Mining Generalized Association Rules Using Prutax and Hierarchical Bitmap Index

Jakub Pieprzyk[1] and Mikołaj Morzy[2]

[1] Comarch S.A.
Składowa 4, 61-897 Poznań
Jakub.Pieprzyk@comarch.pl

[2] Institute of Computing Science
Poznań University of Technology
Piotrowo 3A, 60-965 Poznań, Poland
Mikolaj.Morzy@cs.put.poznan.pl

Abstract. Association rules are among the most popular and widely used data mining techniques. Often, associations are sought between items forming a taxonomy. Patterns discovered between items from different levels of a taxonomy provide aggregated view over the data and allow to discover trends and regularities that are not apparent in the raw transactional data. Generalized association rule mining, i.e. mining in presence of a taxonomy of items, is an important augmentation to the original association rule mining framework. Unfortunately, currently available algorithms do not allow to efficiently discover generalized association rules.
In this paper we present the state-of-the-art in generalized association rule mining. We describe the hierarchical bitmap index, an efficient physical structure optimized for set processing. Next, we modify the *Prutax* algorithm by incorporating the hierarchical bitmap index as the crucial internal structure, resulting in the advent of the *PrutaxHBI* algorithm. An experimental evaluation and comparison of the proposed solution with currently available algorithms clearly shows that the proposed algorithm outperforms current algorithms under all circumstances.

1 Introduction

Mining of association rules is by far the most popular and widely used data mining technique. An association rule is an expression of the form $X \Rightarrow Y$, where X and Y are sets of items. An intuitive meaning of an association rule is that whenever an itemset X appears in a collection of itemsets, with a given probability, the itemset Y is also present. Application domains of association rule discovery range from market basket analysis, recommender systems, fraud detection, to numerous practical systems, e.g., insurance policies, investment portfolios, medical database analysis, and many more.

There are several efficient algorithms for mining association rules. Unfortunately, many researchers point to the fact, that association rules discovered in

the raw transactional data are useless for analysts and decision makers, because such rules are too detailed to be actionable or understandable. In several applications, items constituting mined itemsets are organized into a taxonomy. Such taxonomy can reflect a discretization of supermarket goods into product categories, division of books into genres, etc. A challenging, yet indispensable task is to incorporate item taxonomies into association rule mining process. However, currently available algorithms for association rule mining are not well-suited for this task.

In this paper we present an efficient algorithm that aims at generalized association rule discovery by incorporating the taxonomy of mined items into the physical indexing structure used by the algorithm. First, we present the hierarchical bitmap index, an indexing structure capable of efficient indexing of large sets with items drawn from huge domains. Next, we briefly describe *Prutax*, the best state-of-the-art algorithm for mining generalized association rules, and we show how we can significantly enhance *Prutax* by using the hierarchical bitmap index as the core physical structure for the algorithm. This leads to the development of the *PrutaxHBI* algorithm. A set of experiments proves the validity and efficiency of the proposed solution.

This paper is organized as follows. In Section 2 we present the related work. Basic definitions used throughout the paper are presented in Section 3. In Section 4 we describe the hierarchical bitmap index which is the core structure used in our algorithm. We present the *PrutaxHBI* algorithm in Section 5, and we report on the results of the experimental evaluation of our proposal in Section 6. The paper concludes in Section 7 with a summary and a future work agenda.

2 Related Work

The problem of association rule mining was first introduced in [2]. The paper identified the discovery of frequent itemsets as a key step in association rule mining. In [3] the authors introduced the *Apriori* algorithm, that quickly became the seed for numerous other association rule mining algorithms, e.g. [7]. In particular, the modification of the *Apriori* algorithm that allowed to mine generalized association rules was presented in [8]. The authors presented *Apriori Basic* algorithm that simply extended each database transaction with all ancestors of all items contained in the transaction. In addition, three optimizations of the original Apriori were proposed: *Cumulate*, *Stratify*, *Estimate*, and *EstMerge*. A similar direction has been followed in [9], where several new pruning strategies exploiting the taxonomy of items have been presented, and a new pruning strategy called *Genex* has been introduced. Another attempt to modify existing Apriori-based algorithms in the direction of efficient generalized association rule mining was presented in [4]. The authors present the family of ML-T* algorithms that mine associations between items from different levels of taxonomy, with minimum support thresholds varying between subsequent levels.

An entirely different approach is represented by the *Prutax* algorithm [5]. *Prutax* uses a vertical database layout and avoids unnecessary candidate item-

set generation by performing a depth-first traversal of the itemset lattice. Each candidate is evaluated immediately after generation and pruning is applied to remove candidate itemsets that contain both an item and its ancestor. In addition, *Prutax* enforces frequency ordering on items, thus directing the search through the itemset space from the most general itemsets to the most specific itemsets. However, these optimizations come at the cost of transforming the database to the vertical layout, which may be prohibitively expensive.

3 Basic Definitions

Let $I = \{i_1, \ldots, i_n\}$ be a set of literals called items. Let τ be a directed acyclic graph defining a taxonomy over the set I. An item i_p is the *parent* of a *child* item i_c if there exists an edge between vertices i_p and i_c in the graph τ. An item i_a is the *ancestor* of a *descendant* item i_d if there exists a path between vertices i_a and i_r in the graph τ. An item i_b is the *base item* if it has no descendants in the graph τ. Let D be a set of variable length transactions and $\forall T \in D : T \subseteq I \wedge \forall x \in T : x$ is a base item. We say that the transaction T supports an item x if $x \in T$. We say that the transaction T supports an itemset X if it supports every element $x \in X$. The *support* of an itemset is the number of transactions supporting the itemset. The problem of discovering frequent itemsets consists in finding all itemsets with the support higher than user-defined minimum support threshold denoted as *minsup*. An itemset with the support higher than *minsup* is called a *frequent itemset*.

An association rule is an expression of the form $X \rightarrow Y$ where $X \subset I$, $Y \subset I$, $X \cap Y = \emptyset$, and all items occurring in X and Y are base items. X is called the *body* of the rule whilst Y is called the *head* of the rule. Two measures represent statistical significance and strength of a rule. The *support* of a rule is the number of transactions that support $X \cup Y$. The *confidence* of a rule is the ratio of the number of transactions that support the rule to the number of transactions that support the head of the rule. The problem of discovering association rules consists in finding all rules with support and confidence higher than the user-specified thresholds of minimum support and confidence, called *minsup* and *minconf* respectively. Generalized association rules extend this base framework by allowing non-base items to appear as elements of rule body or head as well.

4 Hierarchical Bitmap Index

The hierarchical bitmap index (HBI) was first introduced in [6]. Hierarchical bitmap index is based on signature index framework. It employs the idea of exact set element representation and uses hierarchical structure to compact resulting signature and reduce its sparseness. The index on a given attribute consists of a set of index keys, each representing a single set. An example of an index key is depicted in Figure 1. Every index key comprises a signature divided into n-bit chunks (called *index key leaves*) and a set of *inner nodes* of the index key

organized into a tree structure. The highest inner node is called the *root* of the index key. The signature must be long enough to represent all possible elements appearing in the indexed set (usually hundreds of thousands of bits). Every element i_j of the attribute domain A, $i_j \in dom(A)$ is mapped to an integer i.

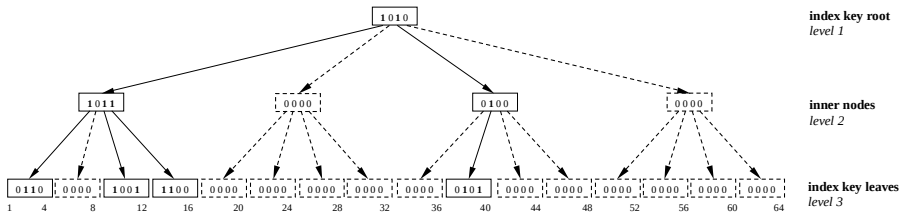

Fig. 1. Hierarchical bitmap index

Given an indexed set $S = \{i_1, i_2, \ldots, i_m\}$. The set is represented in the index in the following way. Let l denote the length of the index key node. An item $i_m \in S$ is represented by a '1' on the j-th position in the k-th index key leaf, where $k = \lceil m/l \rceil$ and $j = m - (\lceil m/l \rceil - 1) * l$. Therefore, the set S is represented by n '1's set at appropriate positions of the index key leaves. An index key node (either leaf node or inner node) which contains '1' on at least one position is called a *non-empty* node, while an index key node which contains '0' on all positions is called an *empty* node. The next level of the index key compresses the signature representing the set S by storing information only about the non-empty leaf nodes. A single bit in an inner node represents a single index key leaf. If this bit is set to '1' then the corresponding index key leaf contains at least one position set to '1'. The i-th index key leaf is represented by j-th position in the k-th inner index key node, where $k = \lceil i/l \rceil$ and $j = i - (\lceil i/l \rceil - 1) * l$. Every upper level of the inner nodes represents the lower level in an analogous way. This procedure repeats recursively to the index key root. The index key stores only the non-empty nodes (marked on Figure 1 with solid lines). Empty nodes (marked on Figure 1 with dashed lines) are not stored anywhere in the index key. In other words, index key leaves form an exact signature of the indexed set and subsequent levels represent coarser signatures of the indexed set.

Two parameters which affect shape and capacity of the index are: l — the length of a single index key node and d — the depth of the index key tree structure. HBI allows to index attributes with a domain up to l^d distinct items. An important factor that strongly affects the performance of the index is the mapping function. This function determines the mapping of items of the indexed domain on positions in the bitmap B of the HBI key. The feature that makes the hierarchical bitmap index suitable for generalized association rule mining is the fact that HBI makes no assumptions about mapping function chosen to map domain items on signature bit positions. One possible mapping function is hierarchical mapping.

Hierarchical mapping is performed by the function $f(i_j) = H(i_j)$. This mapping considers taxonomy τ defined over items. For every item i_j the function $H(i_j)$ returns the hierarchy category of the item. Item hierarchies are application-dependent and must be provided by the domain experts. Using hierarchical mapping the index not only represents the physical data contained in the indexed sets, but captures the logical properties of the indexed data as well. Hierarchical mapping allows to efficiently answer queries pertaining to higher logical level of the data without the need to physically store information about the taxonomy. To put it in other words, the taxonomy over items is physically encoded in the structure of the hierarchical bitmap index.

5 PrutaxHBI Algorithm

In this section we present modifications introduced to the original *Prutax* algorithm. One thing to note is the fact that *Prutax* operates on the vertical database layout. A crucial operation during *Prutax* execution is the join of transaction identifier (*tid*) lists pertaining to different items. The resulting list contains *tid*s of transactions containing both joined items, therefore, the length of the joined list is simply the support of the itemset consisting of joined items. As *Prutax* operates in the depth-first direction, the join of long *tid* lists is performed many many times. Potentially, every optimization of this expensive process could result in huge savings in algorithm's running time. Hierarchical bitmap index is very well suited to represent large sets of items. In this case, we've decided to transform *tid* lists into hierarchical bitmap index keys, one key per item. Then, instead of joining original *tid* lists, we significantly speed up this operation by performing it directly on hierarchical bitmap index keys.

Building of hierarchical bitmap index keys is performed iteratively. The main parameter governing the building phase is the size of the memory buffer allocated to the process. Based on the available buffer space a set of items is chosen for which hierarchical bitmap index keys will be computed during single iteration. Each iteration performs a single database scan in search of transactions that support any item being processed in current iteration. After the database scan is completed, all hierarchical bitmap index keys are created and written to file. While building hierarchical bitmap index keys for items assigned to the current iteration, in parallel we create hierarchical bitmap index keys for their ancestors. This requires on-the-fly extension of each transaction with ancestors of all items contained in the transaction.

Finally, hierarchical bitmap index keys are computed only for single items. During the execution of the algorithm, several hierarchical bitmap index keys are created dynamically to represent sets of items. The number of hierarchical bitmap index keys created is quite large, and some keys might be re-used for computing the support of their supersets. Unfortunately, writing these intermediate results back to the index file is extremely expensive and significantly slows down the algorithm. On the other hand, simply discarding these results wastes computational effort undertaken to create these index keys. In our implementa-

Table 1. Computing the cardinality of intersection of multiple HBIs

```
int intersect(HBI[] hbis) {
  BitSet common = new BitSet();
  common.set(0, nodeSize, true)       // set '1' for all positions
  for (h: hbis) {
    common.and(h.getLevel(0)); }
  if (common.cardinality() == 0) return 0
  BitSet[] omit = new BitSet[hbis.size()];
  for (int i = 0; i < hbis.length; i++) {
    omit[i] = new BitSet();
    ∀ k: omit[i].set(k, true) if common.get(k) == false; }
  int currentLevel = 1;
  while (true) {
    set all bits in common to '1';
    for (int i = 0; i < hbis.length; i++) {
      BitSet currentCommonLevel = new BitSet();
      copy to currentCommonLevel all nodes from hbis[i].getLevel(currentLevel)
      if respective bit in omit[i] is set to '0';
      common.and(currentCommonLevel);
    }
    if (common.cardinality() == 0) return 0;
    for (int i = 0; i < hbis.length; i++) {
      BitSet newOmit = new BitSet();
      for (int j = 0; j < hbis[i].getLevel(currentLevel).cardinality(); j++) {
        set newOmit[i].set(k,true) if k-th bit belongs to a node, whose parent was
        set to '1' in omit[i] or respective bit in common is set to '0';
      }
      ommit[i] = newOmmit;
    }
    currentLevel++;
    if (currentLevel == depth) return common.cardinality();
  }
}
```

tion we have chosen not to store the intermediate results in the index file, but we put dynamically created hierarchical bitmap index keys in an LRU-managed memory buffer. Our experiments clearly indicate, that the utilization of even a small buffer may have tremendous impact on the performance of the *PrutaxHBI* algorithm.

In Table 1 we present the pseudo-code of the core function that computes the cardinality of the set of transactions supporting a given candidate itemset based on hierarchical bitmap index keys representing items contained in the candidate itemset. It is worth noticing, that the function does not have to actually determine the set of supporting transactions, it is sufficient to compute its cardinality. For a given candidate $(k+1)$-itemset C, we assume it has been generated from two frequent (k)-itemsets H_i and H_j that share a common $(k-1)$-prefix. The `intersect(HBI[])` function, which computes the support of the candidate

itemset C, takes as input an array of $k+1$ hierarchical bitmap index keys, one for each item contained in the candidate itemset C. The function iteratively computes the binary intersection of hierarchical bitmap index keys on each level of the hierarchical bitmap index. Hierarchical bitmap index keys are bitwise intersected until the leaf level is reached, or the intersection becomes void. The final result of the function `intersect(HBI[])` is the number of bits set to '1' in the intersection of all compared hierarchical bitmap index keys. During implementation a library class `java.util.BitSet` has been used to represent both leaves and internal nodes of hierarchical bitmap index keys. This base class was sub-classed to allow quick serialization and writing of bit vectors.

6 Experiments

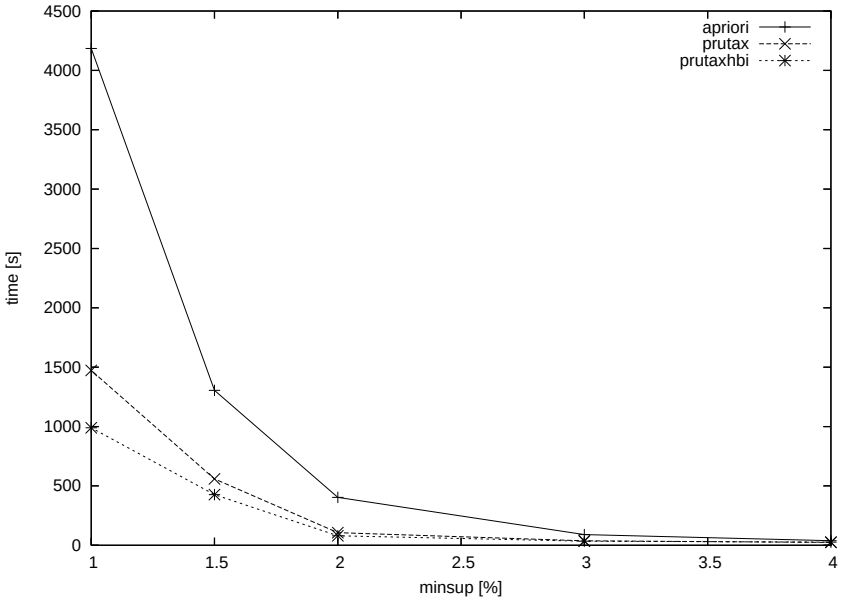

Fig. 2. Running times of algorithms when varying the *minsup* threshold

In this section we report on the results of the experimental evaluation of the *PrutaxHBI* algorithm. We compare our algorithm with *Apriori Cumulate* [8] and *Prutax* [5]. All experiments were conducted on a computer with two Dual Core AMD Opteron 1 Ghz processors and 8 GB RAM running under Linux 2.6.9 operating system. Data sets were created using DBGen generator from the Quest Project [1]. Input data were stored in flat transactional format using a simple schema of `<transaction_id,item_id>`. For the *Prutax* algorithm input data have been transformed into a vertical database layout of `<item_id,`*`list`*

of transactions and the transformation time has been added to algorithm's running times. Likewise, the time needed to construct a hierarchical bitmap index for the *PrutaxHBI* algorithm has been included in the results. The taxonomy of items has been synthetically generated after base items had been created by DBGen.

Figure 2 presents running times of the three algorithms when varying the *minsup* threshold. In our experiment the *minsup* threshold changes from 1% to 4%. Presented results are averaged over five different datasets of 10 000 transactions each, with the average transaction length set to 8 and the average frequent itemset size set to 3. The number of items in the database was set to 100 000. As can be seen from the figure, the *PrutaxHBI* algorithm outperforms both *Apriori Cumulate* and *Prutax*, with the gain greater for low minimum support thresholds. We attribute this behavior to the fact that lower minimum support thresholds induce more frequent itemsets, which in turn increases the profit of hierarchical bitmap indexing of base transactions.

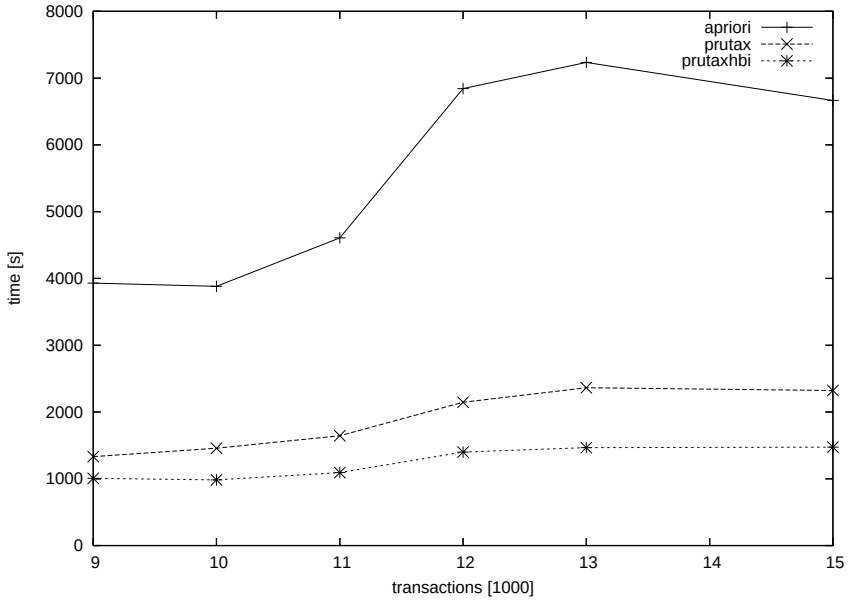

Fig. 3. Running times of algorithms when varying number of transactions

Figure 3 shows the scalability of the proposed algorithm. The results are averaged over five independent datasets. The *minsup* threshold was constantly set to 1% and the number of different items was set to 100 000. What is apparent from the figure is the fact, that using the taxonomy to prune candidate itemsets (as *Prutax* and *PrutaxHBI* algorithms do) significantly improves the performance. Furhtermore, both algorithms scale better than the original *Apriori* algorithm.

When increasing the size of the database from 9 000 to 11 000, the running time of *Apriori* grew by 84% while the running time of *Prutax* grew only by 46%. We are glad to note that the PrutaxHBI algorithm outperforms the original Prutax for all database sizes.

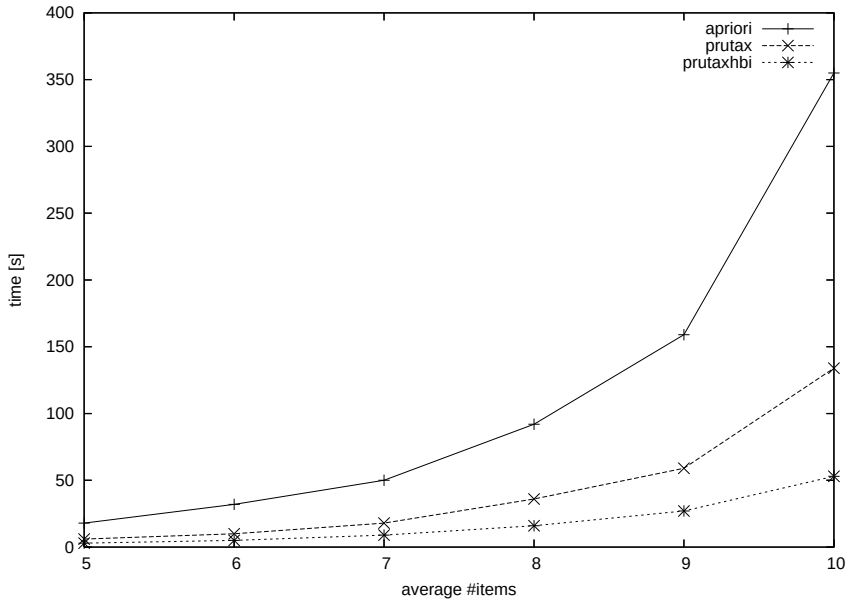

Fig. 4. Running times of algorithms when varying average transaction size

In the next experiment we have investigated running times of algorithms under varying average transaction size. The results depicted in Figure 4 are averaged over six datasets of 1000 transactions each. The number of distinct items was set to 10 000 and the *minsup* threshold was set to 2%. The average transaction size varied from five items to ten items. The results clearly show that all algorithms decrease performance while increasing the average transaction size. Again, our *PrutaxHBI* algorithm outperforms the other two algorithms, with *Apriori Cumulate* being the most sensitive to the average number of items in a transaction. The explanation of the result is straighforward. Larger transactions imply larger database file to be processed during each iteration of the *Apriori Cumulate* algorithm. Both *Prutax* and *PrutaxHBI* utilize a compressed vertical database layout, therefore the increase of the average transaction size has lesser impact on them. Furthermore, for *Apriori Cumulate* larger transactions require extending transactions with more ancestor items, which has a direct influence on the performance of the algorithm.

The last experiment concerns the number of patterns in the mined database. The results depicted in Figure 5 are averaged over a set of database files with 1000

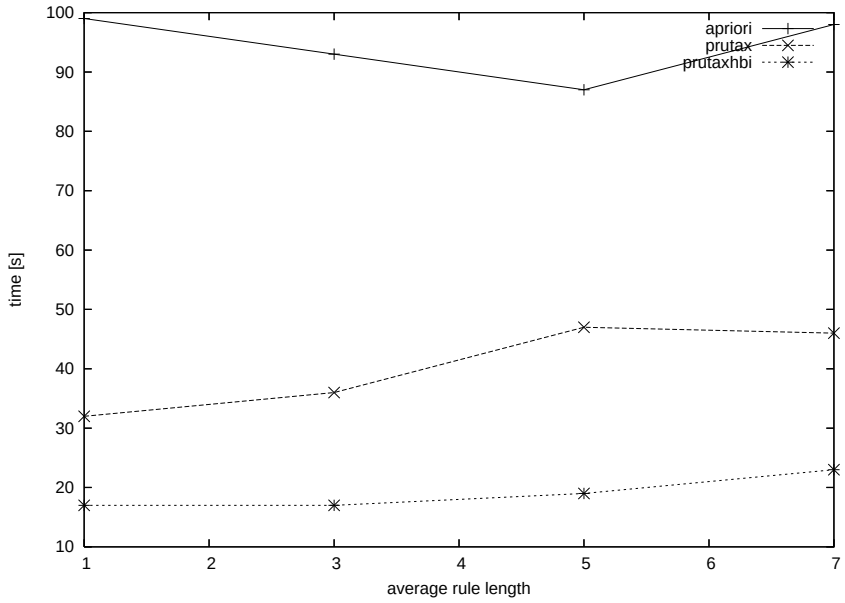

Fig. 5. Running times of algorithms when varying average itemset size

transactions each. The number of distinct items was set to 10 000 and the *minsup* threshold was set to 2%. The average frequent itemset size varied from one to seven. The variance in running times of *Apriori Cumulate* is probably random, because it should not affect the algorithm at all (recall that we are changing the average frequent itemset size, which does not prohibit the existance of shorter or longer itemsets in the database). Both *Prutax* and *PrutaxHBI* slightly worsen performance for larger frequent itemsets due to the increasing depth at which the candidate itemset graph must be traversed. Nevertheless, our algorithm still significantly outperforms competitors.

7 Conclusions

In this paper we have presented a new approach to generalized association rule mining that uses the *Prutax* algorithm and the hierarchical bitmap index structure. Our *PrutaxHBI* algorithm outperforms state-of-the-art algorithms for mining generalized association rules. Experiments conducted on synthetic datasets prove the efficiency of the proposed solution.

Certainly, the work presented in this paper may be extended in several directions. There are numerous tweaks of the *Prutax* algorithm that might increase the performance, most notably, caching of intermediate results might prove useful. Our future work agenda includes, among others, improving the integration of *Prutax* and the hierarchical bitmap index, verifying the top-down breadth-first

approach of generating candidate itemsets using the hierarchical bitmap index, and applying hierarchical bitmap indexing technique to other data mining problems.

References

1. R. Agrawal, M. J. Carey, C. Faloutsos, S. P. Ghosh, M. A. W. Houtsma, T. Imielinski, B. R. Iyer, A. Mahboob, H. Miranda, R. Srikant, and A. N. Swami. Quest: A project on database mining. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, page 514, Minneapolis, Minnesota, may 1994. ACM Press.
2. R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 207–216. ACM Press, 1993.
3. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.
4. J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *VLDB'95, Proceedings of 21st International Conference on Very Large Data Bases, Zürich, Switzerland, September 1995*, pages 420–431, 1995.
5. J. Hipp, A. Myka, R. Wirth, and U. Güntzer. A new algorithm for faster mining of generalized association rules. In *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery (PKDD '98)*, pages 74–82, Nantes, France, September 23-26 1998.
6. M. Morzy, T. Morzy, A. Nanopoulos, and Y. Manolopoulos. Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes. In L. A. Kalinichenko, R. Manthey, B. Thalheim, and U. Wloka, editors, *Advances in Databases and Information Systems, 7th East European Conference, ADBIS 2003, Dresden, Germany, September 3-6, 2003, Proceedings*, volume 2798 of *Lecture Notes in Computer Science*, pages 236–252. Springer, 2003.
7. R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 1–12, Montreal, Quebec, Canada, 1996.
8. R. Srikant and R. Agrawal. Mining generalized association rules. *Future Generation Computer Systems*, 13(2–3):161–180, 1997.
9. I. Weber. On pruning strategies for discovery of generalized and quantitative association rules. In L. Bing, W. Hsu, and W. Ke, editors, *PRICAI'98, Proceedings of Knowledge Discovery and Data Mining Workshop, Singapore, November 1998*, 1998.