Mining Frequent Itemsets using Patricia Tries [*]

Andrea Pietracaprina and Dario Zandolin
Department of Information Engineering
University of Padova
andrea.pietracaprina@unipd.it, dzandol@tin.it

Abstract

We present a depth-first algorithm, PatriciaMine, that discovers all frequent itemsets in a dataset, for a given support threshold. The algorithm is main-memory based and employs a Patricia trie to represent the dataset, which is space efficient for both dense and sparse datasets, whereas alternative representations were adopted by previous algorithms for these two cases. A number of optimizations have been introduced in the implementation of the algorithm. The paper reports several experimental results on real and artificial datasets, which assess the effectiveness of the implementation and show the better performance attained by PatriciaMine with respect to other prominent algorithms.

1. Introduction

In this work, we focus on the problem of finding *all frequent itemsets* in a dataset $\mathcal{D}$ of transactions over a set of items $\mathcal{I}$, that is, all itemsets $X \subseteq \mathcal{I}$ contained in a number of transactions greater than or equal to a certain given threshold [2].

Several algorithms proposed in the literature to discover all frequent itemsets follow a depth-first approach by considering one item at a time and generating (recursively) all frequent itemsets which contain that item, before proceeding to the next item. A prominent member of this class of algorithms is FP-Growth proposed in [7]. It represents the dataset $\mathcal{D}$ through a standard trie (*FP-tree*) and, for each frequent itemset X, it materializes a projection $\mathcal{D}_X$ of the dataset on the transactions containing X, which is used to recursively discover all frequent supersets $Y \supset X$. This approach is very effective for dense datasets, where the trie achieves high compression, but becomes space inefficient when the dataset is sparse, and incurs high costs due to the frequent projections.

[*]This research was supported in part by MIUR of Italy under project "ALINWEB: Algorithmics for Internet and the Web".

Improved variants of FP-Growth appeared in the literature, which avoid physical projections of the dataset (Top-Down FP-Growth [14]), or employ two alternative array-based and trie-based structures to cope, respectively, with sparse and dense datasets, switching adaptively from one to the other (H-mine [12]). The most successful ideas developed in these works have been gathered and further refined in OpportuneProject [9] which opportunistically selects the best strategy based on the characteristics of the dataset.

In this paper, we present an algorithm, PatriciaMine, which further improves upon the aforementioned algorithms stemmed from FP-Growth. Our main contribution is twofold:

- We use a compressed (Patricia) trie to store the dataset, which provides a space-efficient representation for both sparse and dense datasets, without resorting to two alternative structures, namely array-based and trie-based, as was suggested in [12, 9]. Indeed, by featuring a smaller number of nodes than the standard trie, the Patricia trie exhibits lower space requirements, especially in the case of sparse datasets, where it becomes comparable to the natural array-based representation, and reduces the amount of bookkeeping operations. Both theoretical and experimental evidence of these facts is given in the paper.
- A number of optimizations have been introduced in the implementation of PatriciaMine. In particular, a heuristic has been employed to limit the number of physical projections of the dataset during the course of execution, with the intent to avoid the time and space overhead incurred by projection, when not beneficial. Moreover, novel mechanisms have been developed for directly generating groups of itemsets supported by the same subset of transactions, and for visiting the trie without traversing individual nodes multiple times. The effectiveness of these optimizations is discussed in the paper.

We coded PatriciaMine in C, and compared its performance with that of a number of prominent algorithms, whose source/object code was made available to us, on several real and artificial datasets. The experiments provide clear evidence of the higher performance of PatriciaMine with respect to these other algorithms on both dense and sparse datasets. It must be remarked that our focus is on main-memory execution, in the sense PatriciaMine works under the assumption that the employed representation of the dataset fits in main memory. If this is not the case, techniques such as those suggested in [13, 9] could be employed, but this is beyond the scope of this work.

The rest of the paper is organized as follows. Section 2 introduces some notation and illustrates the datasets used in the experiments. Section 3 presents the main iterative strategy adopted by PatriciaMine, which can be regarded as a reformulation (with some modifications) of the recursive strategies adopted in [7, 12, 14, 9]. Sections 4 and 5 describe the most relevant features of the algorithm implementation, while the experimental results are reported and discussed in Section 6.

2. Preliminaries

Let $\mathcal{I}$ be a set of *items*, and $\mathcal{D}$ a set of *transactions*, where each transaction $t \in \mathcal{D}$ consists of a distinct identifier t_{id} and a subset of items $t_{\mathrm{set}} \subseteq \mathcal{I}$. For an *itemset* $X \subseteq \mathcal{I}$, its *support* in $\mathcal{D}$, denoted by $\mathrm{supp}_{\mathcal{D}}(X)$, is defined as the number of transactions $t \in \mathcal{D}$ such that $X \subseteq t_{\mathrm{set}}$. Given an absolute support threshold *min_sup*, with $0 < \mathrm{min\_sup} \leq |\mathcal{D}|$, an itemset $X \subseteq \mathcal{I}$ is *frequent w.r.t.* $\mathcal{D}$ and min_sup, if $\mathrm{supp}_{\mathcal{D}}(X) \geq \mathrm{min\_sup}$. With a slight abuse of notation, we call an item $i \in \mathcal{I}$ *frequent* if $\{i\}$ is frequent, and refer to $\mathrm{supp}_{\mathcal{D}}(\{i\})$ as the support of i[1]. We study the problem of determining the set of all frequent itemsets for given $\mathcal{D}$ and min_sup, which we denote by $\mathcal{F}(\mathcal{D}, \mathrm{min\_sup})$. For an itemset $X \subseteq \mathcal{I}$, we denote by $\mathcal{D}_X$ the subset of $\mathcal{D}$ projected on those transactions that contain X.

Let $\mathcal{I}' = \{i_1, i_2, \ldots\} \subseteq \mathcal{I}$ denote the subset of frequent items ordered by increasing support, and assume that the items in each frequent itemset are ordered accordingly. As observed in [1, 9], the set $\mathcal{F}(\mathcal{D}, \mathrm{min\_sup})$ can be conveniently represented through a standard trie [8], called *Frequent ItemSet Tree* (FIST), whose nodes are in one-to-one correspondence with the frequent itemsets. Specifically, each node v is labelled with an item i and a support value σ_v, so that the itemset associated with v is given by the sequence of items labelling the path from the root to v, and has support σ_v. The root is associated with the empty itemset and is labelled with $(\cdot, |\mathcal{D}|)$. The children of every node are arranged right-to-left consistently with the ordering of their labelling items.

[1]When clear from the context, we will refer to frequent items or itemsets, omitting $\mathcal{D}$ and min_sup.

TID	Items
1	**A** **B** **D** E **F** **G** **H** I
2	**B** C E **L**
3	**A** **B** **D** **F** **H** **L**
4	**A** **B** C **D** **F** **G** **L**
5	**B** **G** **H** **L**
6	**A** **B** **D** **F** I

Figure 1. Sample dataset (items in bold are frequent for min_sup $= 3$)

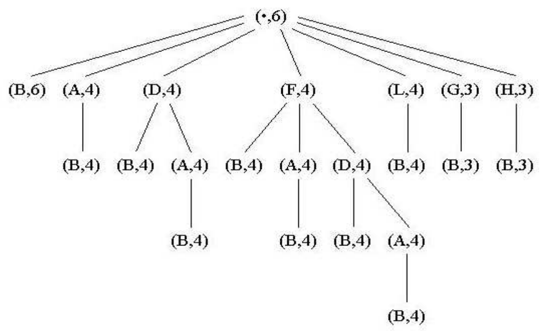

Figure 2. FIST for the sample dataset with min_sup $= 3$

A sample dataset and the corresponding FIST for $\mathrm{min\_sup} = 3$ are shown in Figures 1 and 2. Notice that a different initial ordering of the items in $\mathcal{I}'$ would produce a different FIST. Most of the algorithms that compute $\mathcal{F}(\mathcal{D}, \mathrm{min\_sup})$ perform either a breadth-first or a depth-first exploration of some FIST. In particular, our algorithm performs a depth-first exploration of the FIST defined above.

2.1. Datasets used in the experiments

The experiments reported in this paper have been conducted on several real and artificially generated datasets, frequently used in previous works. We briefly describe them below and refer the reader to [4, 16] for more details (see also Table 1).

Pos: From Blue-Martini Software Inc., contains years worth of point-of-sale data form an electronics retailer.

WebView1, WebView2: From Blue-Martini Software Inc., contain several months of clickstream data from e-commerce web sites.

Pumsb, Pumsb*: derived by [4] from census data.

Mushroom: It contains characteristics of various species of mushrooms.

Connect-4, Chess: are relative to the respective games.

```
1. Determine $\mathcal{I}'$ and $\mathcal{D}'$;
2. Create IL and link it to $\mathcal{D}'$;
   $X \leftarrow \emptyset$; $h \leftarrow 0$; $\ell \leftarrow 0$;
   while ($\ell < |\text{IL}|$) do
3.     if (IL[$\ell$].count $<$ min_sup) then $\ell \leftarrow \ell + 1$;
       else
4.         if (($h > 0$) AND (IL[$\ell$].item $= X[h-1]$))
5.         then $\ell \leftarrow \ell + 1$; $h \leftarrow h - 1$;
           else
6.             $X[h] \leftarrow$ IL[$\ell$].item;
7.             $h \leftarrow h + 1$;
8.             Generate itemset $X$;
9.             for $i \leftarrow \ell - 1$ downto 0 do
                   make IL[$i$].ptr point to head of t-list($i, \mathcal{D}'_X$);
                   IL[$i$].count $\leftarrow$ support of IL[$i$].item in $\mathcal{D}'_X$;
               $\ell \leftarrow 0$;
```

Figure 3. Main Strategy

IBM-Artificial: a class of artificial datasets obtained using the generator developed in [3]. A dataset in this class is denoted through the parameters used by the generator, namely as Dx.Ty.Iw.Lu.Nz, where x is the number of transactions, y the average transaction size, w the average size of maximal potentially large itemsets, u the number of maximal potentially large itemsets, and z the number of items.

Datasets from Blue-Martini Software Inc. and (usually) the artificial ones are regarded as sparse, while the other ones as dense.

3. The main strategy

The main strategy adopted by PatriciaMine is described by the pseudocode in Figure 3 and is based on a depth-first exploration of the FIST, similar to the one employed by the algorithms in [7, 12, 14, 9]. However, it must be remarked that while previous algorithms were expressed in a recursive fashion, PatriciaMine follows an iterative exploration strategy, which avoids the burden of managing recursion.

A first scan of the dataset $\mathcal{D}$ is performed to determine the set $\mathcal{I}'$ of frequent items, and a pruned instance $\mathcal{D}'$ of the original dataset where non-frequent items and empty transactions are removed (Step 1). Then, an *Item List* (IL) vector is created (Step 2), where each entry IL[ℓ] consists of three fields: IL[ℓ].item, IL[ℓ].count, and IL[ℓ].ptr, which store, respectively, a distinct item of $\mathcal{I}'$, its support and a pointer. The entries are sorted by decreasing value of the support field, hence the most frequent items are positioned to the top of the IL. The IL is linked to $\mathcal{D}'$ as follows. For each entry IL[ℓ], the pointer IL[ℓ].ptr points to a list that threads together all occurrences of IL[ℓ].item in $\mathcal{D}'$. We call such a list the *threaded list* for IL[ℓ].item with respect to $\mathcal{D}'$, and denote it by t-list($\ell, \mathcal{D}'$). The initial IL for the sample

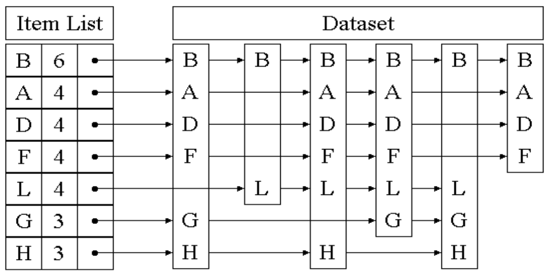

Figure 4. Initial IL and t-lists for the sample dataset

dataset and the t-lists built on a natural representation of the dataset, are shown in Figure 4. (The actual data structure used to represent $\mathcal{D}'$ will be discussed in the next section.)

Then, a depth-first exploration of the FIST is started visiting the children of each node by decreasing support order (i.e., left-to-right with respect to Figure 2). This exploration is performed by the while-loop in the pseudocode. A vector X and an integer h are used to store, respectively, the itemset associated with the last visited node of the FIST and its length (initially, X is empty and $h = 0$, meaning that the root has just been visited).

Let us consider the beginning of a generic iteration of the while-loop and let v be the last visited node of the FIST, associated with itemset $X = (a_1, a_2, \ldots, a_h)$, where a_h is the item labelling v, and, for $j < h$, a_j is the item labelling the ancestor w_j of v at distance $h-j$ from it. For $1 \leq j \leq h$, let ℓ_j be the IL index such that IL[ℓ_j].item $= a_j$, and note that $\ell_h < \ell_{h-1} < \cdots < \ell_1$; also denote by X_j the prefix $(a_1, a_2, \ldots, a_j)$ of X, which is the itemset associated with w_j (clearly, $X = X_h$).

The following invariant holds at the beginning of the iteration. Let ℓ' be an arbitrary index of the IL, and suppose that $\ell_{j+1} < \ell' \leq \ell_j$, for some $0 \leq j \leq h$, setting for convenience $\ell_0 = |\text{IL}| - 1$ and $\ell_{h+1} = -1$. Then, IL[ℓ'].count stores the support of item IL[ℓ'].item in $\mathcal{D}'_{X_j}$, and IL[ℓ'].ptr points to t-list($\ell', \mathcal{D}'_{X_j}$), that threads together all occurrences of IL[ℓ'].item in $\mathcal{D}'_{X_j}$ (we let $X_0 = \emptyset$ and $\mathcal{D}'_{X_0} = \mathcal{D}'$).

During the current iteration and, possibly, a number of subsequent iterations, the node u which is either the first child of v, if any, or the first unvisited child of one of v's ancestors is identified (Steps 3÷5). If no such node is found the algorithm terminates. It is easily seen that the item labelling u is the first item IL[ℓ].item found scanning the IL from the top, such that IL[ℓ].count $\geq$ min_sup and $\ell \neq \ell_j$ for every $1 \leq j \leq h$. If node u is found, its corresponding itemset is generated (Steps 6÷8). (Note that if u is the

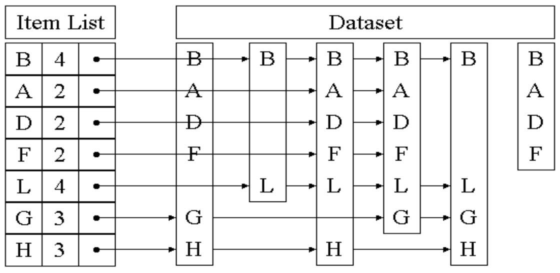

Figure 5. IL and t-lists after visiting (L,4)

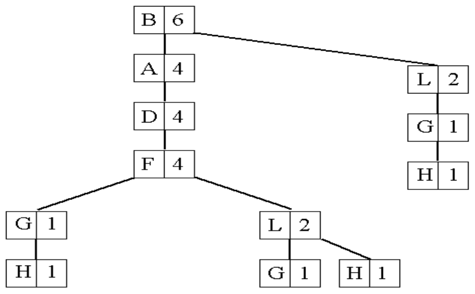

Figure 6. Standard trie for the sample dataset

child of an ancestor w of v, we have that before Step 6 is executed $X[0 \dots h-1]$ correctly stores the itemset associated with w.) Then, the first ℓ entries of the IL are updated so to enforce the invariant for the next iteration (for-loop of Step 9). Figure 5 shows the IL and t-lists for the sample dataset at the end of the while-loop iteration where node u=(L,4) is visited and itemset $X = (L)$ is generated. Observe that while the entries for items G and H (respectively, IL$[5]$ and IL$[6]$) are relative to the entire dataset, all other entries are relative to $\mathcal{D}'_X$.

The correctness of the whole strategy is easily established by noting that the invariant stated before holds with $h = 0$ at the beginning of the while-loop, i.e., at the end of the visit of the root of the FIST.

4. Representing the dataset as a Patricia trie

Crucial to the efficiency of the main strategy presented in the previous section is the choice of the data structure employed to represent the dataset $\mathcal{D}'$. Some previous works represented the dataset $\mathcal{D}'$ through a standard trie, called *FP-tree*, built on the set of transactions, with items sorted by *decreasing* support [7, 14]. The advantage of using the trie is substantial for dense datasets because of the compression achieved by merging common prefixes, but in the worst case, when the dataset is highly sparse, the number of nodes may be close to the size N of the original dataset (i.e., the sum of all transaction lengths). Since each node of the trie stores an item, a count value, which indicates the number of transactions sharing the prefix found along the path from the node to the root, plus other information needed for navigating the trie (e.g., pointers to the children and/or to the father), the overall space taken by the trie may turn out to be αN, where α is a constant greater than 1.

For these reasons, it has been suggested in [12, 9] that sparse datasets, for which the trie becomes space inefficient, be stored in a straightforward fashion as arrays of transactions. However, these works also encourage to switch to the trie representation during the course of execution, for portions of the dataset which are estimated to be sufficiently dense. However, an effective heuristic to decide when to switch from one structure to another is hard to find and may be costly to implement. Moreover, even if a good heuristic was found, the overhead incurred in the data movement may reduce the advantages brought by the compression gained.

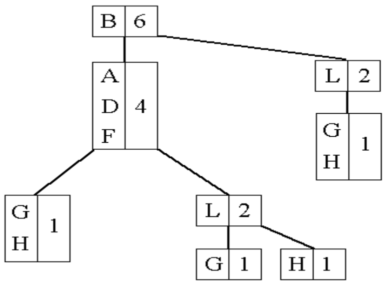

Figure 7. Patricia trie for the sample dataset

To avoid the need for two alternative data structures to attain space efficiency, our algorithm resorts to a compressed trie, better known as *Patricia trie* [8]. The Patricia trie for a dataset $\mathcal{D}'$ is a modification of the standard trie: namely, each maximal chain of nodes $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k$, where all v_i's have the same count value c and (except for v_k) exactly one child, is coalesced into a single node that inherits count value c, v_k's children, and stores the sequence of items previously stored in the v_i's. (A Patricia trie representation of a transaction dataset has been recently adopted by [6] in an dynamic setting where the dataset evolves with time, and on-line queries on frequencies of individual itemsets are supported.)

The standard and Patricia tries for the sample dataset are compared in Figure 6 and 7, respectively. As the figure shows, a Patricia trie may still retain some single-child nodes, however these nodes identify boundaries of transactions that are prefixes of other transactions. The following theorem provides an upper bound on the overall size of the

Patricia trie.

Theorem 1 *A dataset $\mathcal{D}'$ consisting of M transactions with aggregate size N can be represented through a Patricia trie of size at most $N + O(M)$.*

Proof. Consider the Patricia trie described before. The trie has less than $2M$ nodes since each node which has either zero or one child accounts for (one or more) distinct transactions, and, by standard properties of trees, all other nodes are at most one less than the number of leaves. The theorem follows by noting that the total number of items stored at the nodes is at most N. $\square$

It is important to remark that even for sparse datasets, which exhibit a moderate sharing of prefixes among transactions, the total number of items stored in the trie may turn out much less than N, and if the number of transactions is $M \ll N$, as is often the case, the Patricia trie becomes very space efficient. To provide empirical evidence of this fact, Table 1 compares the space requirements of the representations based on arrays, standard trie, and Patricia trie, for the datasets introduced before, on some fixed support thresholds. For each dataset the table reports: the number of transactions, the average transaction size (AvTS), the chosen support threshold (in percentage), and the sizes in bytes of the various representations (data are relative to datasets pruned of non-frequent items). An item is assumed to fit in one word (4 bytes). For the array-based representation we considered an overhead of 1 word for each transaction, while for the standard and Patricia tries, we considered an overhead per node of 4 and 5 words, respectively, which are needed to store the count, the pointer to the father and other information used by our algorithm (the extra word in each Patricia trie node is used to store the number of items at the node).

The data reported in the table show the substantial compression achieved by the Patricia trie with respect to the standard trie, especially in the case of sparse datasets. Also, the space required by the Patricia trie is comparable to, and often much less than that of the simple array-based representation. In the few cases where the former is larger, indicated in bold in the table, the difference between the two is rather small (and can be further reduced through a more compact representation of the Patricia trie nodes). Furthermore, it must be observed that in the execution of the algorithm additional space is required to store the threaded lists connected to the IL. Initially, this space is proportional to the overall number of items appearing in the dataset representation, which is smaller for the Patricia trie due to the sharing of prefixes among transactions.

Construction of the Patricia trie Although the Patricia trie provides a space efficient data structure for representing $\mathcal{D}'$, its actual construction may be rather costly, thus influencing the overall performance of the algorithm especially if, as it will be discussed later, the dataset is projected a number of times during the course of the algorithm.

A natural construction strategy starts from an initial empty trie and inserts one transaction at a time into it. To insert a transaction t, the current trie is traversed downwards along the path that corresponds to the prefix shared by t with previously inserted transactions, suitably updating the count at each node, until either t is entirely covered, or a point in t is reached where the shared prefix ends. In the latter case, the remaining suffix is stored into a new node added as a child of the last node visited. In order to efficiently search the correct child of a node v during the downward traversal of the trie, we employ a hash table whose buckets store pointers to the children of v based on the first items they contain. (A similar idea was employed by the Apriori algorithm [3] in the hash tree.) The number of buckets in the hash table is chosen as a function of the number of children of the node, in order to strike a good trade-off between the space taken by the table and the search time. Moreover, since during the mining of the itemsets the trie is only traversed upwards, the space occupied by the hash table can be freed after the trie is build.

5. Optimizations

A number of optimizations have been introduced and tested in the implementation of the main strategy described in Section 3. In the following subsections, we will always make reference to a generic iteration of the while-loop of Figure 3 where a new frequent itemset X is generated in Step 8 after adding, in Step 6, item IL$[\ell]$.item. Also, we define as *locally frequent items* those items IL$[j]$.item, with $j < \ell$, such that their support in $\mathcal{D}'_X$ is at least min_sup.

5.1. Projection of the dataset

After frequent itemset X has been generated, the discovery of all frequent supersets $Y \supset X$ could proceed either on a physical projection of the dataset (i.e., a materialization of $\mathcal{D}'_X$) and on a new IL, both restricted to the locally frequent items, or on the original dataset $\mathcal{D}'$, with $\mathcal{D}'_X$ is identified by means of the updated t-lists in the IL (in this case, a new IL or the original one can be used).

The first approach, which was followed in FP-Growth [7], is effective if the new IL and $\mathcal{D}'_X$ shrink considerably. On the other hand, in the second approach, employed in Top-Down FP-Growth [14], no time and space overheads are incurred for building the projected datasets and maintaining in memory all of the projected datasets along a path of the FIST.

Dataset	Transactions	AvgTS	min_sup %	Array	Trie	Patricia
Chess	3,196	35.53	20	467,060	678,560	250,992
Connect-4	67,557	31.79	60	8,861,312	69,060	55,212
Mushroom	8,124	22.90	1	776,864	532,720	380,004
Pumsb	49,046	33.48	60	6,765,568	711,800	349,180
Pumsb*	49,046	37.26	20	7,506,220	5,399,120	2,177,044
T10.I4.D100k.N1k.L2k	100,000	10.10	0.002	4,440,908	14,294,760	**5,129,212**
T40.I10.D100k.N1k.L2k	100,000	39.54	0.25	16,217,064	71,134,380	**16,935,176**
T30.I16.D400k.N1k.L2k	397,487	29.30	0.5	48,175,824	163,079,980	41,023,616
POS	515,597	6.51	0.01	15,497,908	32,395,740	13,993,508
WebView1	59,601	2.48	0.054	831,156	1,110,960	618,292
WebView2	77,512	4.62	0.004	1,742,516	4,547,380	**1,998,316**

Table 1. Space requirements of array-based, standard trie, and Patricia trie representations

Ideally, one should implement a hybrid strategy allowing for physical projections only when they are beneficial. This was attempted in OpportuneProject [9] where physical projections are always performed when the dataset is represented as an array of transactions (and if sufficient memory is available), while they are inhibited when the dataset is represented through a trie, unless sufficient compression can be attained. However, in this latter case, no precise heuristic is provided to decide when physical projection must take place. In fact, the compression rate is rather hard to estimate without doing the actual projection, hence incurring high costs.

In our implementation, we experimented several heuristics for limiting the number of projections. Although no heuristic was found superior to all others in every experiment, a rather simple heuristic exhibited very good performance in most cases: namely, to allow for physical projection only at the top s levels of the FIST and when the locally frequent items are at least k (in the experiments, $s = 3$ and $k = 10$ seemed to work fairly well). The rationale behind this heuristic is that the cost of projection is justified if the mining of the projected dataset goes on for long enough to take full advantage of the compression it achieves. Moreover, the heuristic limits the memory blowup by requiring at most s projected datasets to coexist in memory. Experimental results regarding the effectiveness of the heuristic, will be presented and discussed in Section 6.1

5.2. Immediate generation of subtrees of the FIST

Suppose that at the end of the for-loop every locally frequent item IL$[j]$.item, with $j < \ell$, has support IL$[j]$.count $=$ IL$[\ell]$.count $= c$ in $\mathcal{D}'_X$. Let Z denote the set of the locally frequent items. Then, for every $Z' \subseteq Z$ we have that $X \cup Z'$ is frequent with support c. Therefore, we can immediately generate all of these itemsets and set $\ell = \ell + 1$ rather than resetting $\ell = 0$ after the for-loop.[2] Viewed on the FIST, this is equivalent to generate all nodes in the subtree rooted at the node associated with X, without actually exploring such a subtree.

A similar optimization was incorporated in previous implementations, but limited to the case when the t-list$(\ell, \mathcal{D}'_X)$, pointed by IL$[\ell]$.ptr, consists of a single node. Our condition is more general and encompasses also cases when t-list$(\ell, \mathcal{D}'_X)$ has more than one node.

5.3. Implementation of the for loop

Another important issue concerns the implementation of the for-loop (Step 9), which contributes a large fraction of the overall running time. By the invariant previously stated, we have that, before entering the for-loop, IL$[\ell]$.ptr points to head of t-list$(\ell, \mathcal{D}'_X)$, that is, it threads together all of the occurrences of IL$[\ell]$.item in nodes of the trie corresponding to transactions in $\mathcal{D}'_X$. Moreover, the algorithm must ensure that the count of each such node is relative to $\mathcal{D}'_X$ and not to the entire dataset. Let T_X denote the portion of the trie whose leaves are threaded together by t-list$(\ell, \mathcal{D}'_X)$.

The for-loop determines t-list$(j, \mathcal{D}'_X)$ for every $0 \leq j < \ell - 1$, and updates IL$[j]$.count to reflect the actual support of IL$[j]$.item in $\mathcal{D}'_X$. To do so, one could simply take each occurrence of IL$[\ell]$.item threaded by t-list$(\ell, \mathcal{D}'_X)$ and walk up the trie suitably updating the count of each node encountered, and the count and t-list of each item stored at the node. This is essentially, the strategy implemented by Top Down FP-growth [14] and OpportuneProject (under trie representation) [9]. However, it has the drawback of traversing every node $v \in T_X$ multiple times, once for each leaf in v's subtree. It is not difficult to show an example

[2]This optimization is inspired by the concept of *closed frequent itemset* [11] in the sense that only $X \cup Z$ is closed and would be generated when mining this type of itemsets.

where, with this approach, the number of node traversals is quadratic in the size of T_X.

In our implementation, we adopted an alternative strategy that, rather than traversing each individual leaf-root path in T_X, performs a global traversal from the leaves to the root guided by the entries of the IL which are being updated. In this fashion, each node in T_X is traversed only once. We refer to this strategy as the *item-guided traversal*. Specifically, the item-guided traversal starts by walking through the nodes threaded together in t-list$(\ell, \mathcal{D}'_X)$. For each such node v, the count and t-list of each item IL$[j]$.item stored in v, with $j < \ell$, are updated, and v is inserted in t-list$(j, \mathcal{D}'_X)$ marked as *visited*. Also, the count and t-list of the last item, say IL$[j']$.item, stored in v's father u are updated and u is inserted in t-list$(j', \mathcal{D}'_X)$ marked as *unvisited*. After all nodes in t-list$(\ell, \mathcal{D}'_X)$ have been dealt with, the largest index $j < \ell$ is found such that t-list$(j, \mathcal{D}'_X)$ contains some unvisited nodes (which can be conveniently positioned at the front of the list). Then, the item-guided traversal is iterated walking through the unvisited nodes in t-list$(j, \mathcal{D}'_X)$. It terminates when no threaded list is found that contains unvisited nodes (i.e., the top of the IL is reached). The following theorem is easily proved.

Theorem 2 *The item-guided traversal correctly visits all nodes in T_X. Moreover, each such node with k direct children is touched k times and fully traversed exactly once.*

6. Experimental results

This section presents the results of several experiments we performed on the datasets described in Section 2.1. Specifically, in Subsection 6.1 we assess the effectiveness of our implementation, while in Subsections 6.2 and 6.3 we compare the performance of PatriciaMine with that of other prominent algorithms. The experiments reported in the first two subsections have been conducted on an IBM RS/6000 SP multiprocessor, using a single 375Mhz POWER3-II processor, with 4GB main memory, and two 9.1 GB SCSI disks under the AIX 4.3.3 operating system. On this platform, running times as well as other relevant quantities (e.g., cache and TLB hits/misses) have been measured with hardware counters, accessed through the HPM performance monitor by [5]. Instead, since for OpportuneProject only the object code for a Windows platform was made available to us by the authors, the experiments in Subsection 6.3 have been performed on a 1.7Ghz Pentium IV PC, with 256MB RAM, and 100GB hard disk, under Windows 2000 Pro.

6.1. Effectiveness of the heuristic for conditional projection

A first set of experiments was run to verify whether allowing for physical projections of the dataset improves performance and if the heuristic we implemented to decide when to physically project the dataset is effective. The results of the experiments are reported in Figures 8 and 9 (running times do not include the output of the frequent itemsets). For each dataset, we compared the performance of PatriciaMine using the heuristic (line “WithProjection”) with the performance of a version of PatriciaMine where physical projection is inhibited (line “WithoutProjection”), on four different values of support, indicated in percentage. It is seen that the heuristic yields performance improvements, often very substantial, at low support values (e.g., see Connect-4, Pumsb*, WebView1/2, T30.I16.D400k.N1k.L2k, and T40.I10.D100k.N1k.L2k) while it has often no effect or incurs a slight slowdown at higher supports. This can be explained by the fact that at high supports the FIST is shallow and the projection overhead cannot be easily hidden by the subsequent computation. Note that the case of Pos is anomalous. For this dataset the heuristic, and in fact all of the heuristics we tested, slowed down the execution, hence suggesting that physical projection is never beneficial. This case, however, will be further investigated.

We also tested the speed-up achieved by immediately generating all supersets of a certain frequent itemset X when the locally frequent items have the same support as X. In particular, we observed that the novelty introduced in our implementation, that is considering also those cases when the threading list t-list$(\ell, \mathcal{D}'_X)$ consists of more than one node, yielded a noticeable performance improvement (e.g., a factor 1.4 speed-up was achieved on WebView1 with support 0.054%, and a factor 1.6 speed-up was achieved on WebView2 with support 0.004%).

We finally compared the effectiveness of the implementation of the for-loop of Figure 3 based on the novel item-guided traversal, with respect to the straightforward one. Although the item-guided traversal is provably superior in an asymptotic worst-case sense (e.g., see Theorem 2 and the discussion in Section 5.3) , the experiments provided mixed results. For all dense datasets and for Pos, the item-guided traversal turned out faster than the straightforward one up to a factor 1.5 (e.g., for Mushroom with support 5%), while for sparse datasets it resulted actually slower by a factor at most 1.2. This can be partly explained by noting that if the tree to be traversed is skinny (as is probably the case for the sparse datasets, except for Pos) the item-guided traversal cannot provide a substantial improvement while it suffers a slight overhead for the scan of the IL. Moreover, for some sparse datasets, we observed that while the item-guided traversal performs a smaller number of instructions, it exhibits less locality (e.g., it incurs higher TLB misses) which causes the higher running time. We conjecture that a refined implementation could make the item-guided traversal competitive even for sparse datasets.

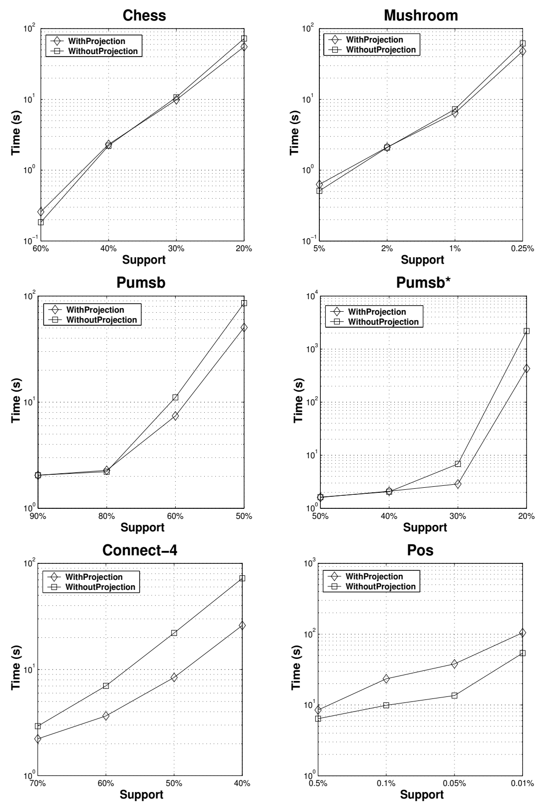

Figure 8. Comparison between PatriciaMine with and without projection on Chess, Mushroom, Pumsb, Pumsb*, Connect-4, Pos

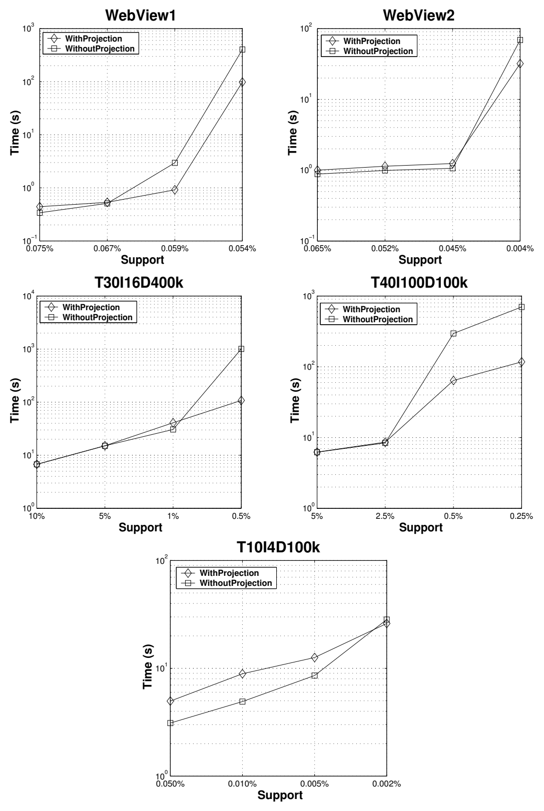

Figure 9. Comparison between PatriciaMine with and without projection on WebView1, WebView2, and some artificial datasets

6.2. Comparison with other algorithms

In this subsection, we compare PatriciaMine with other prominent algorithms whose source code was made available to us: namely FP-Growth [7], which has been mentioned before, DCI [10], and Eclat [15].

DCI (Direct Count & Intersect) performs a breadth-first exploration of the FIST, generating a set of candidate itemsets for each level, computing their support, and then determining the frequent ones. It employs two alternative representations for the dataset, a horizontal and a vertical one, and, respectively, a count-based and intersection-based method to compute the supports, switching adaptively from one to the other based on the characteristics of the dataset.

Eclat, instead is based on a depth-first exploration strategy (like FP-Growth and PatriciaMine). It employs a vertical representation of the dataset which stores with each item the list of transaction IDs (TID-list) where it occurs, and determines an itemset's support through TID-lists intersections. The counting mechanism was successively improved in dEclat [16] by using diffsets, that is, differences between TID-lists, in order to avoid managing very long TID-lists.

For FP-Growth and Eclat, we used the source code developed by Goethals[3], while for DCI we obtained the source code directly from the authors. The implementation of Eclat we employed includes the use of diffsets.

The experimental results are reported in Figures 10 and 11. For each dataset, a graph shows the running times achieved by the algorithms on four support values, indicated in percentages. (Here we included the output time since for DCI the writing on file of frequent itemsets is functional to the algorithm's operation.) It is easily seen that the performance of PatriciaMine is significantly superior to that of Eclat and FP-Growth on all datasets and supports. We also observed that Eclat features higher locality than FP-Growth, exhibiting in some cases a better running time, though performing a larger number of instructions.

Compared to DCI, PatriciaMine is consistently and often substantially faster at low values of support, while at higher supports, where execution time is in the order of a few sec-

[3]Available at `http://www.cs.helsinki.fi/u/goethals`

onds, the two algorithms exhibit similar performance and sometimes PatriciaMine is slightly slower, probably due to the trie construction overhead. However, it must be remarked that small differences between DCI and Patricia at low execution times could also be due to the different format required of the initial dataset, and different input/output functions employed by the two algorithms.

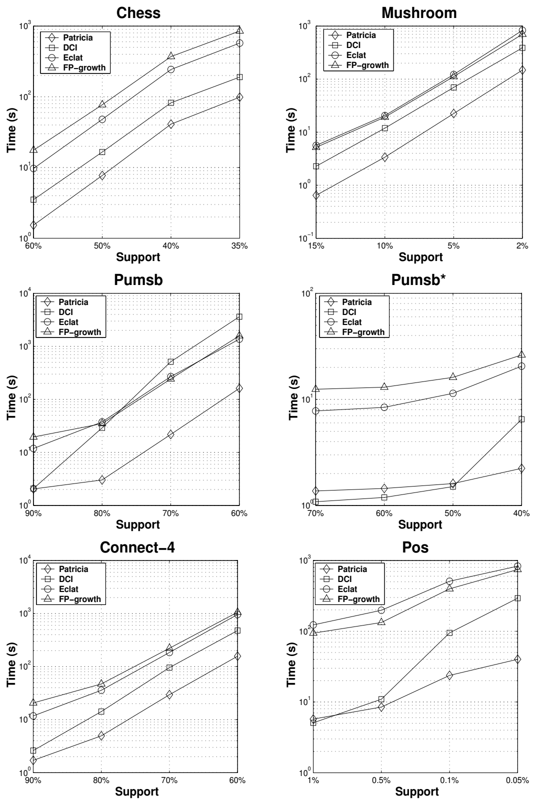

Figure 10. Comparison of PatriciaMine, DCI, Eclat and FP-Growth on Chess, Mushroom, Pumsb, Pumsb*, Connect-4, Pos

6.3. Comparison with OpportuneProject

Particularly relevant for our work is the comparison between PatriciaMine and OpportuneProject [9], which, to the best of our knowledge, represents the latest and most advanced algorithm in the family stemmed from FP-Growth. For lack of space, we postpone a detailed and critical discussion of the strengths and weaknesses of the two algorithms to the full version of the paper.

Figures 12 and 13, report the performances exhibited by PatriciaMine and OpportuneProject on the Pentium/Windows platform for a number of datasets and supports. It can be seen that, the performance of PatriciaMine is consistently superior, up to one order of magnitude (e.g., in Pumsb*). The only exception are Pos (see graph labelled “Pos with projection”) and the artificial dataset T30.I16.D400k.N1k.L2k. For Pos, we have already observed that our heuristic for limiting the number of physical projections does not improve the running time. In fact, it is interesting to note that by inhibiting projections, PatriciaMine becomes faster than OpportuneProject (see graph labelled “Pos without projection”). This suggests that a better heuristic could eliminate this anomalous case.

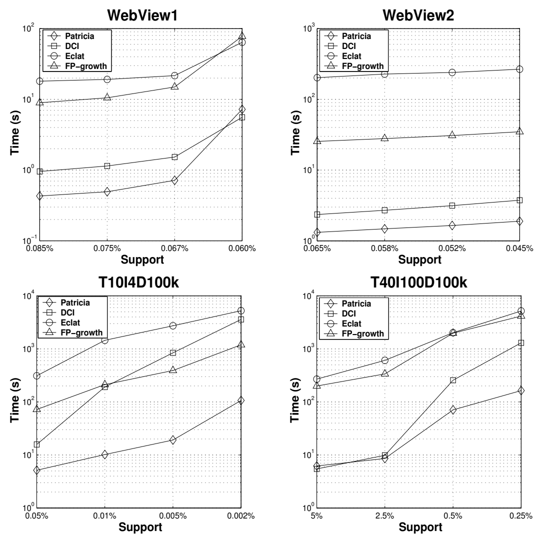

Figure 11. Comparison of PatriciaMine, DCI, Eclat and FP-Growth on WebView1, WebView2, and some artificial datasets

As for T30.I16.D400k.N1k.L2k, some measurements we performed revealed that the time taken by the initialization of the Patricia trie accounts for a significant fraction of the running time at high support thresholds, and such an initial overhead cannot be hidden by the subsequent mining activity. However, at lower support thresholds, where the computation of the frequent itemsets dominates over the trie construction, PatriciaMine becomes faster than OpportuneProject.

Finally we report that on WebView1 for absolute support 32 (about 0.054%), OpportuneProject ran out of memory while PatriciaMine successfully completed the execution.

References

[1] R. Agrawal, C. Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc.*

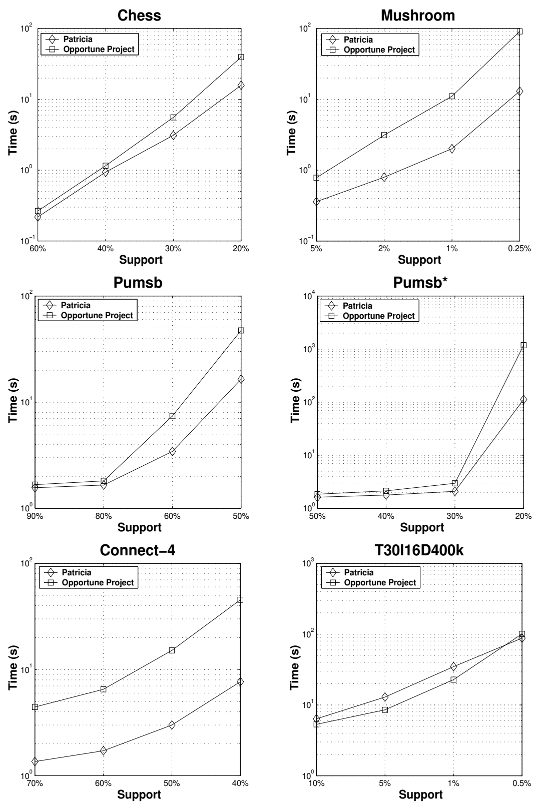

Figure 12. Comparison of PatriciaMine and OpportuneProject on Chess, Mushroom, Pumsb, Pumsb*, Connect-4, and T30I16D400k

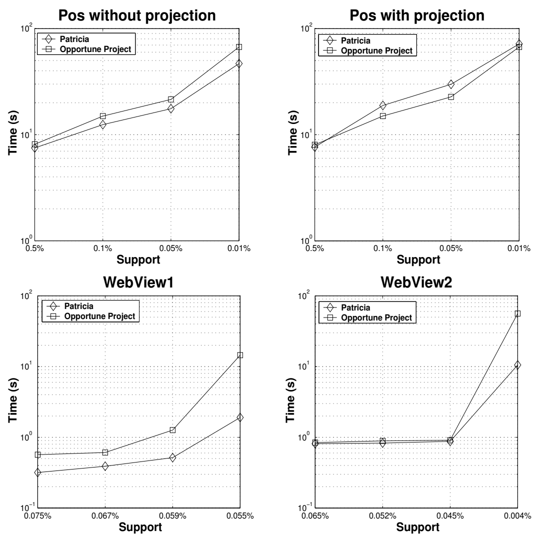

Figure 13. Comparison of PatriciaMine and OpportuneProject on Pos, WebView1, WebView2

of the ACM SIGMOD Intl. Conference on Management of Data, pages 207–216, 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Very Large Data Base Conference*, pages 487–499, 1994.

[4] R. Bayardo. Efficiently mining long patterns from databases. In *Proc. ot the ACM SIGMOD Intl. Conference on Management of Data*, pages 85–93, 1998.

[5] L. DeRose. Hardware Performance Monitor (HPM) toolkit. version 2.3.1. Technical report, Advanced Computer Technology Center, Nov. 2001.

[6] A. Hafez, J. Deogun, and V. Raghavan. The item-set tree: A data structure for data mining. In *Proc. of the 1st Int. Conference on Data Warehousing and Knowledge Discovery*, LNCS 1676, pages 183–192, 1999.

[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12, 2000.

[8] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison Wesley, Reading, MA, 1973.

[9] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *Proc. of the 8th ACM SIGKDD Intl. Conference on Knowledge Discovery and Data Mining*, pages 229–238, July 2002.

[10] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive resource-aware mining of frequent sets. In *Proc. of the IEEE Intl. Conference on Data Mining*, pages 338–345, Dec. 2002.

[11] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of the 7th Int. Conference on Database Theory*, pages 398–416, Jan. 1999.

[12] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proc. of IEEE Intl. Conference on Data Mining*, pages 441–448, 2001.

[13] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21st Very Large Data Base Conference*, pages 432–444, Sept. 1995.

[14] K. Wang, L. Tang, J. Han, and J. Liu. Top down FP-Growth for association rule mining. In *Proc. of the 6th Pacific-Asia Conf. on Advances in Knowledge Discovery and Data Mining*, LNCS 2336, pages 334–340, May 2002.

[15] M. Zaki. Scalable algorithms for association mining. *IEEE Trans. on Knowledge and Data Engineering*, 12(3):372–390, May-June 2000.

[16] M. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. of the 9th ACM SIGKDD Intl. Conference on Knowledge Discovery and Data Mining*, Aug. 2003.