FP-Bonsai: the Art of Growing and Pruning Small FP-Trees

Francesco Bonchi[1] and Bart Goethals[2]

[1] Pisa KDD Laboratory, ISTI - CNR, Area della Ricerca di Pisa, Italy
[2] HIIT - BRU, Dept. of Computer Science, University of Helsinki, Finland

Abstract. In the context of mining frequent itemsets, numerous strategies have been proposed to push several types of constraints within the most well known algorithms. In this paper, we integrate the recently proposed ExAnte data reduction technique within the FP-growth algorithm. Together, they result in a very efficient frequent itemset mining algorithm that effectively exploits monotone constraints.

1 Introduction

The problem of how to push different types of constraints into the frequent itemsets computation has been extensively studied [5, 6, 3]. However, while pushing anti-monotone constraints deep into the mining algorithm is easy and effective, the case is different for monotone constraints. Indeed, anti-monotone constraints can be used to effectively prune the search space to a small downward closed collection, while the upward closed collection of the search space satisfying the monotone constraints cannot be pruned at the same time. Recently, it has has been shown that a real synergy of these two opposite types of constraints exists and can be exploited by reasoning on both the itemset search space and the input database *together*, using the ExAnte data-reduction technique [2]. This way, pushing monotone constraints does not reduce anti-monotone pruning opportunities, but on the contrary, such opportunities are boosted. Dually, pushing anti-monotone constraints boosts monotone pruning opportunities: the two components strengthen each other recursively. This idea has been generalized in an Apriori-like computation in ExAMiner [1].

In this paper we show how this synergy can be exploited even better within the well known FP-growth algorithm [4]. Thanks to the recursive projecting approach of FP-growth, the ExAnte data-reduction is pervasive all over the computation. All the FP-trees built recursively during the FP-growth computation can be pruned extensively by using the ExAnte property, obtaining a computation with a smaller number of smaller trees. We call such a tiny FP-tree, obtained by growing and pruning, an *FP-bonsai*.

The resulting method overcomes on one hand the main drawback of FP-growth, which is its memory requirements, and on the other hand, the main drawback of ExAMiner which is the I/O cost of iteratively rewriting the reduced datasets to disk.

2 Preliminaries

Let $\mathcal{I} = \{x_1, ..., x_n\}$ be a set of *items*. An *itemset* X is a non-empty subset of $\mathcal{I}$. A *transaction* is a couple $\langle tid, X \rangle$ where *tid* is the transaction identifier and X is an itemset. A *transaction database* $\mathcal{D}$ is a set of transactions. An itemset X is contained in a transaction $\langle tid, Y \rangle$ if $X \subseteq Y$. The *support* of an itemset X in database $\mathcal{D}$, denoted by $supp_{\mathcal{D}}(X)$ is the number of transactions in $\mathcal{D}$ that contain X. Given a user-defined *minimum support* σ, an itemset X is called *frequent* in $\mathcal{D}$ if $supp_{\mathcal{D}}(X) \geq \sigma$. The *frequent itemset mining problem* requires to compute the set of all frequent itemsets.A constraint on itemsets is a function $\mathcal{C} : 2^{\mathcal{I}} \rightarrow \{true, false\}$. We say that an itemset I satisfies a constraint if and only if $\mathcal{C}(I) = true$. Let $Th(\mathcal{C}) = \{X \mid \mathcal{C}(X) = true\}$ denote the set of all itemsets X that satisfy constraint $\mathcal{C}$. In general given a conjunction of constraints $\mathcal{C}$ the the *constrained frequent itemsets mining problem* requires to compute $Th(\mathcal{C}_{freq}) \cap Th(\mathcal{C})$, where $\mathcal{C}_{freq}$ is the frequency constraint.

In particular we focus on two kinds of constraint: a constraint $\mathcal{C}_{AM}$ is *anti-monotone* if $\mathcal{C}_{AM}(X) \Rightarrow \mathcal{C}_{AM}(Y)$ for all $Y \subseteq X$; a constraint $\mathcal{C}_M$ is *monotone* if: $\mathcal{C}_M(X) \Rightarrow \mathcal{C}_M(Y)$ for all $Y \supseteq X$. Since any conjunction of anti-monotone constraints is an anti-monotone constraint, and any conjunction of monotone constraints is a monotone constraint, we consider without loss of generality the conjunction $\mathcal{C}_{freq} \cap Th(\mathcal{C}_M)$ where $\mathcal{C}_M$ is a simple monotone constraint such as $sum(X.prices) \geq n$.

The recently introduced ExAnte method [2] exploits monotone constraints in order to to reduce the input database and thus to prune the search space. This method is based on the synergy of the following two data-reduction operations: (1) μ*-reduction*, which deletes transactions in $\mathcal{D}$ which do not satisfy $\mathcal{C}_M$; and (2) α*-reduction*, which deletes from all transactions in $\mathcal{D}$ singleton items which do not satisfy $\mathcal{C}_{freq}$. The ExAnte property states that a transaction which does not satisfy the given monotone constraint can be deleted from the input database (μ-reduction) since it will never contribute to the support of any itemset satisfying the constraint. A major consequence of reducing the input database in this way is that it implicitly reduces the support of a large amount of itemsets. As a consequence, some singleton items can become infrequent and can not only be removed from the computation, but they can be deleted from all transactions in the input database (α-reduction). This removal also has another positive effect. That is, the reduced transaction might violate the monotone constraint. Obviously, we are inside a loop where two different kinds of pruning (α and μ) cooperate to reduce the search space and the input dataset, strengthening each other step by step until no more pruning is possible (a fix-point has been reached). This is the key idea of the ExAnte preprocessing method [2]. In the end, the reduced dataset resulting from this fix-point computation is usually much smaller than the initial dataset

Given a transaction database $\mathcal{D}$, a conjunction of monotone constraints $\mathcal{C}_M$, and a conjunction of anti-monotone constraints $\mathcal{C}_{AM}$, we define the reduced dataset obtained by the fix-point application of μ and α pruning as: $\mu^{+}_{\mathcal{C}_{AM},\mathcal{C}_M}(\mathcal{D})$.

3 FP-Bonsai

The FP-growth algorithm [4] stores the actual transactions from the database in a trie structure (prefix tree), and additionally stores a header table containing all items with their support and the start of a linked list going through all transactions that contain that item. This data structure is denoted by *FP-tree* (Frequent-Pattern tree) [4]. For example, consider the transaction database in Figure 2(a) and a minimal support threshold of 4. First, all infrequent items are removed from the database, all transactions are reordered in support descending order and inserted into the FP-tree, resulting in the tree in Figure 2(b).

Given a transaction database $\mathcal{D}$ and a minimal support threshold σ, we denote the set of all frequent itemsets with the same prefix $I \subseteq \mathcal{I}$ by $\mathcal{F}[I](\mathcal{D}, \sigma)$. FP-growth recursively generates for every singleton item $\{i\} \in Th(\mathcal{C}_{freq})$ the set $\mathcal{F}[\{i\}](\mathcal{D}, \sigma)$ by creating the so called *i-projected* database of $\mathcal{D}$. This database, denoted $\mathcal{D}^i$, is made of all transactions in $\mathcal{D}$ containing i, from which i and all items which come before i, w.r.t. the support descending order, are deleted. This *i-projected* database, which is again stored as an FP-tree, is recursively mined by FP-growth. The FP-growth algorithm is shown in Figure 1.

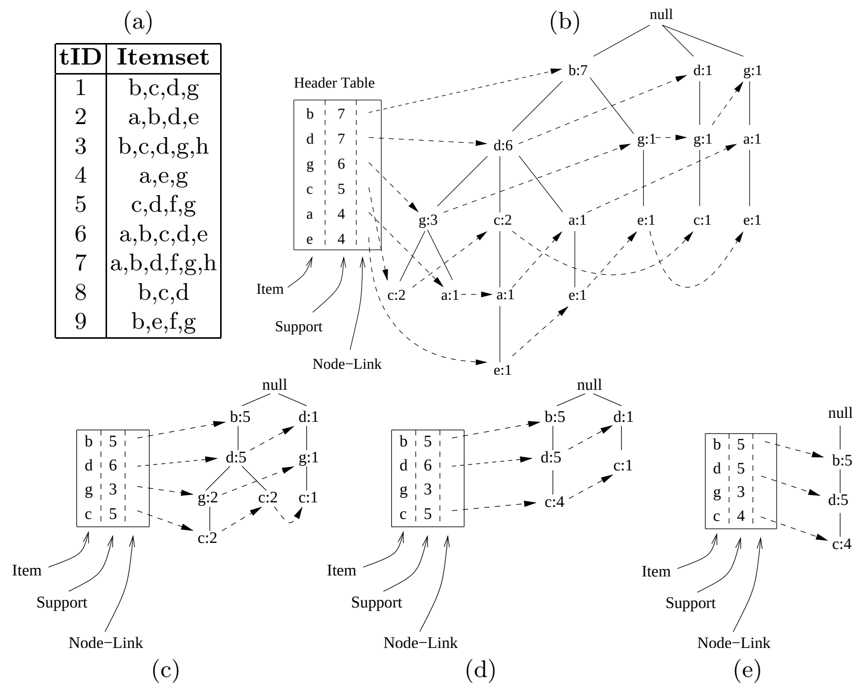
```
Algorithm FP-growth
Input: $\mathcal{D}, \sigma, I \subseteq \mathcal{I}$
Output: $\mathcal{F}[I](\mathcal{D}, \sigma)$
  $\mathcal{F}[I] := \{\}$
  for all $i \in \mathcal{I}$ occurring in $\mathcal{D}$ do
    $\mathcal{F}[I] := \mathcal{F}[I] \cup \{I \cup \{i\}\}$
    $H := \{\}; \mathcal{D}^i := \{\}$
    for all $j \in \mathcal{I}$ occurring in $\mathcal{D}$ such that
    $j > i$ do
      if $supp_{\mathcal{D}}(I \cup \{i, j\}) \geq \sigma$ then
        $H := H \cup \{j\}$
    for all $(tid, X) \in \mathcal{D}$ with $i \in X$ do
      $\mathcal{D}^i := \mathcal{D}^i \cup \{(tid, X \cap H)\}$
    Compute $\mathcal{F}[I \cup \{i\}](\mathcal{D}^i, \sigma)$
    $\mathcal{F}[I] := \mathcal{F}[I] \cup \mathcal{F}[I \cup \{i\}]$
```

```
Algorithm FP-pruning
Input: $\mathcal{D}, \mathcal{C}_{AM}, \mathcal{C}_M, I$
Output: $\mu^+_{\mathcal{C}_M, \mathcal{C}_{AM}}[\mathcal{D}]$
  repeat
    // $\mu$-pruning of $\mathcal{D}$
    for all transactions $t$ occurring in $\mathcal{D}$
    do
      if $\mathcal{C}_M(I \cup t) = false$ then
        Remove $t$ from $\mathcal{D}$
    // $\alpha$-pruning of $\mathcal{D}$
    for all items $i$ occurring in $\mathcal{D}$ do
      if $\mathcal{C}_{AM}(I \cup \{i\}) = false$ then
        Remove $i$ from $\mathcal{D}$
  until nothing changed
```

Fig. 1. The FP-growth and FP-pruning algorithms.

The main trick exploited in FP-growth is that it only needs to find all singleton frequent itemsets in the given database. Then, for every such item, it creates the corresponding projected database in which again, only the (local) singleton frequent itemsets have to be found. This process goes on until no more (local) items exist. The FP-tree structure guarantees that all this can be done efficiently. In this way, FP-growth implicitly creates a lot of databases, represented by FP-trees. The good news is that all these datasets (trees) can be reduced (pruned) using the ExAnte technique. We call such a pruned FP-tree an *FP-bonsai.*

The FP-pruning procedure is shown in Figure 1. In order to obtain the complete algorithm that finds all itemsets satisfying the given constraints, the FP-pruning algorithm should be called before the first line of the FP-growth algorithm. The fact that the database is stored as an FP-tree is not specifically mentioned. That is because this is actually not necessary, but the FP-tree is simply the most effective data structure for these algorithms to use. How the pruning mechanisms can be effectively applied on the FP-tree structure is described by the following Example.



tID	Itemset
1	b,c,d,g
2	a,b,d,e
3	b,c,d,g,h
4	a,e,g
5	c,d,f,g
6	a,b,c,d,e
7	a,b,d,f,g,h
8	b,c,d
9	b,e,f,g

Fig. 2. A transaction database (a), the corresponding FP-tree for minimal support threshold of 4 (b), the initial FP-bonsai for Example 1 (c), the FP-bonsai after the removal of item g (d), and (e) the final FP-bonsai.

Example 1. Consider again the transactional database in Figure 2(a) and the price-table: $\{a : 5, b : 8, c : 14, d : 30, e : 20, f : 15, g : 6, h : 12\}$. Suppose that we want to compute itemsets with support no less than 4 and sum of prices no less than 45. The FP-bonsai construction starts with a first scan of $\mathcal{D}$ to count the support of all singleton items. Note that transaction 4 is not used since it does not satisfy the monotone constraint. This causes item a and e to be infrequent and are not included in the header table. Frequent items are ordered in support descending order and the tree is built as seen in Figure 2(c). At this point we find that the item g is no longer frequent than we remove all its occurrences in the tree using the link-node structure. The resulting pruned tree is in Figure 2(d). This α-pruning has created a new opportunity for μ-pruning. in fact, the path on the right edge of the tree does no longer satisfy the monotone constraint and hence it can be removed from the tree. In Figure 2(e) we have the final FP-bonsai (the fix-point has been reached) for the given problem. Note the the final size of the FP-bonsai is 3 nodes (which represents the unique solution to the given problem: itemset *bcd* with *support* = 4 and sum of prices = 52, while the size of the usual FP-tree for the same problem (Figure 2(b)) is 18 nodes!

Once the FP-bonsai has been built (i.e. once the fix-point of α and μ pruning has been reached) we can efficiently mine all frequent itemsets satisfying the given constraints using FP-growth. Thanks to the recursive structure of the FP-growth based algorithm, the ExAnte property is deeply amalgamated with the frequent itemset computation: not only the initial tree is a pruned tree (an FP-bonsai), but also all the other projected trees, built during the recursive growing phase will be much more smaller in number and in size.

The reduction of number of trees built w.r.t. FP-growth (which is the computation with minimum sum of prices = 0) is dramatic, as illustrated in Figure 3(a).

Our experimental study confirms that FP-bonsai outperforms ExAMiner, which is the state-of-the-art algorithm for the computational problem addressed in many occasions.

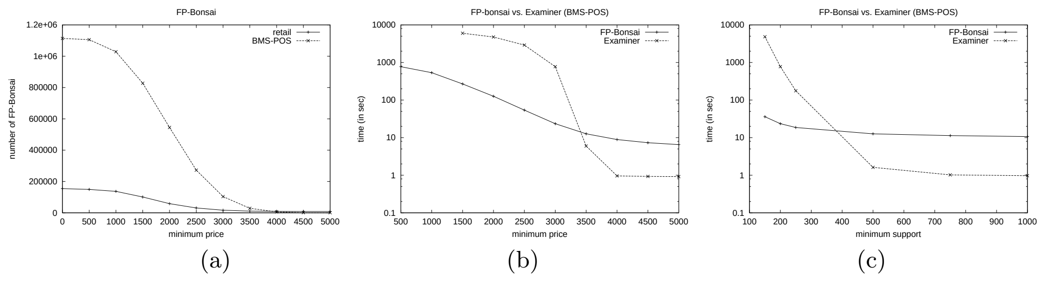


Fig. 3. Number of FP-bonsai built with fixed *minimum_support* and moving monotone threshold (a); run time of ExAMiner and FP-bonsai on dataset *BMS-POS* with *minimum_support* = 200 (b); and minimum sum of prices = 3000 (c).

From those pictures we can see that ExAMiner is faster only when the selectivity of one of the two constraints is very strong, and hence the set of solutions very small. In particular, ExAMiner is faster in recognizing when the user-defined constraints are so selective that the problem has an empty set of solutions. But in all the other cases FP-bonsai performs much better. In particular, when one of the two constraints is not-so-selective, FP-bonsai exhibits a much more stable behavior, while ExAMiner's computation time increases quickly. Consider, for instance Figure 3(c): at an absolute minimum support of 150, FP-bonsai takes 36 seconds against the 4841 seconds (1 hour and 20 minutes) taken by ExAMiner.

References

1. F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. ExAMiner: Optimized level-wise frequent pattern mining with monotone constraints. In *Proc. of ICDM'03*.
2. F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. Exante: Anticipated data reduction in constrained pattern mining. In *Proc. of PKDD03*.
3. C. Bucila, J. Gehrke, D. Kifer, and W. White. DualMiner: A dual-pruning algorithm for itemsets with constraints. In *Proc. of ACM SIGKDD'02*.
4. J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc of ACM SIGMOD'00*.
5. R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. of ACM SIGMOD'98*.
6. J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent item sets with convertible constraints. In *Proc. of ICDE'01*.