Data Min Knowl Disc (2014) 28:773–807
DOI 10.1007/s10618-013-0304-3

A time-efficient breadth-first level-wise lattice-traversal algorithm to discover rare itemsets

Luigi Troiano · Giacomo Scibelli

Received: 9 July 2011 / Accepted: 6 February 2013 / Published online: 12 May 2013

Abstract In this paper we face the problem of searching for rare itemsets. A main issue regards the strategy to adopt in exploring the power set lattice. Assuming a power set lattice with full set at the top and empty set at the bottom, the most of the algorithms adopt a bottom-up exploration, i.e. moving from smaller to larger sets. Although this approach is advantageous in the case of frequent itemsets, it might not be worth being used for rare itemsets, as they occur on the top of the lattice. We propose Rarity, a top-down breadth-first level-wise algorithm. Experimental results and comparisons are illustrated in order to provide a quantitative characterization of algorithm performances and complexity. Application to some UCI benchmark and real world datasets is provided. An algorithm parallelization is outlined. Experiments showed that this approach takes advantage of finding all rare non-zero itemsets in less time than other solutions, at expenses of higher memory demand.

Keywords Data mining methods and algorithms · Rare itemsets

1 Introduction

Knowledge discovery in databases aims at searching for interesting patterns concerning sets, collections, sequences and relations of data. The identification of target itemsets, i.e. subsets of occurring items, represents the starting point of analysis.

Responsible editor: M. J. Zaki.

L. Troiano (✉) · G. Scibelli
Department of Engineering, University of Sannio, 81031 Benevento, Italy
e-mail: luigi.troiano@unisannio.it; troiano@unisannio.it

G. Scibelli
e-mail: giacomo.scibelli@gmail.com

Although most frequent itemsets are surely appealing in searching for interesting and previously unknown facts (Agrawal et al. 1993a,b; Agrawal and Srikant 1994; Agrawal et al. 1996), less occurring itemsets are still of interest, especially in larger databases. Indeed, the most of the times the focus is on finding rules among frequent patterns, but it might be interesting to search for rare patterns, i.e. those that do not frequently occur in the database.

The search for less frequent collections of items can lead algorithms to identify relationships between items that, although supported by a minority of cases, can still reveal non-trivial, novel and valuable information. This is the case for patterns whose support is unexpectedly low or the appearance of items together in low support itemsets is unexpectedly high. Examples can be drawn in medical research, homeland security, basket analysis and other applications. As an example, let us consider data concerning passenger behavior in airports. Frequent patterns are due to normal behavior, whereas rare patterns could identify suspicious and still unknown threats. As another example, searching for rare patterns of adverse drug effects in pharmacovigilance can help to identify cases where drugs have serious or fatal consequences. Other examples deal with discovery of rare DNA gene co-occurrences, or infrequent product baskets able to reveal unexpected and profitable market niches.

Both the search for frequent or rare patterns are NP-hard (Yang 2004). Until now few algorithms and techniques have been developed for mining rare itemsets, that are those patterns whose support is below a given threshold. Algorithms designed to mine frequent itemsets are not suitable for mining rare patterns. Indeed the search for rarities entails different but still relevant difficulties for data mining algorithms (Weiss 2004). There is a need for developing specific algorithms more than adapting existing strategies. In this paper we investigate a breadth-first level-wise lattice-traversal algorithm, Rarity, able to mine rare patterns performing a time-efficient exploration of the search space and support computation, moving from largest to smaller itemsets. Therefore if we assume an orientation of power set lattice such that the full set is at the top and the empty set at the bottom, lattice traversal per formed by Rarity can be described as top-down. This convention will be followed in the remainder.

This paper is a revised and extended version of results presented in Troiano et al. (2009). The main contribution is to investigate properties and performances of this algorithm and to suggest enhancements to scale problem complexity. This approach makes use of a procedure for computing the support count that takes advantage of the regularity in the itemset lattice structure. Indeed itemsets at upper levels in the lattice contribute to the support count of their sub-itemsets. For each itemset, the different contribution to the support count is held by a vector whose entries keep memory of how many times transactions of the corresponding length comprise the associated itemset. Support count is computed by summing up the vector entries divided by the number of times such a contribution has been considered. Computing the support count aims at discarding those itemsets of length $k - 1$, generated by itemsets of length k, occurring over the given threshold. Discarded itemsets are kept in a veto list in order to further discard the sub-itemsets they entail. The proposed method is experimented against synthetically generated datasets in order to assess the algorithm behavior under different circumstances, and against some benchmark datasets from UCI repository with different characteristics. We also provide three applications as motivating examples. As the lattice traversal is memory demanding, issues regarding problem decomposition and parallelization have to be considered. This paper mainly focuses on explaining the algorithm logic, although considerations regarding effective implementation on real datasets are considered. Pre-filtering is suggested in order to discard those itemsets which are spurious. Parallelization is only outlined as mostly based on consolidated techniques.

The remainder is organized as follows: Sect. 2 illustrates three motivating examples; Sect. 3 provides some preliminary definitions and properties of rare itemsets; Sect. 4 overviews related works; Sect. 5 describes the algorithm; Sect. 6 presents experimental results; Sect. 7 suggests algorithm enhancements in order to address some issues; Sect. 8 illustrates experimental results; Sect. 9 estimates algorithm complexity; Sect. 10 suggests a possible algorithm parallelization; Sect. 11 shows possible solutions to problems stated in Sects. 2; 12 outlines conclusions.

2 Motivating examples

The mining of rare itemsets could be limited to classes which would include potential itemsets never occurring in data (zero-itemsets) and those that are sporadic (occurring a number of times below a minimal threshold). Both cases are not of interest in many circumstances, and testing occurrences of all potential itemsets belonging to these classes only adds complexity to the mining process, since all possible combinations should be taken into account and explored.

In this section we state three problems, where the list of occurring non-sporadic rare itemsets would be of interest. These problems do not mean to be exhaustive, but only to motivate and suggest when and why the problem of listing rare itemsets could arise in similar situations. Solution to following examples is given in Sect. 11.

2.1 Spare parts

For machine maintenance, each repair involves a set of spare parts to change. In production plants, it becomes critical to have spare parts stocked, so that when a stop occurs, plant can be repaired and restarted as soon as possible, without any delay due to parts provisioning. So that stocks should be managed in order to assure a given level of quality of service (QoS). Keeping stocks of spare parts can be expensive, but being ready to repair unexpected and unusual stops can be relevant in managing maintenance inventory. Therefore identifying non-frequent (rare) groups of spare parts can help to better manage spare part stocks.

The data source is the maintenance log. Each record provides the list of spare parts (items) used to repair a stop. Besides parts that are generally occurring in repairs, infrequent groups of spare parts can provide useful information. A group can implement a machine function, so that parts belonging to that group assume a specific meaning. Mining those groups can help to identify critical stops and to level the stocks of parts involved, in order to face future stops.

2.2 Unit test

The problem of testing software units is known to be infeasible due to the large number of paths that can be activated by all potential inputs to the unit. This problem is particular relevant for units implementing data analysis or automatic control algorithms. In this case, it would worth to find groups of values that although rare, could be able to lead to possible lacks in the unit code.

In this case, the source is given by the log of values (the items) being passed to the unit as arguments at each invocation. If the number and type of arguments is kept constant, the log structure is regular, and each column represent the sequence of values to pass to the unit as corresponding argument. Looking for rare subgroups of values can help to identify conditions to test, that although infrequent could lead to unit failures.

2.3 Text analysis

The focus of text analysis, in particular text categorization (or classification), is on sub-sequences of words able to characterize documents. Therefore sequences should be not too frequent (common terms and phrases) and not too sporadic (unrelated terms and phrases). In this case, mining rare itemsets would help to identify those patterns worth of being analysed. For example, those patterns that can help document indexing.

Data sources used by this example are collections of (stemmed) sets of terms (the items). Mining can be addressed to identify those groups of terms specific for a subset of documents. Those groups should not be too infrequent to be considered sporadic and unrelated, therefore a minimum support threshold should be fixed accordingly.

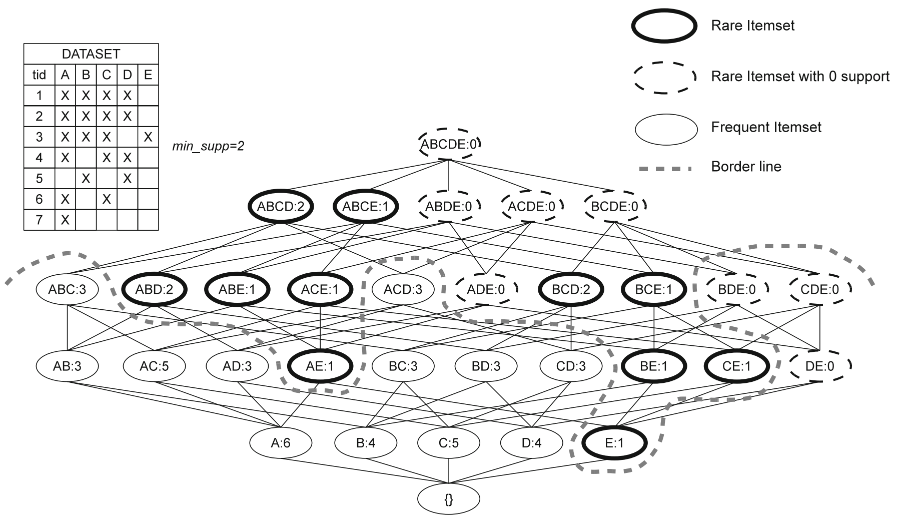
3 Preliminaries

A group or set of items entailed by database records, e.g. the set of items a customer collects in a market basket, is referred to as an *itemset*. More formally, let $I = \{i_1, i_2, ..., i_m\}$ be a set of m distinct literals called *items*. Let the database $\mathcal{D}$ be a collection of transactions over I. Each transaction T is associated to a unique identifier *TID* and contains a subset of items $X \equiv \{i_i, i_j, ..., i_k\} \subseteq I$. The number of items in X provides its *length*. Transactions in $\mathcal{D}$ can entail different itemsets of different length k, referred to as k-itemsets. The number of times an itemset X occurs in transactions is the *support count* of X (Agrawal et al. 1993a,b), denoted by $supp(X)$.[1]

Frequent and rare itemsets are defined with respect to support count threshold t_f and t_r, with $t_r < t_f$. In particular, itemsets are said:

– *frequent*, iff $supp(X) \geq t_f$
– *rare*, iff $supp(X) \leq t_r$

[1] Terms "support" and "support count" assume different meaning in data mining. In the context of this work, they refer to how frequent an itemset is. When this is computed in terms of number of occurrences, it is more appropriate the use of "support count". More in general, we will refer to "support" as expression of itemset occurence.



DATASET					
tid	A	B	C	D	E
1	X	X	X	X	
2	X	X	X	X	
3	X	X	X		X
4	X		X	X	
5		X		X	
6	X		X		
7	X				

Fig. 1 A dataset and corresponding itemset lattice

The itemset's support count is related to the support count of its supersets. Indeed, given two itemsets X and Y such that $X \subset Y$, the number of occurrences of X is at least the number of occurrences of Y, as X is the part of Y. Therefore,

$$supp(X) \geq supp(Y), \ \forall X \subset Y \tag{1}$$

It is useful to make a distinction between *non-zero itemsets* that are actually in $\mathcal{D}$, from *zero itemsets* that are not having any occurrence in any transaction of $\mathcal{D}$ (Szathmary et al. 2007).

As proven in Yang (2004) the problem of counting the number of distinct itemsets in a dataset, given an arbitrary support threshold, is NP-complete and the problem of mining itemsets is NP-hard. If $m = |I|$ is the cardinality of the item collection I, the number of possible distinct itemsets is 2^m. In order to reduce the combinatorial complexity of search space, most algorithms exploit the following two properties:

- *Downward closure*: all subsets of a frequent itemset are frequent;[2]
- *Anti-monotonocity*: all supersets of a rare itemsets are infrequent.

Complexity comes out from the need of traversing the item power set lattice P. An example of item power set lattice is given by Fig. 1 presented in Sect. 4. In the example we define $t_f = 3$ and $t_r = t_f - 1 = 2$, so that each infrequent itemset is also a rare itemset. The support count is reported in the top right-hand corner of itemsets. The set of all rare itemsets forms a join-semilattice since, as we said, it is closed under the join operation, i.e., for any rare itemsets X and Y, $X \cup Y$ is rare as well. On the other hand, it does not form a meet-semilattice, since if X and Y are rare, it does not entail $X \cap Y$

[2] The name of the property comes from the fact that the set of frequent itemsets is closed with respect to set inclusion.

is rare. Note that frequent itemsets form a meet-semilattice, i.e. for any two frequent itemsets, X and Y, $X \cap Y$ is also frequent. In drawing the structure of the power set lattice, we follow the convention of placing the full set at the top and the empty set at the bottom, as depicted in Fig. 1. Between frequent and rare itemsets a border can be drawn. In Fig. 1 such a border is denoted by a dashed line. The border emphasizes that rare itemsets have place on the top of the lattice whereas frequent itemsets populate the bottom. At the bottom we find the smallest common itemset, i.e. the empty set $\emptyset$, whereas at the top of the lattice we find the largest one, i.e. I. At each level there are itemsets of the same length.

4 Related work

Traditionally, frequent patterns mining is associated to the discovery of association rules (Agrawal et al. 1993b; Piatetsky-Shapiro and Frawley 1991) as they provide a subset of patterns which to search among. Techniques are different if they address the search for frequent itemsets, or rare itemsets. Whereas the problem of mining frequent itemsets has been largely investigated and many solutions have been proposed, research on the problem of mining rare itemset is still on progress, although some algorithms have been proposed. They differ by the representation of data, the strategy of traversing the itemset lattice, and the way of counting occurrences. Although the topic investigated in this paper is related to the search for rare itemsets, an overview of techniques used in mining frequent itemsets is still of interest in order to compare the different approaches. The following subsections provide a review of main contributions to mining both frequent and rare itemsets.

4.1 Frequent itemsets mining

The first and most noticeable algorithm for mining frequent itemsets is *Apriori*, proposed independently by Agrawal and Srikant (1994) and by Mannila et al. (1994), and later joined in Agrawal et al. (1996). Apriori is a level-wise, breadth-first, bottom-up algorithm. The algorithm starts by considering frequent 1-itemsets collecting them in F_1. From these, candidate 2-itemsets, are generated by joining frequent 1-itemsets. Their support is computed by searching for their occurrences in the database and those below the given threshold are discarded. The process is iterated, so that at step k, candidates are generated from frequent itemsets in F_{k-1} and filtered out in F_k according to their support. The algorithm stops when F_k is empty or k is the size of the longest record, that is the maximum length of itemsets. Frequent itemsets are obtained by merging collections $F_1, \ldots, F_k$. As levelwise algorithm, Apriori is based on downward closure and anti-monotonicity property. The algorithm structure entails to pass through the database at each step, thus affecting performances. Performance issues regarding time and memory have been further addressed by Savasere et al. (1995) and Brin et al. (1997).

Pasquier et al. (1999) proposed *Apriori-Close*, an extension of Apriori which is able to find closed frequent itemsets.[3] In this solution, at step k all frequent itemsets are marked as *closed*. At step $k+1$, frequent itemsets in F_{k+1} are compared to sub-itemsets in F_k by support, in order to establish if the latter are really closed or not. The algorithm ends-up with the enumeration of all frequent itemsets, some of them keeping the label as closed.

Bastide et al. (2000) proposed an algorithm named *Pascal*, which makes use of a strategy known as *pattern counting inference*. The authors introduce the notion of key patterns (also called generators [4]) and show that other frequent patterns can be inferred from them without accessing the database. In particular, their algorithm is based on the concept of *equivalence class* which collects itemsets appearing in the same subset of transactions, thus having the same support. Therefore, using a level-wise traversal, first the shortest frequent itemsets of an equivalence class are discovered. From those, new candidates are generated and their support is computed. When a candidate belongs to an already known class, support is assigned by the class avoiding to further access the database. The algorithm is able to find both frequent and closed itemsets, resulting to be faster than Apriori.

Szathmary et al. (2007) introduced *Zart*, as variant of Pascal. It is a levelwise algorithm that first enumerates generators of an equivalence class, and after filters frequent closed itemsets exploiting the pattern counting inference.

Zaki et al. (1997) presented *Eclat*, an algorithm designed to find frequent itemsets by a depth-first strategy. It makes use of columnar database, i.e. with vertical representation of data, counting the itemset's support by co-occurence of transaction IDs. An evolution of Eclat, called *dEclat*, is presented in Zaki and Gouda (2003). It introduces the concept of *diffset*, that is the transaction subset, in which a k-itemset does not appear, but its prefix $(k-1)$-itemset does. Therefore, support is computed by subtracting the cardinality of diffset from the support of its prefix.

Han et al. (2004) introduced *FP-growth*, a novel approach based on *frequent pattern tree* (FP-tree), that is an extended prefix-tree structure for storing compressed information about frequent itemsets. This solution allows to pack the representation of database transactions, to avoid the generation of candidate itemsets and to employ a partitioning-based method able to decompose the mining task into a set of smaller subtasks. A variant of FP-growth, *H-mine* (Pei et al. 2001), makes use of array-based and tree-based data structures to deal with sparse and dense datasets respectively.

Uno et al. (2003, 2004, 2005) proposed *LCM*, a series of algorithms for enumerating frequent itemsets in linear time. LCM was proposed for enumerating closed itemsets and then LCMfreq and LCMmax were presented for mining respectively all frequent and maximal itemsets. LCM integrates accelerated techniques and belongs to the category of backtracking algorithms, thus it inputs a frequent itemset I and then generates new itemsets adding one of the unused itemsets to I. To avoid duplications the algorithms adds items with indices larger than the tail of I.

[3] An itemsets X is said to be *closed* iff it is the largest subset of items in common to transactions in which X appears (Zaki et al. 1997).

[4] A generator is a frequent itemset with none of its proper subsets having its same support.

Song and Rajasekaran (2006) suggested to use a *transaction mapping* (TM) algorithm. The basic idea is to map and compress transaction IDs of each itemset to continuous transaction intervals in a different space. Thus, the count of itemsets is performed by intersecting intervals in a depth-first order along the lexicographic tree.

4.2 Rare itemsets mining

Most algorithms designed for mining rare itemsets have been proposed as variant of Apriori algorithm, and focused on the search for association rules. In particular, they investigate the need of including those itemsets that, although infrequent, provide high confidence rules.

In order to address this problem, Liu et al. (1999) proposed *MSapriori*. In their approach, they use multiple minimum support thresholds, assigned to each item in the database, and minimum support of a rule is expressed in terms of minimum support (*MIS*) of the items that appear in the rule. This way, the user expresses different support requirements for different rules. Let $MIS(i)$ denote the *MIS* value of item i. The minimum support of a rule is the lowest MIS value among the items in the rule. So it is possible to change the support threshold of rules dynamically in order to use higher thresholds for those rules that involve frequent items, and lower thresholds for rules involving less frequent items. This change is driven by parameter β expressed by the user making the algorithm sensitive to the user preferences.

A solution that attempts to avoid this issue is Relative Support and Apriori Algorithm (*RSAA*), presented by Yun et al. (2003). This technique makes use of relative support defined as the ratio between the itemset's support and the MIS of items belonging to it. Therefore the support threshold for items having low frequency is automatically increased, while the support threshold for items having high frequency is automatically decreased. Similarly to Apriori and MSapriori, RSAA is exhaustive in generating rules, focusing on those which are not spurious. If the minimum-allowable relative support value is close to zero, then RSAA takes a similar amount of time to Apriori, when generating low-support rules.

Both RSAA and MSapriori are direct adaption of Apriori algorithm to less frequent itemsets, so that they search for itemsets whose support is over a given (adaptive) threshold. Koh and Rountree (2005) proposed *Apriori-Inverse*, a levelwise bottom-up variation of Apriori that makes use of the notion of maximum support instead of MIS to generate candidate itemsets. In particular, it makes use of two thresholds: max_supp, to find rare itemsets and min_supp, to discard those are considered to be extremely rare. Apriori-Inverse first identifies rare 1-itemsets collecting them in in R_1. Similarly to Apriori, it generates longer itemsets which will be rare themselves and new generators of the same length are added. The process is iterated until the maximum length k is reached. The union of $R_1, \ldots, R_k$ provides perfect rare itemsets, which are itemsets that only consist of items below the maximum support threshold. Rules are generated in the same way done by Apriori. However, Apriori-Inverse is not able to capture all the candidate itemsets, but only the perfectly rare itemsets. There could be itemsets that are still infrequent, but whose items are not rare by themselves. Slight modifications have been proposed in order to extend the result provided by Apriori-

Inverse. Koh et al. (2008) proposed *MIISR*, to find rules with a single-item consequent below the max_supp. This condition ensures the whole itemset is below the given threshold, but there are still rare itemsets that does not fall in this class.

Szathmary et al. (2007) proposed a solution called *ARIMA* which is based on a two-step algorithm, able to search for all rare itemsets. The first step consists in traversing the frequent zone in the lattice by levels in order to find minimal rare itemsets which are those rare itemsets whose proper subsets are frequent; after that, the other rare itemsets are generated as supersets. Indeed, in order to find rare itemsets it is sufficient to identify the minimal rare generators and related supersets as they will be surely rare. If rare zero itemsets are not of interest, the algorithm can prune the branch with a stop when a minimal zero generator is found. The strategy aims at identifying the border line which separates the frequent itemsets from the rare, starting from the bottom of lattice and moving upwards in order to reach the limit. All itemsets above that limit belong to the class of rare itemsets. The search for minimal rare itemsets can be performed by two algorithms. The first is (*Apriori-Rare*), that is still a variant of Apriori and consists in traversing the lattice, and collecting at each level the itemsets usually discarded by the original algorithm. A better approach is provided by *MRG-Exp*, which focuses exclusively on frequent generators and their downset in the lattice filtering minimal rare generators at the same time.

Traversing the frequent itemsets can be improved. For instance, Haglin and Manning (2007) proposed *MINIT*, an algorithm aimed at discovering minimal infrequent itemsets, i.e. those itemsets whose proper subsets are frequent. The approach is based on a recursive procedure which explores the lattice by fixing items in turn. At each step, focus is restricted on transactions containing items fixed so far, and remaining items are sorted by occurrence is such transactions, so that most frequent items come first. A set of properties allows to cut the search when a minimal infrequent itemset is found. Since minimal infrequent itemsets are the first rare itemsets met by traversing the lattice bottom-up, they can be used as starting point to search the rare itemsets.

Recently, Tsang et al. (2011) proposed *RP-Tree*, an algorithm for mining a subset of rare association rules based on a tree structure. The authors make difference between three types of rare itremsets: those which consist of rare items only; those which consist of both rare and frequent items; those which consist of only frequent items. The first and second types are referred as *rare-item itemsets*, the third as *non-rare-item itemsets*. The algorithm uses this classification to perform a mining strategy similar to FP-growth. In particular it performs a preliminary database scan in order to count items' support, and a second scan in order to map a tree on the itemset lattice pruning transactions without at least one rare item. Using this initial tree, the algorithm constructs conditional pattern bases and conditional tree for each rare item. Finally FP-growth is applied to each conditional tree in order to get a subset of rare itemsets, in particular those in which at least one item (i.e. singleton) is rare, the so called *rare-item itemsets*.

A different strategy consists in traversing the lattice top-down. Adda et al. (2007) proposed an Apriori-like approach, called *AfRIM*, aimed at exploiting anti-monotone property and exploring level-wise the lattice starting from the itemset composed of all items in I, that is the top of the itemset lattice. If this approach assures the exploration of the whole search space on one side, it is not suitable for larger datasets and search spaces, generally characterized by sparser distribution of items, on the other side. Indeed, the strategy proposed by Adda et al. forces the exploration of zero itemsets. Their experimentation is restricted to generated datasets with a limited number of items (up to 10) and dense with transactions made of at most 8 items.

5 Algorithm

Except for AfRIM, algorithms presented above performs a traversal of itemset lattice from bottom to top, thus exploring the frequent zone first. So that, the smaller is the threshold t_r, the wider is the zone to explore. Instead, it would be worth exploring the lattice from top, where rare itemsets are placed. Rarity is a top-down breadth-first level-wise algorithm which exploits this feature of the search space. It starts by identifying the longest rare itemsets in the database, and moves downwards the power set lattice discarding those itemsets which result frequent, and developing only those that are confirmed to be rare. Indeed a frequent itemset entails sub-itemsets that are necessarily frequent, whereas rare itemsets can provide sub-itemsets that are possibly but not necessarily rare. Different from other approaches, e.g. ARIMA (Szathmary et al. 2007) and AfRIM (Adda et al. 2007), starting from the longest record in the database and not from the longest itemset of the search space, allows Rarity to avoid the generation of rare zero-itemsets and the scan of the database in order to compute the support of itemsets. This is a major advantage for sparse databases, especially when the number of items largely exceed the length of transactions.

In order to implement its search strategy, Rarity requires three data structures: the candidate list C, the veto list V, and the rare itemset list R.[5] The candidate list collects itemsets that are possibly rare. The veto list contains itemsets known to be frequent. Both lists are organized by levels. Therefore $C(l)$ and $V(l)$ refer only to l-itemsets. An additional list R contains rare itemsets resulting from processing. Algorithm 1 outlines the algorithm's logic.

The algorithm initializes the candidate list $C(l)$ with candidate rare l-itemsets by passing through the database to count l-itemset's support. Initially, V (the veto list) is empty and the algorithm begins from the top of lattice by considering the longest itemsets. More specifically, if $l_m = argmax C(l) | C(l) \neq \emptyset$, Rarity starts by considering candidates in $C(l_m)$. After, for each $l = l_m..1$, the algorithm moves down processing candidate itemsets $is \in C(l)$. When $supp(is)$ is greater than threshold $t_r = min\_supp$, it is frequent and is moved to veto list $V(l)$. Differently, itemset is is rare and moved to the rare list R. In addition, its sub-itemsets long $l - 1$ are possibly rare, then assigned to $C(l - 1)$. The next step consists in scanning $V(l)$ so that each known frequent itemset $f_j \in V(l)$ is compared to shorter candidates $g_k \in C(h)$ with $h < l$. Intersection $e_{jk} = f_j \cap g_k$ is used to find sub-itemsets in common. Indeed, since e_{jk} is known to be frequent as part of f_j, it is moved (or added if never considered so far) to the veto list $V(l_{jk})$ where $l_{jk} = len(e_{jk})$. The ultimate level considered by Rarity is $l = 1$ made of 1-itemsets. However, an early stop happens when there

[5] They are actually sets, as they do not admit duplicates, but for the sake of simplicity, we refer to them as lists.

Algorithm 1 Rarity

```
Input: $\mathcal{D}$ - Dataset
Output: $R$ - Rare itemset collection
1: $l_m = \max len(t)\ \forall t \in \mathcal{D}$
2: for all record $t \in \mathcal{D}$ do
3:   add $t$ to $C(len(t))$
4: end for
5: for $l = l_m..1$ do
6:   if $C(l) \neq \emptyset$ then
7:     for all $is \in C(l)$ do
8:       if $supp(is) > min\_supp$ then
9:         remove $is$ from $C(l)$
10:        add $is$ to $V(l)$
11:      else
12:        add $is$ to $R(l)$
13:        if $len(is) > 1$ then
14:          for all $sub \in subsets(is)|len(sub) = l - 1$ do
15:            if $sub \notin V$ then
16:              add $sub$ to $C(len(sub))$
17:              $\mathbf{v}_{sub} = \mathbf{v}_{sub} + \mathbf{v}_{is}$
18:            end if
19:          end for
20:        end if
21:      end if
22:    end for
23:    for all $is \in V(l)$ do
24:      if $len(is) > 1$ then
25:        for $k = l - 1..1$ do
26:          for all $c \in C(k)$ do
27:            $cis = c \bigcap is$
28:            remove $cis$ from $C(len(cis))$, if $cis \in C$
29:            add $cis$ to $V(len(cis))$
30:          end for
31:        end for
32:      end if
33:    end for
34:  end if
35: end for
```

is no more candidate to consider at lower levels, that is $C(l) = \emptyset$. Indeed if a level contains only frequent itemset, it is not possible to further generate candidates and the algorithm stops.

Rarity performs an efficient computation of support based on support vector $\mathbf{v}_{is}$, which provides contributions to support count of is arriving from supersets at different levels. This vector is defined as

$$\mathbf{v}_{is} = \mathbf{v}_{is}^{0} + \sum_{i \in P(is)} \mathbf{v}_i \tag{2}$$

with $P(is)$ being the collection of is' parents, that are super-itemsets at one level over, and $\mathbf{v}_{is}^{0}$ being a vector expressing the number of times is occurs as record in $\mathcal{D}$, that is $\mathbf{v}_{is}^{0}(h) = 0$ for any entry $h \neq len(is)$. When a candidate is evaluated, all contributions to its support are available in $\mathbf{v}$, and support count can be computed from $\mathbf{v}_{is}$ as stated by the following proposition

Proposition 1

$$supp(is) = \sum_{h=l_{is}}^{l_m} \frac{\mathbf{v}_{is}(h)}{(h - l_{is})!} \tag{3}$$

where $l_{is} = len(is)$ provides the is's length and l_m the maximal itemset's length.

Proof According to Eq. (2), vector $\mathbf{v}_{is}$ takes into the account the contribution to is's support count provided by super-itemsets at different levels. Such a contribution arrives to is by different paths on the lattice. In particular, $\mathbf{v}_{is}(h)$, $h = l_{is}$ holds the number of times is occur in $\mathcal{D}$ as record. Itemsets below l_{is} are not of interest for is and over l_m do not stand in $\mathcal{D}$. Therefore, computation expressed by Eq. (3) can be restricted at levels $h = l_{is}..l_m$.

Since $\mathbf{v}_{is}$ is computed recursively, contributions from upper levels are summed up more than once. The number of times a contribution is considered by Eq. (2) depends on the difference between levels. This number is known in advance to be $(h - l_{is})!$, that is the number of paths joining a super-itemset at level h to is, as depicted in Fig. 4. $\square$

Thanks to Proposition 1, the algorithm is able to pass through the database only once at initialization time. This feature comes with other optimizations in order to improve overall performances. For instance, if at level l the candidate list $C(l)$ is empty, the algorithm skips the whole step and no scan of the veto list $V(l)$ is performed. Indeed, if there is no candidate $c_i \in C(l)$, the itemsets in $V(l)$ have been necessarily obtained by intersection with candidates longer than l, thus attaining a previous level. This also entails that only new itemsets determined to be frequent at current level are those able to veto candidates subsets. This makes it possible to avoid intersections that would be necessary required by elements already in the veto list. Finally itemsets are represented through compressed bitmaps, in order to reduce memory requirements and to perform fast binary operations such as intersection and union.

The algorithm execution on dataset $\mathcal{D}$ (see Fig. 1) with $min\_supp = 2$ is illustrated by Fig. 2. The algorithm first performs the database scan in order to set up its structures (C, V, R). In particular C is filled with the itemsets stored in the dataset. After that, Rarity starts by exploring each level l. Since there is no 5-itemset in the database, $C(5)$ is empty, and Rarity begins to analyze the level 4. It computes the support count of the two largest itemsets $ABCD$ and $ABCE$ using Eq. (3) and since it is less than the threshold min_supp the itemsets are moved to R and subsets are generated and added to C. The same has place at level 3. The elements in $C(3)$ are analyzed and two of them (ABC and ACD) are moved to V. This leads Rarity to process the itemsets in $V(3)$ in order to inhibit forthcoming candidates. So, after the process of V, itemsets AB, AD, BC, CD, AC, A, B, C and D (which are subsets of ABC and ACD) are moved to V.

Figure 3 explains how the support vector $\mathbf{v}_{is}$ is computed. This example is referred to the value of $\mathbf{v}_{AC}$. At the initialization time $\mathbf{v}_{AC}$ is $\mathbf{v}^0_{AC} \equiv (0, 0, 1, 0, 0, 0)$, entailing that

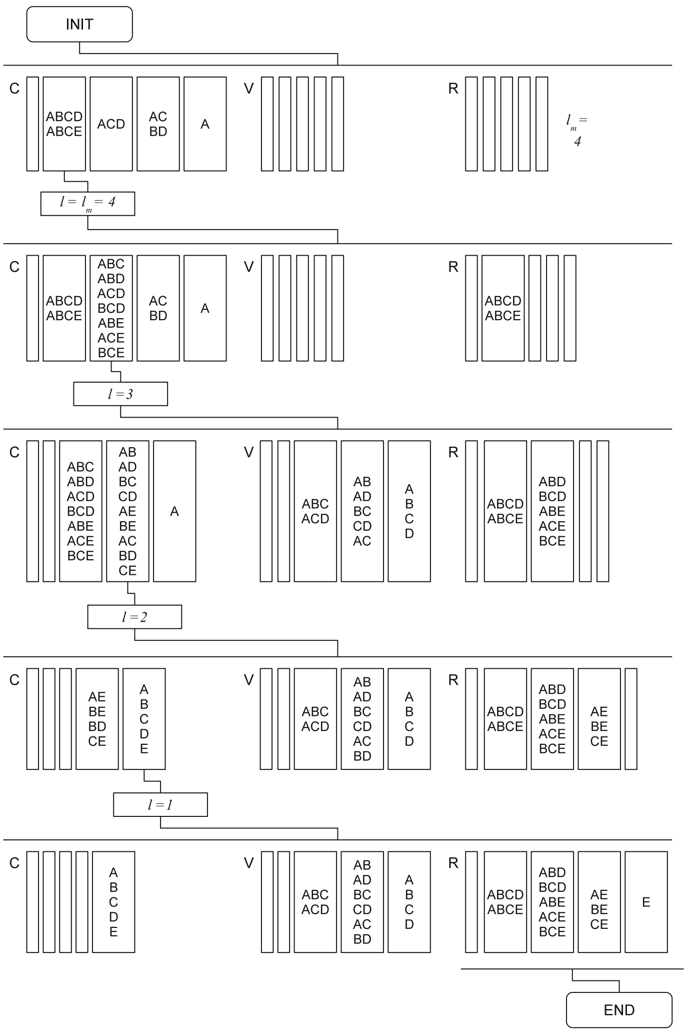
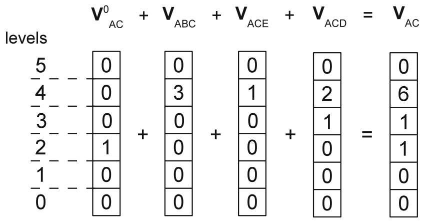
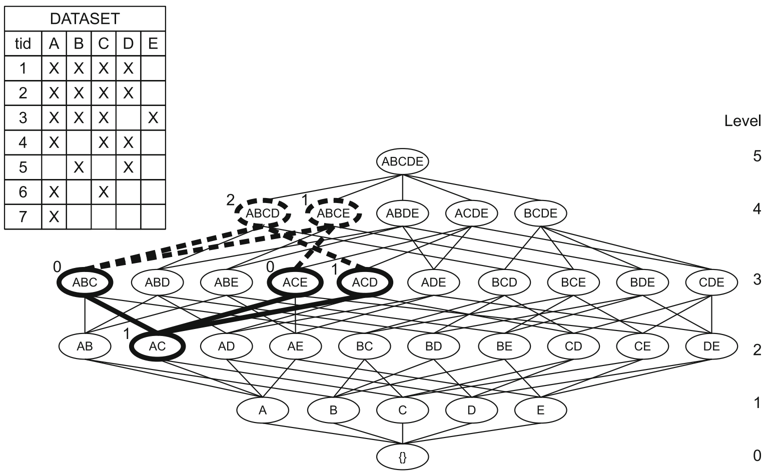


Fig. 2 Execution of Rarity

Fig. 3 Computation of vector **v** and support count


$\mathbf{V}^0_{AC}$ + $\mathbf{V}_{ABC}$ + $\mathbf{V}_{ACE}$ + $\mathbf{V}_{ACD}$ = $\mathbf{V}_{AC}$
levels
5
4
3
2
1
0
0 0 0 1 0 0
0 3 0 0 0 0
0 1 0 0 0 0
0 2 1 0 0 0
0 6 1 1 0 0
+
+
+
=

$$supp(AC) = \sum_{h=2}^{4} \frac{v_{AC}(h)}{(h-2)!} = \frac{6}{2!} + \frac{1}{1!} + \frac{1}{0!} = 5$$



DATASET					
tid	A	B	C	D	E
1	X	X	X	X	
2	X	X	X	X	
3	X	X	X		X
4	X		X	X	
5		X		X	
6	X		X		
7	X				

Fig. 4 An example of (indirect) contribution to support count

only one occurrence in $\mathcal{D}$ is contributing to AC (level 2) support count. At the same time we have $\mathbf{v}^0_{ABCD} \equiv (0, 0, 0, 0, 2, 0)$ and $\mathbf{v}^0_{ABCE} \equiv (0, 0, 0, 0, 1, 0)$ that is the number of time $ABCD$ and $ABCE$ respectively occur in $\mathcal{D}$. Moving to level 3, this latter contribution is given to both itemsets ABC and ACE. So that $\mathbf{v}_{ABC} \equiv (0, 0, 0, 0, 3, 0)$ and $\mathbf{v}_{ACE} \equiv (0, 0, 0, 0, 1, 0)$, in order to remember this contribution is provided by an upper level. Also ACD occurs once in the dataset providing contribution to AC support count. So contributions are given by different paths whose number depends on the level difference between the longer (l) and the shorter itemset (h). Therefore, contribution is divided by $(l - h)!$ when support vector are summed up, according to Eq. (2). This is depicted by Fig. 4, where itemsets at upper levels ($ABCD$ and $ABCE$) contribute to AC's support count by different paths (dashed lines).

When the algorithm reaches level 2, itemsets in $C(2)$ are all rare except BD, the only moved to V. Scanning $V(2)$ the algorithm inhibits singletons A, B, C, D. The final step consists in analyzind $C(1)$. E is the only itemset in $C(1)$ and since its support count is lower than min_supp it is moved to R.

Figure 5 provides details on level 3 processing. As the support of all itemsets in $C(3)$ is evaluated, frequent itemsets are moved to the veto list $V(3)$, while the others are moved to R. Next, the algorithm generates the candidate for level 2 which are subsets of rare itemsets at level 3, the result of intersection between $C(2)$ and $V(3)$ is the list of frequent itemsets, thus moved from $C(2)$ to $V(2)$. In our example these itemsets are AE, BE, BD and CE.

6 Experimentation

We compared Rarity to ARIMA (Szathmary et al. 2007) and AfRIM (Adda et al. 2007), as both algorithms are able to provide the full list of rare itemsets. In order to get a fair comparison, Rarity have been implemented in Java as original implementation of

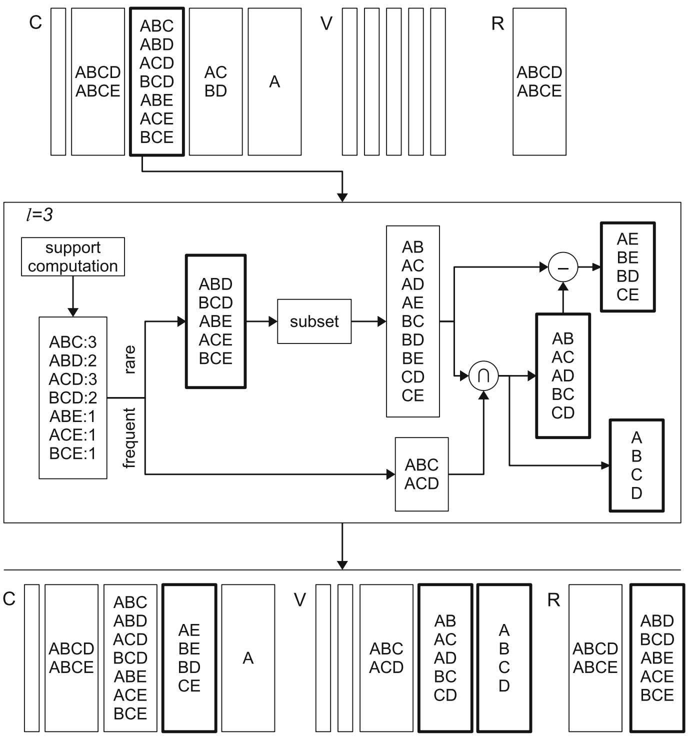


Fig. 5 Algorithm processing at $l = 3$

ARIMA provided by authors. We also developed an implementation of AfRIM based on the pseudo-code provided in Adda et al. (2007).

Codes have been instrumented in order to collect execution time and memory usage besides the data loading, in order to avoid side effects in measurements.

Experimentation was performed on Intel Xeon 2.66 GHz, with 4 GB of RAM, equipped with Windows Server 2003 Enterprise Edition Service Pack 2.

We performed a preliminary test on four different sparse datasets made of 10, 000 records respectively with 5, 10, 15 and 20 different items. From results outlined in Table 1 we observed that AfRIM was not able to highlight a time-efficient computation with respect to performances provided by Rarity and ARIMA except for datasets with a very small number of different items. In particular, as the number of items $|I|$ increases, execution time exceeds the limit to perform a test in a reasonable time. In addition, for larger datasets AfRIM was not able to complete the computation, so it will not be further considered in our comparison.

As test cases, we randomly generated 200 datasets differing by the number of rows and the maximum number of items in a transaction. In particular we generated datasets with a number of rows ranging from 1,000 to 10,000 and with maximum transaction length still of 5, 10, 15, and 20 items, each configuration considered 5 times. For each of those databases we have at most 20 different items. This entails at most 2^{20} itemsets to be analyzed. For the support count limit we assumed different values of

Table 1 Preliminaries Experiments

Algorithm	Support (%)	Supp. count limit	Rare itemsets	Time (ms)	Memory (MB)
Dataset 1–5 items					
Rarity	10	1,000	6	15	520
AfRIM	10	1,000	6	106	12
ARIMA	10	1,000	6	188	530
Dataset 2–10 items					
Rarity	20	2,000	968	46	510
AfRIM	20	2,000	968	596	13
ARIMA	20	2,000	968	1,018	531
Dataset 3–15 items					
Rarity	15	1,500	32,551	875	515
AfRIM	15	1,500	32,551	20,406	16
ARIMA	15	1,500	32,551	10,312	512
Dataset 4–20 items					
Rarity	20	2,000	954,694	35,532	93
AfRIM	20	2,000	954,694	10,113,320	115
ARIMA	20	2,000	954,694	752,619	500

$t_s = min\_supp$: specifically 0.1, 0.5, 1.0, 5.0, 10.0 % of the dataset size and 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 60, 70, 80 occurrences.

Figure 6 compares the execution time of ARIMA and Rarity. The comparison about memory usage is outlined in Fig. 7. All figures highlight Rarity to be faster at cost of higher memory demand, while ARIMA is slower with a smaller memory footprint. This result becomes more evident by plotting time versus memory as depicted in Fig. 8. Instead, Fig. 9 provides the speed ratio of Rarity versus ARIMA at varying the number of rare itemsets. We notice that Rarity speedup decreases when the number of rare itemsets increases, although it is constantly greater than 1. The same analysis has been performed on extremely rare itemsets, selected as those with a support count below 1, 5, 10 and 20 occurrences. Figures 10 and 11 respectively compare execution time and memory usage. In this case, itemsets become extremely rare, and Rarity performances become spreader in time, using more memory in general as depicted in Fig. 12. This outcome is confirmed by Fig. 13 related to speed of Rarity versus ARIMA.

One could argue that improving the search of border would lead to faster search. To test this hypothesis we performed an additional experiment employing MINIT (Haglin and Manning 2007), which shares with ARIMA (Szathmary et al. 2007) the idea of traversing the lattice from the bottom. Since the goal of MINIT is to discover minimal infrequent itemsets, this leads to reach the border. From the border we can move further to discover the remaining rare itemsets. We developed an implementation of MINIT based on the pseudo-code provided in Haglin and Manning (2007). For the second step we implemented a mining strategy inspired to FP-growth (Han et al. 2004). As test cases we considered sparse and dense datasets made of 10, 15 and 20 items, respectively with records of 5, 10 and 15 records. In the case of dense datasets, records have the same length; in the case of sparse datasets, record length is variable up to the given length limit. Support threshold were chosen at 1 and 10 %. Results are collected

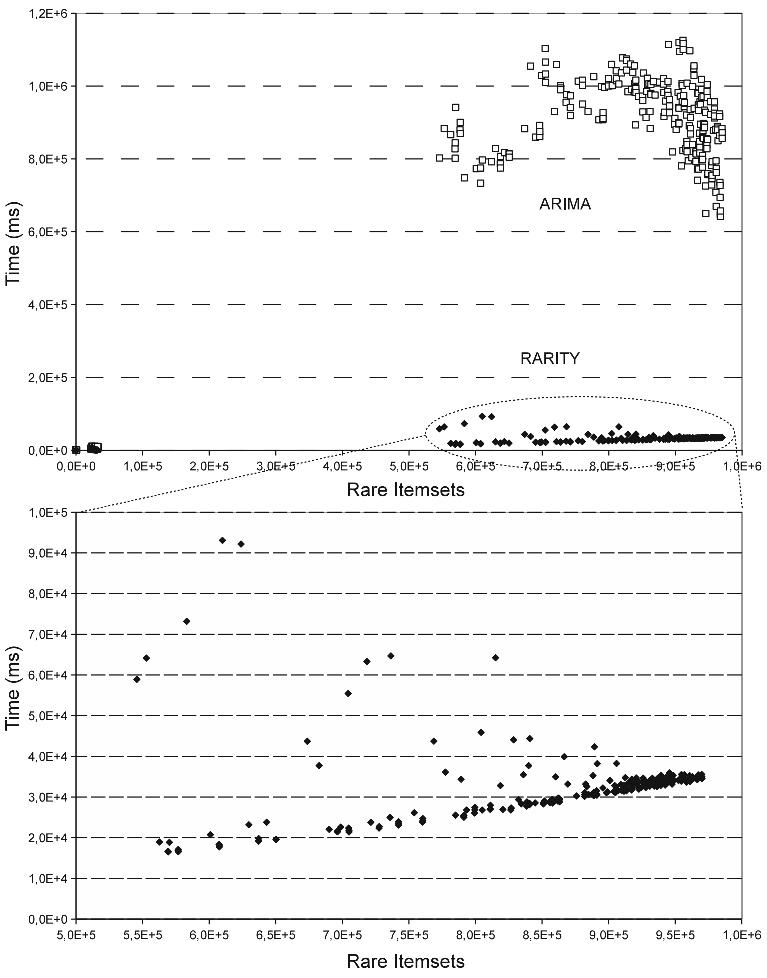


Fig. 6 Execution time: ARIMA and Rarity (Troiano et al. 2009)

in Table 2. Although MINIT is very efficient in searching for minimal infrequent itemsets, overall search completed in longer time when compared to Rarity. In one cases, MINIT was not able to complete the search. Two considerations are interesting. The first is that, even when the search is aimed only at testing non-zero rare itemsets, MINIT performed worse than Rarity (see Dense Dataset 1, Support threshold at 1 %). The second is that MINIT improves when the support threshold is lowered, since the first step is moved forward. In any case Rarity performed better.

7 Enhancements

As shown in the previous section, Rarity lacks of memory usage. This depends on the number of itemsets to explore and on their dimension, therefore on the generation of

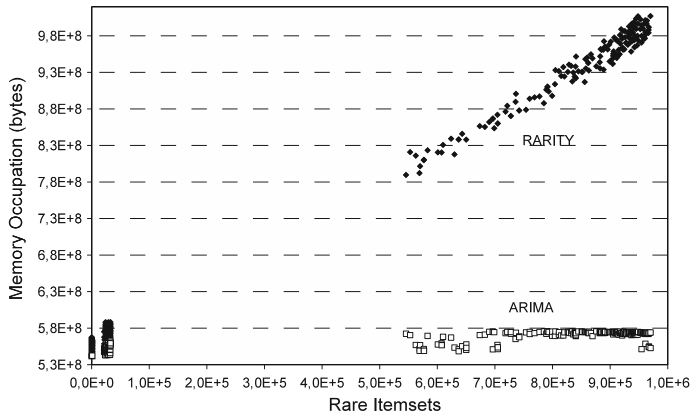


Fig. 7 Memory occupation: ARIMA and Rarity (Troiano et al. 2009)

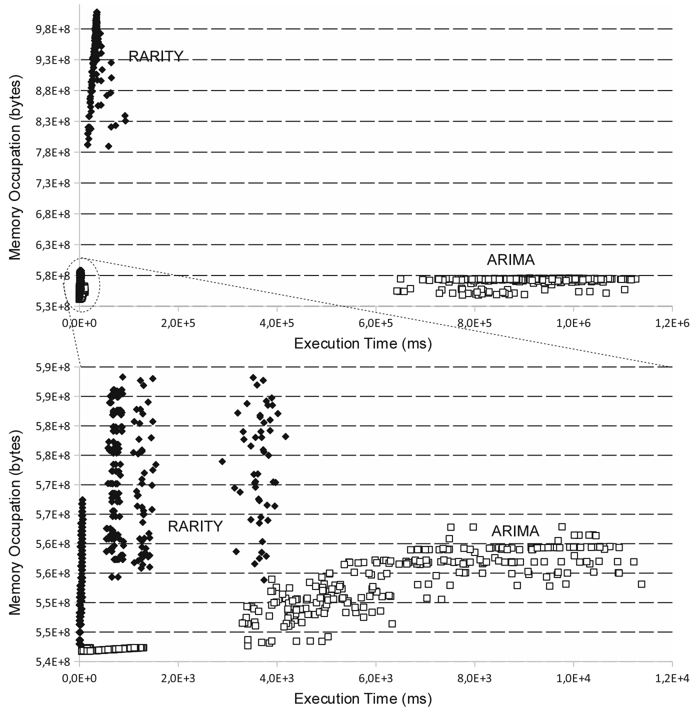


Fig. 8 Memory versus time: RARITY versus ARIMA (Troiano et al. 2009)

candidates and on veto lists. Keeping all information in memory can be very demanding for medium and large databases. Indeed, although lists have no duplicates and each itemset is stored in only one of them at once, the quantity of information to store and process can easily exceed memory capacity. This is mainly due to combinatorial complexity of the search space.

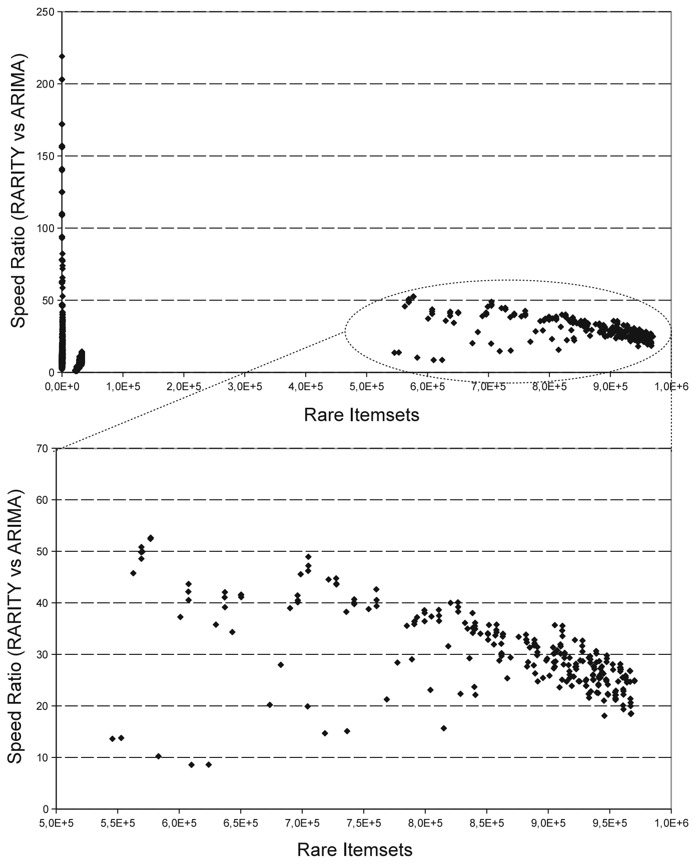


Fig. 9 Speed ratio: RARITY versus ARIMA (Troiano et al. 2009)

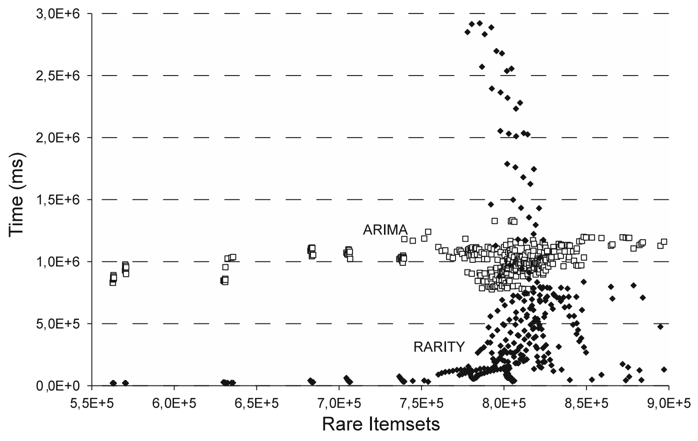


Fig. 10 Extremely rare itemsets. Execution time: RARITY versus ARIMA (Troiano et al. 2009)

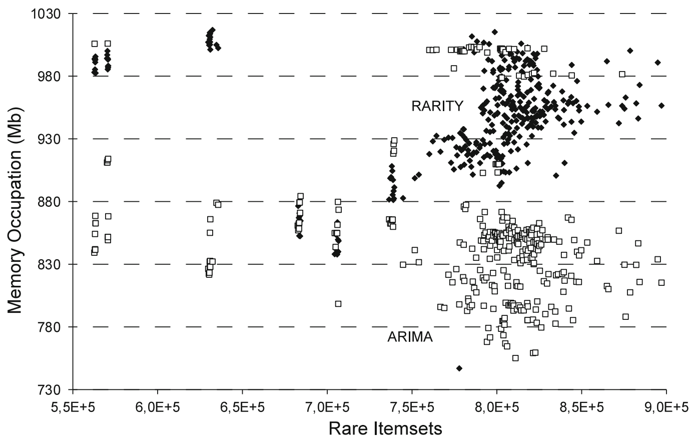


Fig. 11 Extremely rare itemsets. Memory occupation: RARITY versus ARIMA (Troiano et al. 2009)

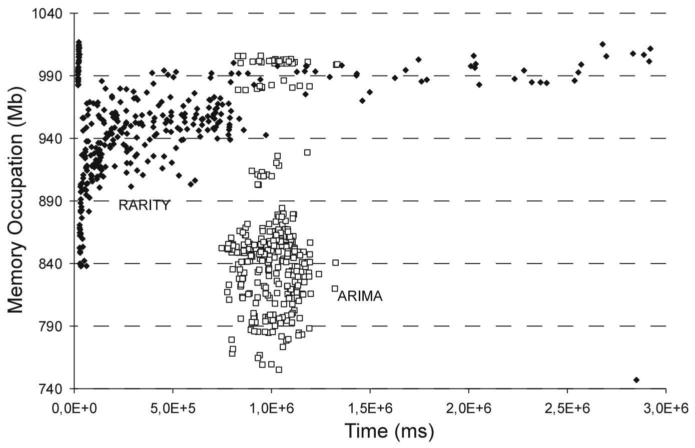


Fig. 12 Extremely rare itemsets. Memory versus time. (Troiano et al. 2009)

Indeed only 20 items can lead the algorithm to explore a search space potentially made of $2^{20} = 1,048,576$ itemsets, and since each of them requires about 50 bytes, there is a need of 50 Mb to store them. If the number of items becomes 30, the memory required to store the whole lattice becomes 50 Gb. The number of itemsets increases exponentially, so the memory required. This means that larger databases would be very memory demanding. For example if we use a database containing protein localization sites, such as in ECOLI database (Nakai 1996a), we could fail our analysis.

In order to solve this problem we need to rely on mass memory for storing partial results. The idea is to keep in memory only a part of itemsets, keeping the remaining on files. For this, we set the maximum number of itemsets $max\_itemsets$ that can be made available in memory, and use a file for each level of each list. When a list in memory reaches that limit, it is cleared out and itemset details such as (i) the bitmap, (ii) the length, (iii) the contribution vector $\mathbf{v}$ are stored to files, so that new itemsets can

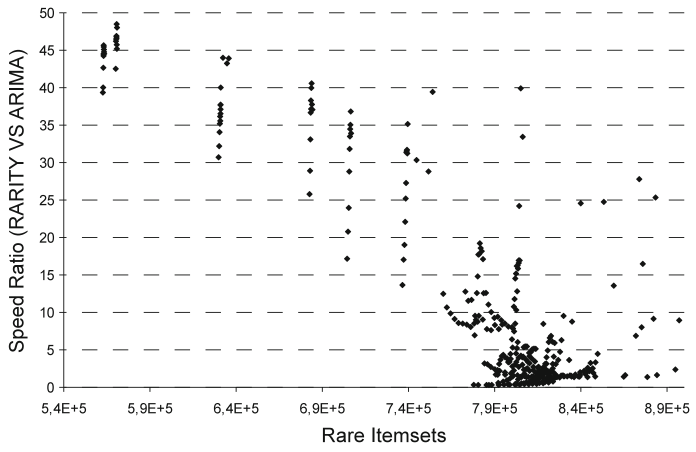


Fig. 13 Extremely rare itemsets. Speed ratio: RARITY versus ARIMA (Troiano et al. 2009)

Table 2 Rarity versus MINIT

Support (%)		MINIT			Rarity	
	Minimal itemsets	Time (ms)	Rare itemsets	Time (ms)	Rare itemsets	Time (ms)
		Dense dataset 1–10 items 5 column 10, 000 row				
1	252	465	252	498	252	143
10	120	344	27,473	811	27,473	666
		Dense dataset 2–15 items 10 column 10, 000 row				
1	6,396	3,543	13,832	5,425	13,832	629
10	3,003	1,178	28,314	7,634	28,314	452
		Dense dataset 3–20 items 15 column 10, 000 row				
1	157,646	72,407	394,757	139,714	394,757	520
10	77,518	10,207	–	–	964,924	5,975
		Sparse dataset 1–10 items 5 column 10, 000 row				
1	111	113	317	142	317	81
10	45	112	372	137	372	115
		Sparse dataset 2–15 items 10 column 10, 000 row				
1	2,988	1,681	18,373	3,040	18,373	471
10	455	316	20,208	707	20,208	309
		Sparse dataset 3–20 items 15 column 10, 000 row				
1	59,144	5,998	695,517	161,355	695,517	3,665
10	1,140	372	752,021	113,712	752,021	4,011

be hosted in memory. As we know in advance the record size representing an itemset, we can set an optimal value of $max\_itemsets$ in order to avoid memory lackage. The pseudo-code is outlined in Algorithm 2.

Implementation is based on groups of files F_C containing candidate itemsets, F_V frequent itemsets and F_R rare itemsets. As each file is assigned to a level, it contains itemsets of the same length. The idea is to apply Rarity to blocks of itemsets. For this purpose, we introduce functions *load* and *store* for reading and saving blocks of itemsets. They accept two parameters: a list as first, and the itemset to store or the level to read as second. In particular, *store* and *load* performs as cache proxies, so that when a block is full it is saved (store), or when all itemsets have been processed a new block is loaded (load). This makes it possible to preserve the algorithm structure. When itemsets are processed by blocks, it may happen that some of them are generated in more blocks. Indeed an itemset could be generated as subset of itemsets belonging to different blocks, or the generation of itemsets could lead the algorithm to exceed the memory limit and to flush the block to the mass memory.

This means itemset duplicates are possible, entailing that larger files occur and the contributions to support vector are split along different duplicates. The latter issue might lead the algorithm to misclassify an itemset due to an underestimation of item-

Algorithm 2 Enhanced Rarity

```
Input: $\mathcal{D}$ - Dataset
Output: $R$ - Rare itemset collection
1: $l_m = \max len(t)\ \forall t \in \mathcal{D}$
2: $mi = max\_itemsets$
3: for all record $t \in \mathcal{D}$ do
4:   $store(C, t)$
5: end for
6: for $l = l_m..1$ do
7:   if $C(l) \neq \emptyset$ then
8:     $clearCandidates(l)$
9:     for all $is \in load(C, l)$ do
10:      if $supp(is) > min\_supp$ then
11:        remove $is$ from $C(l)$
12:        $store(V, is)$
13:      else
14:        $store(R, is)$
15:        if $len(is) > 1$ then
16:          for all $sub \in subsets(is)$ do
17:            if $sub \notin V$ then
18:              $store(C, sub)$
19:              $\mathbf{v}_{sub} = \mathbf{v}_{sub} + \mathbf{v}_{is}$
20:            end if
21:          end for
22:        end if
23:      end if
24:    end for
25:    $deleteDuplicates.V(l)$
26:    for $lv = l..1$ do
27:      $deleteDuplicates.C(lv)$
28:    end for
29:    for all $is \in load(V, l)$ do
30:      if $len(is) > 1$ then
31:        for $k = l - 1..1$ do
32:          for all $c \in C(k)$ do
33:            $cis = c \bigcap is$
34:            $store(V, cis)$
35:          end for
36:        end for
37:      end if
38:    end for
39:  end if
40: end for
```

set's support count. Therefore, there is a need of removing duplications and summing up the support provided by duplicates. This is done by functions *deleteDuplicates* and *clearCandidates*. The first, *deleteDuplicate*, is responsible of removing duplications in the candidate file F_C or in the veto file F_V at a given level, and to merge the support vector of duplicates. The latter operation is not necessary for vetoed itemsets. For this reason we provide two versions *deleteDuplicates.C* for candidate and *deleteDuplicates.V* for vetoed itemsets. After this operation, the actual support count of some candidates could be over the limit, thus they are added to the veto list. However, some duplicates could have been already identified as frequent by themselves and discarded. So that a candidate could get a support count that does not take into account its clones that have been vetoed. The function *clearCandidates* makes a double check, verifying that remaining itemsets do not have duplicates in the corresponding veto file: they are vetoed in that case. The support count of remaining itemsets is correct and if under the limit, they can be added to R.

Although we make use of files, search and comparison in lists made difficult due to the size of files, that does not allow to process them fully in memory. Indeed, the size of files is increased by duplicates, whose number can be estimated in the worst case. At each level l, the number of possible itemsets is at most equal to

$$N(l) = \binom{m}{l} = \frac{m!}{(m-l)!l!} \tag{4}$$

where $m = |I|$ is the number of items in $\mathcal{D}$. Each of them can give birth to l subsets long $l-1$, so that the overall number of possible subsets is $l \cdot \binom{m}{l}$. Among these, $\binom{m}{l-1}$ are unique, so that the number of dublicates is

$$l \cdot \binom{m}{l} - \binom{m}{l-1} = \frac{l(m-l)}{m-l+1}\binom{m}{l}. \tag{5}$$

Therefore, the overall number of possible duplicates becomes

$$\sum_{l=2}^{m} \frac{l(m-l)}{m-l+1}\binom{m}{l} \tag{6}$$

since the last level, which lead to empty set, is not of interest.

The structure in blocks requires to use a cross-matrix approach, as depicted in Fig. 14 and outlined in Algorithm 3 and Algorithm 4. By this approach, a faster cursor (Q) is used within a slower cursor (P), so that each block is compared to the others in a file. In particular, deletion of duplicates is performed by looking at previous blocks. If the itemset has been already considered, that means the itemset in the current block is a duplication, thus removed. This is performed by operation *Clean* in both Fig. 14-a, b. In the case of candidates, there is also the need of updating the support vector, by summing up contributions provided by the duplicates. This is done by looking forward at duplicates in the following blocks. This is performed by operation *Update* in Fig. 14-b. Those duplicates will be removed later on.

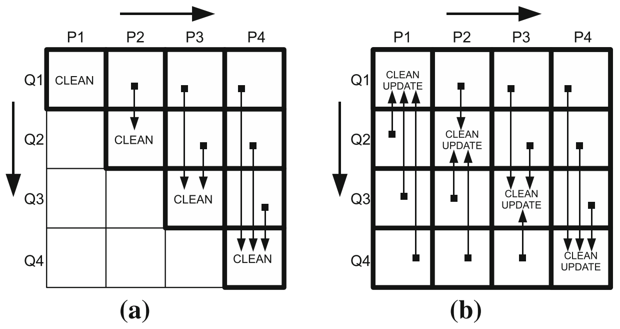


Fig. 14 Deleting duplicates in veto (**a**) and candidate (**b**) files

Algorithm 3 deleteDuplicate.V(l)

```
Input: $V(l)$ - Veto list at level $l$
Output: $V(l)$ cleaned up of duplicates
1: repeat
2:   $P = load(V(l))$
3:   repeat
4:     $Q = load(V(l))$
5:     for all itemset $i \in Q$ do
6:       if $i \in P$ then
7:         remove $i$ from $P$
8:       end if
9:     end for
10:  until block $P$ is reached
11:  save $P$ in $V(l)$
12: until end of blocks
```

Algorithm 4 deleteDuplicate.C(l)

```
Input: $C(l)$ - Candidate list at level $l$
Output: $C(l)$ cleaned up of duplicates
1: repeat
2:   $P = load(C(l))$
3:   repeat
4:     $Q = load(C(l))$
5:     for all itemset $i \in Q$ do
6:       if $i \in P$ then
7:         if $Q \prec P$ then
8:           remove $i$ from $P$
9:         else
10:          updateVector($i$)
11:        end if
12:      end if
13:    end for
14:  until end of blocks
15:  save $P$ in $C(l)$
16: until end of blocks
```

In some cases of practical interest, the analysis could be directed to mine rare itemsets with a support count over, i.e. equal or greater than, a given minimum threshold t_m (assuming $t_m < t_r$), in order to filter out spurious patterns. For example, this is the case of *rare-but-not-unique* itemsets (i.e. $t_m = 2$). In this case, it would be worth beginning the exploration of lattice from a list of candidates occurring at least t_m times. We call this candidate *pre-filtering*. Candidates with a support count over t_m, can be generated by assuming in sequence intersections made of t_m transactions occurring in the database. For example if $\mathcal{D} \equiv \{T_1, T_2, T_3, T_4\}$ and $t_m = 3$, we will consider intersections $T_1 \cap T_2 \cap T_3$, $T_1 \cap T_2 \cap T_4$, $T_1 \cap T_3 \cap T_4$ and $T_2 \cap T_3 \cap T_4$, all resulting into itemsets whose support count is at least t_m. Generation of those candidates requires to consider $\binom{|\mathcal{D}|}{t_m}$ intersections, where $|\mathcal{D}|$ is the number of transactions in $\mathcal{D}$. Therefore time complexity is $O(|\mathcal{D}|^{t_m})$, that is polynomial w.r.t. t_m. Since t_m is generally small, i.e. $t_m = 1..3$, candidate pre-filtering is feasible and scalable. We can burst pre-filtering removing from the dataset in advance all those items (singleton) which occurs lesser than t_m times. For larger values of t_m, it would be better to stop pre-filtering at a given small threshold, and to continue with the algorithm, discarding those itemsets that are not reaching the minimal number of occurrences along the process (*in-filtering*) and after the process (*post-filtering*). Implementation of both is not considered in the remainder of this paper.

Convergence of Rarity under these circumstances is assured by the following proposition:

Proposition 2 *Let C_1 be the initial candidate list made of itemsets $X_1 \dots X_p$ s.t. $supp(X_i) \geq 1$ obtained without pre-filtering, and C_m the list made of candidates $Z_1 \dots Z_q$ obtained by pre-filtering. It stands that $\forall\ Y \subseteq X_Y$* **s.t.** *$supp(Y) \geq t_m \exists\ Z_Y$* **s.t.** *$Y \subseteq Z_Y$*

Proof Since $supp(X_i) \geq 1\ \forall X_i \in C_1$ we get that $\exists\ T \in D : X_i = T$. On the other side, since C_m is obtained by pre-filtering, $\forall\ Z_j \in C_m$ we get by construction $supp(Z_j) \geq t_m$ and $\exists\ X_i \in C_1$**s.t.** $Z_j \subseteq X_i$.

Ad absurdum let us assume that $\exists Y \subseteq X_Y \in C_1 : supp(Y) \geq t_m$ such that $\neg\exists Z_Y \in C_m : Y \subseteq Z_Y$. Since $supp(Y) \geq t_m$ we have $\exists T_{Y1} \dots T_{Ym} \in D : Y \subseteq T_{Yi}, i = 1..m$. But by construction $Y \subseteq Z_Y = T_1 \cap \cdots \cap T_m$ and $Z_Y \in C_m$. □

This assures that if a candidate X is below the minimum threshold t_m, thus discarded from the initial candidates, its subsets Y, occurring at least t_m times, will be generated by some other candidate $Z \in C_m$ whose support count is not lesser than t_m. Therefore there is no risk to miss any itemset. In addition, since $t_r > t_m$, also non-rare itemsets are considered and vetoed by the algorithm.

Pre-filtering as outlined above requires to consider the intersection of transaction in $\mathcal{D}$. This can be computationally demanding. Thus, in order to get a faster generation of C_m, it is possible to obtain it by considering itemsets in C_1, as they provide a more compact representation of $\mathcal{D}$. For the sake of simplicity we avoid to describe this optimization in details.

We point out that the problem of listing non-zero rare itemsets, or over a given minimal threshold, is extremely hard to scale-up problem complexity due to the extremely large number of solutions. Therefore the raw application of any technique to large item collections will lead to a quantity of data that, despite of their theoretical interest, they could become useless in practice. However, the application of a top-down strategy will lead to discover first those itemsets that are of more interest as generally rarer and longer. Therefore a limit to level exploration can be set.

8 UCI dataset experimentation

Enhancements presented above are necessary in order to apply Rarity to datasets with a large number of items and transactions. In this paper we will consider three known datasets from UCI repository with different characteristics, namely *ECOLI* (Nakai 1996a), *YEAST* (Nakai 1996b) and *WINE* (Forina 1991).

Due to problem complexity the application of Rarity, ARIMA (Szathmary et al. 2007) and AfRIM (Adda et al. 2007), and any other similar algorithm, is not feasible in the case of dataset such as *Mushroom*[6] *T20I6D100K*[7] and *C20D10K*[8] as they have respectively 119, 893 and 192 items.

8.1 Datasets

ECOLI contains data about the cellular localization sites of proteins. There are 100 different items, leading to a possible search space of at most 2^{100} itemsets. A similar dataset is *YEAST*. It also related to data regarding the cellular localization sites of proteins but it is larger and with 106 items. Finally *WINE* contains data from a chemical analysis of wines grown in the same region in Italy but derived from different cultivars. It is not very large but contains a high number of items: 809.

A comparison to ARIMA is not possible for these datasets as processing was not able to complete with datasets with such characteristics, resulting in memory leakage and program crashing.

8.2 Results

Experimental results are shown in Table 3, which reports the number of rare itemsets found, the number of itemsets explored, execution time in milliseconds and memory occupation, to run the algorithm at different support thresholds, expressed as percentage of available records.

These results bring to several important observations. First of all, the number of evaluated itemsets is much lower than the lattice size and closer to the number of rare itemsets. This entails that Rarity is able to accomplish the task efficiently and effectively. Even though the WINE dataset has less instances than the others, it requires higher computation time and memory, since the number of items is larger and the search space sparser.

[6] http://archive.ics.uci.edu/ml/datasets/Mushroom.

[7] http://www.almaden.ibm.com/software/quest/Resources.

[8] http://www.census.gov/.

Table 3 Experimental results with UCI datasets

Supp. (%)	Supp. count limit	Rare itemsets	Evaluated itemsets	Time (ms)	Memory (bytes)
		$ECOLI$-100 items			
0. 5	n/a	n/a	n/a	n/a	n/a
1	3	51,410	51,812	31,183	38,962,760
5	16	52,855	52,899	5,552	47,247,128
10	33	53,020	53,028	5,636	42,674,528
		$YEAST$-106 items			
0. 5	7	361,242	363,076	62,372	22,807,232
1	15	364,957	365,651	58,445	38,734,176
5	74	366,788	366,856	50,045	58,641,336
10	148	366,937	366,955	47,723	52,647,576
		$WINE$-809 items			
0. 5	n/a	n/a	n/a	n/a	n/a
1	2	1,386,138	1,386,403	5,528,117	42,640,856
5	10	1,386,438	1,386,451	5,508,962	162,977,184
10	20	1,386,451	1,386,451	5,512,530	133,005,904

9 An estimation of complexity

Giving an estimation of complexity when an exhaustive search for rare itemsets is employed, we should consider that, given n records, with a maximum length of m items, time and memory complexity would be $O(\frac{1}{2}n(n-1)(2^m-1))$, as each record would entail $2^m - 1$ non-empty subsets, and this should be searched for in the remaining $n-1$ records. Coefficient $1/2$ takes into consideration that only the records following the current would be considered. This limit represent the upper bound for any search strategy of rare itemsets.

Assessing the complexity of Rarity is too hard, since it depends on many variables such as the support count limit, the number of records, the number of items, but mostly from the sequence of decision points taken along the process. With the enhanced version things become even more complicated, as the number of possible duplicates has a relevant effect on processing time and memory.

9.1 Elementary operations

Therefore, renouncing to find a reasonable theoretical complexity estimation, we followed an empirical approach by counting the number of elementary operations, i.e. (i) *add to list*, (ii) *remove from list*, (iii) *support evaluation* and (iv) *itemset intersection*, and relating them to problem characteristics, i.e. *support threshold*, the *number of items*, *number of records*, and to algorithm performances, measured by *time*. *Remove* is the heaviest operation because it entails a search for the itemset in the list for deletion.

In this section we will refer to the basic version of Rarity. Elementary operations occur in Algorithm 1 as follows: *add* at lines 3, 10, 12, 16 and 29; *remove* at lines 9 and 28; *support evaluation* at line 8; *intersection* at line 27. Experiments were carried out using the same test cases mentioned in Sect. 5.

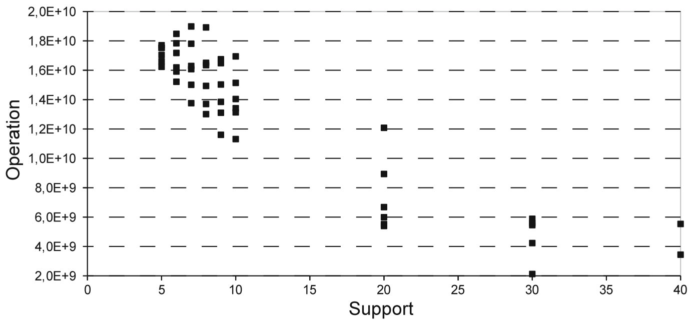


Fig. 15 Support count limit versus operations

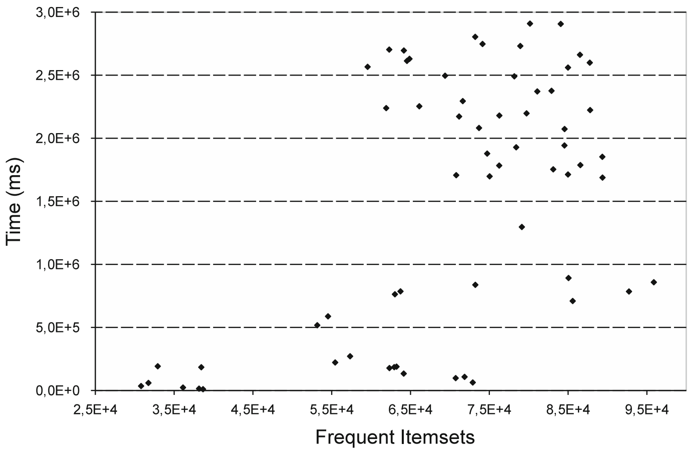


Fig. 16 Number of non-rare itemstes versus time

9.2 Results

Figure 15 outlines the relationship between the support count limit and the weighted number of elementary operations, showing that the higher is the support limit the lower is the number of required operations. Indeed, if the limit is high, the number of rare itemsets becomes larger. Therefore, even though the number of support evaluation increases, since we do not have many non-rare itemsets, we avoid to perform remove operations, the most computational expensive. The same explanation stands for Fig. 16, which shows that when the number of frequent itemsets increases, the execution time increases also. Fig. 17 shows that when the number of support evaluations increases, time decreases, as a high number of frequent itemsets leads to the execution of additional two remove operations.

Figure 18 outlines the relationship between the number of support evaluations and the support count limit, depicting that when the limit increases, the number of evaluations increases too. This is apparently in contradiction with conclusions above. However, Rarity entails a very small number of support evaluations with respect to the other operations. Indeed for each database we had at most $9 \cdot 10^5$ support evaluations

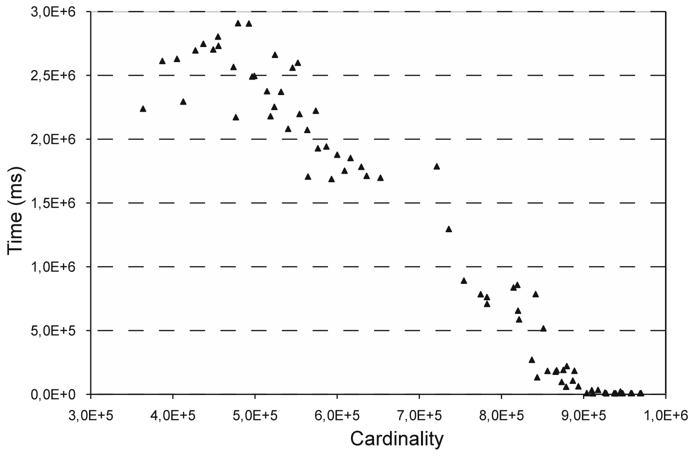


Fig. 17 Support evaluations versus time

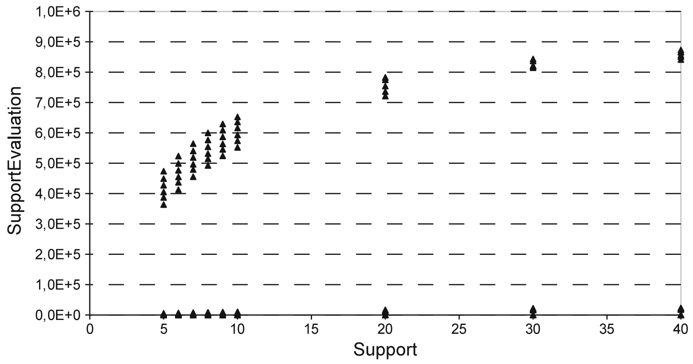


Fig. 18 Maximum support versus support evaluations

within an overall number of about $1 \cdot 10^{10}$ operations. The growth of the number of support evaluations depends on the fact that the algorithm stops later, so that the number of levels to evaluate and the number of candidates becomes larger than in the case of a lower support threshold.

10 Parallelization

Application of Rarity to larger databases as of interest in many real world problems, would require to take into consideration alternative versions of algorithms presented above, with particular attention to parallelization. A possible further enhancement could be thought with respect to representation of data. For instance, Viper (Shenoy et al. 2000) and Mafia (Burdick et al. 2001), attempting to optimize the search for frequent patterns, introduced a vertical database representation having compressed bitmaps to represent the itemset transaction list. They proved that in many cases this approach is better than a horizontal layout. We implemented a similar approach, but it did not improves performances. This brought us to study parallel solutions for our algorithm.

Several parallel implementations have been proposed in literature for mining association rules. Most of them are variants of Apriori. For instance, Agrawal and Shafer (1996) propose three different parallel versions of Apriori for distributed memory environments. The first algorithm replicates the generation of the candidates and it is a straigthforward parallelization of the original version. The other two algorithms distribute the candidate itemsets among the processors. A similar approach is presented in Park et al. (1995), entailing the replication of candidate itemsets on all processors.

In this section we do not provide a parallel implementation but we outline possible solutions. The first consists in partitioning the initial dataset into smaller datasets and distributing them among processors which execute Rarity and each returns a block containing the rare itemsets found in lexicographic order. The merging of blocks is performed by scanning the blocks, using a pointer to the current itemset for each them. We choose the itemset that lexicographically comes first among the pointed ones in each block. If more than one itemset is selected, the support count of them is summed. If the current itemset is rare, we keep it, otherwise we discard it. We move forward the pointer of selected blocks. If there is no more itemsets to consider for a block, that block is dismissed. The procedure stops when there is no further itemset to consider, thus all blocks have been dismissed.

The second solutions consists in executing the function $deleteDuplicates$ used in the Algorithm 2 upon different processors. In order to apply a load balance strategy, files would be distributed taking into account their dimensions. A further enhancement consists in executing the $deleteDuplicates$ as soon as an itemset (frequent or candidate) is generated. This would improve memory management, resulting in early discard of duplications, although poses communication issues in distributed environments.

11 Solving the motivating examples

We provide a solution to problems posed in Sect. 2. The aim of this section is only illustrative, as providing a definitive answer to these problems would be out of the scope. Results are outlined in Table 4.

11.1 Spare parts

In this example, data refer to maintenance activity performed by BS, an industrial machinery manufacturer based in Milan (Italy). Each record lists the spare parts used in maintenance along a period of 6 months (from Feb 1st 2009 to Aug 1st 2009). Due to no-disclosure agreement (NDA), parts have been made anonymous and identified by numbers. The dataset is sparse and made of 1, 244 records, listing collections counting from 4 to 20 spare parts chosen in a list of 28 items. An example of data is given in Table 5. As an example of results we consider two long itemsets:

Table 4 Overall results of examples

Level limit	Supp. (%)	Supp. count limit	Rare itemsets	Time (ms)	Memory (bytes)
Spare					
None	0.5	6	1,890,145	94,830	154,763,872
None	5	62	1,957,540	15,222	164,837,216
Unit					
None	1	30	289,617	2,411	84,220,208
None	5	150	289,849	2,361	84,563,432
Text					
ACQ					
14	15	823	797,755	13,076	756,747,755
EARN					
14	15	823	928,414	20,129	964,545,533

Table 5 An excerpt of spare and unit dataset

Spare	Unit
`1 11 13 14 17 19 2 20 21 23 24 25 3 4 6 7`	`000 036 120 058 041 097 072 091`
`0 1 12 13 14 16 18 20 4 7 9`	`024 023 108 072 046 090 059 097`
`1 10 11 18 19 21 23 25 5 6`	`001 015 121 066 033 094 059 114`
`1 11 13 15 17 2 25 26 5 7 9`	`012 029 121 077 051 100 058 095`
`11 15 17 18 2 20 21 24 25 7 9`	`014 019 119 078 033 093 062 097`
`10 11 13 16 19 22 6 7 8`	`013 020 101 070 044 098 054 113`
`0 1 10 15 2 26 4 6 8 9`	`004 019 106 076 042 087 057 096`
`1 11 12 15 19 20 22 23 24 3 4 9`	`007 036 116 071 047 100 074 101`
`1 13 15 19 21 3 5 6 8`	`005 035 098 076 050 086 073 100`
`11 12 15 16 21 4 5`	`001 028 123 076 047 080 072 096`
`0 1 10 11 13 18 19`	`009 043 119 072 031 096 045 096`
`1 12 14 15 18 20 22 24 26 3 6 7 9`	`003 019 113 064 055 091 065 092`
`1 10 13 14 15 16 17 18 21 22 25 6 8`	`004 025 104 082 035 075 068 109`
`0 1 14 18 19 21 23 24 5 8`	`008 027 104 081 033 084 061 117`
`10 13 14 15 17 18 2 24 7 8`	`019 028 120 072 055 097 051 104`

`1 2 4 5 7 8 9 11 12 15 16 21 22 23 24 26 27`
`2 3 4 5 7 8 9 10 11 16 17 18 19 20 21 27`

We can note that they share items `2 4 5 7 8 9` that are related to parts of an electromechanical group.

11.2 Unit test

This case deals with a unit used to process data collected by 8 accelerators. Inputs are given as words of 7 bits to the function and there processed. Since unit could fail when some inputs are given, we might be interested to those input combinations that are rarely occurring. We consider a collection of $3,000$ records each made of

8 integer values in 0..127. Equal values at different positions are treated as different items. Larger rare itemsets are made of 8 values. For example:

```
024 042 113 083 032 103 070 112
016 039 105 075 035 077 046 096
```

More interesting are short groups of values. Examples of 2-itemsets are:

```
*** *** 106 080 *** *** *** ***
018 *** *** *** *** *** *** 113
*** *** *** 060 *** *** 073 ***
```

11.3 Text analysis

For testing our algorithm in text analysis problems, we chose Reuters-21578[9], R8 instance. We remember that this dataset refers to a collection of documents originally collected and labeled by Carnegie Group, Inc., and Reuters, Ltd., along the development the CONSTRUE text categorization system. Each line is associated to a document, each document is composed by its class and terms. Instance R8 considers 8 of the most frequent classes. Terms have been filtered by considering only words (i.e. sequences of letters) made of at least 3 characters, removing 524 stop words and stemmed by Porter's method. In addition we filtered terms that are below 10 % (too sporadic) and over 25 % (too common). The maximum support has been setup at 15 % and level limit to 14. Examples of itemsets of words we mined for classes ACQ and EARN are reported in Table 6.

12 Conclusions

In this paper we faced the problem of mining rare itemsets. Although, most of the research focused on discovering frequent itemsets, the search for rare itemsets can reveal valuable and unexpected knowledge. Most of the algorithms proposed in literature until now take inspiration from Apriori, the first and most noticeable algorithm for searching for frequent itemsets. Because of this, most algorithms explore the lattice of itemsets from singletons and/or limit the search to some specific classes of rare itemests. However, rare itemsets are the largest, and placed on the top of the lattice according to the convention followed by this paper.

We investigated the possibility of exploring the power set lattice from the top, reaching the border of non-rare itemsets. This strategy has been employed in Rarity,

[9] http://web.ist.utl.pt/~acardoso/datasets/

Table 6 Examples of itemsets mined for ACQ and EARN classes.

ACQ													
Global	Enorm	Intern	Growth	Take	You	Want	Your	Realist	Valuat	Enhanc	Abil	Kind	Endeavor
670	671	109	262	672	673	326	674	675	676	677	678	679	680
Drexel	Offici	Stake	Epsilon	Data	Epsi	Senior	Burnham	Lambert	Commiss	Acquir	Father	Told	Secur
940	312	585	941	942	943	944	945	946	422	33	947	145	589
EARN													
Declin	Dollar	Research	Develop	Engin	Expenditur	Will	Rang	Compani	Alloc	Anoth	Capit	Earlier	Fourth
222	292	251	464	465	466	154	158	15	467	346	22	151	138
Peninsula	Feder	Save	Loan	Associ	Januari	Gain	Per	Share	Exchang	Pan	America	Bank	Inc
549	439	550	101	551	447	105	43	11	442	552	437	392	5

a breadth-first level-wise lattice-traversal algorithm, which moves (top-down) from larger to smaller itemsets and makes use of an effective way to compute itemset's support count. Experimental results proved that this approach takes advantage of finding all rare non-zero itemsets in less time than other solutions, at expenses of higher memory demand. This leads us to consider enhancements in order to apply the algorithm to real datasets of practical interests.

In some circumstances it would be worth searching for itemsets that are rare but occurring at least a minimal number of times. This is the case of rare-but-not-unique itemsets. Pre-filtering has been suggested as means to filter out those itemsets below the minimum threshold. This makes it possible to initialize the algorithm with a reduced number of candidates. Other enhancements are aimed at over-coming memory limitations by splitting the datasets in blocks. A parallel version has been outlined. However, the number of rare itemsets can be extremely large, and this poses questions regarding tractability of the problem of fully listing them for larger and sparser datasets.

References

Adda M, Wu L, Feng Y (2007) Rare itemset mining. In: Proceedings of 6th International Conference on Machine Learning and Applications, ICMLA '07. IEEE Computer Society, Washington, DC, pp 73–80

Agrawal R, Imieliński T, Swami A (1993) Database mining: a performance perspective. IEEE Trans Knowl Data Eng 5(6):914–925

Agrawal R, Imieliński T, Swami A (1993) Mining association rules between sets of items in large databases. ACM SIGMOD Int Conf Manag Data 22:207–216

Agrawal R, Mannila H, Srikant R, Toivonen H, Inkeri Verkamo A (1996) Fast discovery of association rules. In: Advances in knowledge discovery and data mining. AAAI/MIT Press, Cambridge

Agrawal R, Shafer JC (1996) Parallel mining of association rules. IEEE Trans Knowl Data Eng 8(6):962–969

Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: 20th VLDB Conference

Bastide Y, Taouil R, Pasquier N, Stumme G, Lakhal L (2000) Mining frequent patterns with counting inference. SIGKDD Explor Newsl 2(2):66–75

Brin S, Motwani R, Ullman JD, Tsur S (1997) Dynamic itemset counting and implication rules for market basket data. In: SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data. ACM, New York, pp 255–264

Burdick D, Calimlim M, Gehrke J (2001) Mafia: a maximal frequent itemset algorithm for transactional databases. In: Proceedings of the 17th International Conferences on Data Engineering. IEEE Computer Society, Washington, DC, pp 443–452

Forina M (1991) Wine dataset. http://archive.ics.uci.edu/ml/datasets/wine. Accessed 5 Nov 2012

Haglin DJ, Manning AM (2007) On minimal infrequent itemset mining. In: DMIN. CSREA Press, Las Vegas, pp 141–147

Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: a frequent-pattern tree approach. In: Mannila H (ed) Data mining and knowledge discovery. Kluwer, New York, pp 53–87

Koh YS, Rountree N (2005) Finding sporadic rules using apriori-inverse. In: PAKDD. Springer, New York, pp 97–106

Koh YS, Rountree N, O'Keefe RA (2008) Mining interesting imperfectly sporadic rules. Knowl Inf Syst 14(2):179–196

Liu B, Hsu W, Ma Y (1999) Mining association rules with multiple minimum supports. In: KDD '99: Proceedings of 5th ACM SIGKDD International Conferences on Knowledge Discovery and Data Mining. ACM, New York, pp 337–341

Mannila H, Toivonen H, Verkamo I (1994) Efficient algorithms for discovering association rules. In: KDD '94: Proceedings of the AAAI Workshop on Knowledge Discovery in Databases. AAAI Press, Seattle, pp 181–192

Nakai K (1996a) Ecoli dataset. http://archive.ics.uci.edu/ml/datasets/ecoli. Accessed 5 Nov 2012

Nakai K (1996b) Yeast dataset. http://archive.ics.uci.edu/ml/datasets/yeast. Accessed 5 Nov 2012

Park JS, Chen M-S, Yu PS (1995) Efficient parallel data mining for association rules. In: CIKM '95: Proceedings of 4th International Conference on Information and Knowledge Management. ACM, New York, pp 31–36

Pasquier N, Bastide Y, Taouil R, Lakhal L (1999) Closed set based discovery of small covers for association rules. In: Proceedings 15emes Journees Bases de Donnees Avancees. BDA, pp 361–381

Pei J, Han J, Lu H, Nishio S, Tang S, Yang D (2001) H-mine: hyper-structure mining of frequent patterns in large databases. In: ICDM '01: Proceedings of the 2001 IEEE International Conferences on Data Mining. Washington, DC, pp 441–448

Piatetsky-Shapiro G, Frawley WJ (eds) (1991) Knowledge discovery in databases. AAAI/MIT Press, Cambridge

Savasere A, Omiecinski E, Navathe SB (1995) An efficient algorithm for mining association rules in large databases. In VLDB '95: Proceedings of 21st International Conferences on Very Large Data Bases. Morgan Kaufmann, San Francisco, pp 432–444

Shenoy P, Haritsa JR, Sudarshan S, Bhalotia G, Bawa M, Shah D (2000) Turbo-charging vertical mining of large databases. SIGMOD Rec 29(2):22–33

Song M, Rajasekaran S (2006) A transaction mapping algorithm for frequent itemsets mining. IEEE Trans Knowl Data Eng 18(4):472–481

Szathmary L, Napoli A, Kuznetsov SO (2007) ZART: a multifunctional itemset mining algorithm. In: Proceedings of the 5th International Conferences on Concept Lattices and Their Applications (CLA '07). Montpellier, pp 26–37

Szathmary L, Napoli A, Valtchev P (2007) Towards rare itemset mining. In: ICTAI '07: Proceedings of 19th IEEE International Conferences on Tools with Artificial Intelligence. Washington, DC, pp 305–312

Troiano L, Scibelli G, Birtolo C (2009) A fast algorithm for mining rare itemsets. In: ISDA'09, pp 1149–1155

Tsang S, Koh YS, Dobbie G (2011) Rp-tree: rare pattern tree mining. In: Proceedings of CLA, pp 277–288

Uno T, Asai T, Uchida Y, Arimura H (2003) Lcm: an efficient algorithm for enumerating frequent closed item sets. In: FIMI03: Proceedings of Workshop on Frequent Itemset Mining Implementations

Uno T, Kiyomi M, Arimura H (2004) Lcm ver. 2: efficient mining algorithms for frequent/closed/maximal itemsets. In: FIMI '04, Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations

Uno T, Kiyomi M, Arimura H (2005) Lcm ver. 3: collaboration of array, bitmap and prefix tree for frequent itemset mining. In: Proceedings of 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations, ACM, New York, pp 77–86

Weiss GM (2004) Mining with rarity: a unifying framework. SIGKDD Explor Newsl 6(1):7–19

Yang G (2004) The complexity of mining maximal frequent itemsets and maximal frequent patterns. In: KDD '04: Proceedings of 10th ACM SIGKDD International Conferences on Knowledge Discovery and Data Mining. New York, pp 344–353

Yun H, Ha D, Hwang B, Ryu KH (2003) Mining association rules on significant rare data using relative support. J Syst Softw 67(3):181–191

Zaki MJ, Gouda K (2003) Fast vertical mining using diffsets. In: KDD '03: Proceedings of 9th ACM SIGKDD International Conferences on Knowledge Discovery and Data Mining. New York, pp 326–335

Zaki MJ, Parthasarathy S, Ogihara M, Li W (1997) New algorithms for fast discovery of association rules. Technical report, Rochester