%, but the results obtained for other values were similar. Again, as in the previous cases, the savings were of about 5% to 10% in number of transactions and 20% in running times.

We tuned DIC experimentally to what seemed its best value of M for each of the experiments (this used to be near $M = 1000$). Even in this case, R&G was better, although marginally; its advantage for this case being that there is no need to tune any parameter. Again the experiments correspond to decreasing support and increasing size as before. Tables 1 and 2 show the results.

	Apriori	**DIC**	**R&G**
2%	393045	250523	249630
1.5%	491306	254523	253695
1%	491306	256523	254676
0.8%	491306	266523	264007
0.6%	491306	259523	257950
0.3%	589467	279523	277867

Table 1. *Processed transactions as support decreases (optimal M)*

2. http://www.almaden.ibm.com/cs/quest/syndata.html
3. With synthetic data, support threshold has to be considerably low.

	Apriori	DIC	R&G
100000	491306	259523	257950
250000	1228421	651369	648332
500000	2457016	1301807	1299898
750000	3685726	1943291	1941000
1000000	4914741	2583897	2581855

Table 2. *Processed transactions as database size increases (optimal M)*