Mining Frequent Itemsets Using Re-Usable Data Structure

Mohamed Yakout           Alaaeldin M. Hafez           Hussein H. Aly
Department of Computer Science and Systems Engineering, Alexandria University, Egypt.

Abstract - Several algorithms have been introduced for mining frequent itemsets. The recent dataset-transformation approach suffers either from the possible increasing in the number of structures that could be produced through the execution of the algorithm or from the problem of the processing time in either projecting or decomposing the datasets. Moreover, the constructed structure cannot be re-used in ad-hoc mining queries or in other mining processes. In this paper, the ItemSet Tree (IST) structure is used in effectively counting the itemsets' support to overcome the above limitations. To speed up the support counting process, a proposal for using a Guidance Information Bits and tree size reduction is presented. The TDF algorithm will be proposed to find all the frequent itemsets. TDF explores the frequent itemsets search space in depth-first to generate candidates from the search space and count their support in the IST. Several experiments have been conducted to study the performance of the TDF algorithm.

Keywords: Data Mining, Frequent itemsets, itemset tree.

1 Introduction

Mining frequent itemsets is a fundamental problem in the Data Mining. It was originally proposed by Agrawal [1] in his association rule model and the support confidence framework. This problem is further exasperated when dealing with datasets, which contain highly frequent, yet often meaningful patterns. While many different algorithms have been proposed, the fact remains that finding frequent itemsets enables essential data mining tasks such as discovering association rules, correlations between data, causality, sequential patterns episodes multi-dimensional patterns, and other important database tasks.

Two main classes of algorithms have been proposed. The first class uses a process of candidate generate-and-test to find frequent itemsets. The second class of algorithms transform the original data into a representation better suited for frequent itemset mining.

The data-transformation class is the recent trend taken in the researches. Some of these algorithms extend the data representation with extra structures in the main memory that help in identifying the frequent itemsets faster [2], [7]. The others algorithms try to shrink the transaction initial representation to minimize the counting time, by destroying and building other representation in each iteration [1], [2], [4], [8], [7].

The great practical benefits of mining association rules and its wide area of applications have led to several proposals for fast mining of association rules. The recent proposals are targeted to transforming the dataset into another representation. Those proposals, although contributed towards making the process more applicable in practical systems, still suffer from problems appear in different aspects. For the algorithms, which uses an initial representation then extend it, suffer from the possible increasing in the number of extensions that could be produced through the execution of the algorithm. For the algorithms, which starts with an initial representation then try to shrink it, suffer from the problem of the processing time in either projecting or decomposing the datasets.

A common problem exists in the recent algorithms is that the constructed data structure can not be reused in ad-hoc mining queries or another mining process. The reason for that can either be:

1. The constructed structure is built after applying some filters on the transaction file, and these filters depends on collecting information from the whole database. So the whole database should be checked at least once before constructing the data structure.
2. The constructed structure changes in each iteration through the execution of the algorithm.

In this paper, we are presenting an algorithm for finding frequent itemsets. The proposed technique tries to overcome the noted drawbacks in the data-transformation approach algorithms. We tried to find a data structure representation for the dataset that can be used in both finding all the frequent itemsets efficiently and also can be used in ad-hoc mining queries. We had two options to do that; first, use one of the proposed structures for finding the frequent itemsets in ad-hoc mining queries. Second, use an already existing data structure for ad-hoc mining queries in finding the frequent itemsets. The first option was difficult to be realized.
Recently, the ItemSet Tree (IST) for transaction file representation has been introduced in [5] and [6] for efficient handling of data updates and ad-hoc mining queries. In the proposed algorithm, we make use of the IST in the support counting process to find all the frequent itemsets. We are presenting a proposal for using a Guidance Information Bits (GIB) to speed up the process of the support counting in the Itemset Tree. To find all the frequent itemsets, the proposed algorithm explores the frequent itemsets search space in depth-first. We generate candidates from the search space and count their support in the Itemset Tree. The generated candidates are grouped such that they share the leading itemset and differ only in the last item.

The rest of the paper is organized as follows; section 2 presents related work for finding frequent itemsets. In section 3, the IST is discussed with proposal to speedup the support counting process. Section 4 presents the TDF algorithm to find all the frequent itemsets. Finally, section 5 presents the experimental study.

2 Related Work

Despite the considerable amount of algorithms proposed in the last decade for solving the problem of finding frequent itemsets in transactional database, a single best approach still to be found. There are two different approaches for generating frequent itemsets. First, candidate set generate-and-test approach: most of previous algorithms belong to this group. The basic idea is to generate and then test the candidate set. This process is repeated in a bottom up fashion until no candidate set can be formed. Second, data transformation approach: it transforms a dataset to a data structure (e.g. tree, special lists, … etc.) for efficient mining for frequent itemsets. In the following, we will concentrate on the recent approach and explore some of these algorithms.

In [1] TreeProjection constructs a lexicographical tree and projects a large database into a set of reduced, item-based sub-database based on the frequent itemsets mined so far. The number of nodes in its lexicographic tree is exactly that of the frequent itemsets. The main cost in TreeProjection is computing of matrices and transaction projections. In a database with a large number of frequent items, the matrices can become quite large, and the computation cost could become high. Moreover in large databases, transaction projection may become costly.

FP-growth algorithm presented in [2] transforms the dataset into a compact structure, after filtering the dataset using the infrequent 1-itemsets, called FP-Tree and then recursively builds conditional FP-Trees to mine frequent patterns. It has performance gains since it avoids candidates’ generation. However, the number of conditional FP-Trees is of the same order of magnitude as the number of frequent itemsets.

The PD-Decomposition algorithm presented in [4] reduces the dataset during each pass by splitting transactions and combining similar transactions together, thus decreasing counting time. PD-Decomposition does not need to generate candidate sets, since all subsets of any transaction in the reduced set dataset are frequent thus should be counted. It shrinks dataset each time when infrequent itemsets are discovered. The main cost comes from the decomposition process of the transactions using the infrequent itemsets generated.


OpportunityProject algorithm discussed in [7] is a new projection based algorithm. A novel pseudo projection method for tree-based representations in the depth-first search has been presented, which greatly improves the efficiency of counting and projecting operations in dense transaction subsets. Also, an array-based data structure that is simple and space efficient for sparse datasets has been presented. It overcomes the limits in FP-growth, using a tree-based representation, similar to FP-growth, of projected transactions for dense datasets. Moreover, a heuristic was proposed to opportuneely switch between a depth-first and a breadth-first visit of the frequent set tree.

3 A Re-Usable Data Structure for Mining Frequent Itemsets

In this section, we will present the data structure representation for the transaction file and how to use it in the support counting process. Moreover, a suggestion to speedup the process by reducing the structure size and adding Guidance Information Bits is introduced.

3.1 The ItemSet Tree (IST)

The ItemSet Tree (IST) is a trie tree data structure representation for the transaction file. The trie trees structure are widely used in the text-matching field. The IST was presented in [5] and [6] for ad-hoc mining queries. Let $I=\{i_1, i_2, \ldots, i_n\}$ be an ordered set of items. For two transactions $s_1=\{a_1, a_2, \ldots, a_l\}$ and $s_2=\{b_1, b_2, \ldots, b_k\}$, let $s_1 \preceq s_2$ iff $a_s \preceq b_s$ for all $1 \leq s \leq \min(l,k)$. We call $l$ and $k$, the lengths of $s_1$ and $s_2$, respectively [5]. Node $s_i$ is ancestor node of node $s_j$ if $s_j \preceq s_i$ that is $s_i=\{a_1, a_2, \ldots, a_l\}$ and $s_j=\{a_1, a_2, \ldots, a_l, a_{l+1}\}$, for some $1 \leq l < k$. Moreover, a node $s_i$ is a direct
ancestor of node \( s_i \) if \( s_j \) is an ancestor of \( s_i \) and there is no other node \( s_k \) such that \( s_j \subseteq s_k \subseteq s_i \). Frequency of a node \( s \) is denoted by \( f(s) \) representing the count of transactions that have the same transaction group \( s \). Fig. 1 represents an IST construction example if the transaction file contains the \( F = \{ \{1,2,3,4\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,3,4\}, \{2,3,4\} \} \).

**Fig. 1.** Itemset Tree Construction Example.

Counting the support of an itemset \( s \) using the IST is done by adding up frequencies of those encountered itemsets that contain \( s \) [5] [6]. Fig. 2 introduces the algorithm \textit{Count} to count the support of an itemset \( s \) in the IST \( T \).

```
Algorithm Count(\( s,T,f(s) \))
\( s \) is an itemset
\( T \) is the itemset tree
Begin
1: If \( s \subseteq \text{itemset}(T) \) Then
2: \( f(s) = f(s) + f(T) \)
3: exit
4: EndIf
5: If IsInsideSubTree(\( T, s \)) Then
6: \( T' = \text{Child}(T) \)
7: While \( (T' \neq \text{NULL}) \)
8: call Count(\( s,T',f(s) \))
9: \( T' = \text{Sibling}(T') \)
10: Loop
11: EndIf
End
```

**Fig. 2.** Algorithm to count the support of an itemset using the itemset tree

Example: Using the IST in fig. 1, we need to count the support of \( \{2,3\} \):
- Starting from the smallest subtree \( \{1\} \). In this case, \( s \) satisfies \textit{IsInsideSubTree} condition.
- The subtree \( \{1\} \) is orderly traversed; starting with node \( \{1,2\} \). For this node \( s \) satisfy \textit{IsInsideSubTree} condition.
- The subtree \( \{1,2\} \) is orderly traversed; starting with node \( \{1,2,3,4\} \). \( s \subseteq \{1,2,3,4\} \), \( f=1 \).
- Go back to next subtree of \( \{1\}, \) node \( \{1,3\} \). \( s \subseteq \{1,3\} \) and item \( 2 \in \text{LastItem}(\{1,3\}) \) and does not exist in \( \{1,3\} \), so \( s \) can not appear is this subtree.

- Go back to next subtree of \( \{1\}, \) node \( \{2,3\} \). \( s \subseteq \{2,3\} \).

\( f=1+2=3 \) and no further traversing required in this subtree.

### 3.2 Reducing the Size of the Itemset Tree

In the process of finding all the frequent itemsets, we are not interested in the non-supported items in the file. Therefore, if we have an IST with only the supported items, the produced IST will have less number of nodes and less number of items within the tree nodes. Hence, the IST size is reduced and we can call it Support Counting Tree (SCT).

**Transaction File**

<table>
<thead>
<tr>
<th>Items support</th>
<th>Transaction with Supported Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,3,4,5}</td>
<td>{3,4,5}</td>
</tr>
<tr>
<td>{2,3,4,5,6}</td>
<td>{2,3,4,5}</td>
</tr>
<tr>
<td>{3,4,}</td>
<td>{3,4,}</td>
</tr>
<tr>
<td>{3,4,2,3}</td>
<td>{2,3}</td>
</tr>
<tr>
<td>{1,2,4,}</td>
<td>{2,3}</td>
</tr>
<tr>
<td>{1,2,5,}</td>
<td>{2,5}</td>
</tr>
<tr>
<td>{2,5,7,}</td>
<td>{2,5}</td>
</tr>
<tr>
<td>{3,4,2,3,4,5}</td>
<td>{3,4,2,3,4,5}</td>
</tr>
<tr>
<td>{5,6,7,8}</td>
<td>{5}</td>
</tr>
</tbody>
</table>

Non-Supported Items are highlighted MinSupport = 7

**Fig. 3.** Example for reducing the itemset tree size

Example: In fig. 3, a list of the transactions file in the left. After counting the support of each item and the use of a min-support of 7, the non-supported items are 1, 6, 7, and 8. The transactions will be as shown in the right listing. The ISTs before and after removing the non-supported items are also depicted in the figure.
To get the reduced IST, the IST can be built then cleaned up from the non-supported items. After building the IST and before the cleaning process, it can be dumped to a file to be used in other mining processes later.

### 3.3 Support Counting in the Itemset Tree Using Guided Search

One of the drawbacks in the proposed support counting algorithm using the IST is that the algorithm may make a lot of False Sub-Tree Checking.

**Definition: False Sub-Tree Checking:** If we say that we checked sub-tree r for the existence of itemset s was False Sub-Tree Checking, then:

1. a ∈ r for each item a ∈ s, where a < Last_Item(r).
2. s does not exist in r’s sub-tree effectively.

To reduce the number of False Sub-Tree checking, we suggest additional information, the Guidance Information Bits (GIB), to be added in some tree nodes to help guiding the search. The GIB is a bit map for whether an item exists in the current sub-tree or not. For a GIB of max-size d bits, the bits will be set for a tree node r = [a1, a2,…, ad] as follows.

- If a1 = i1, then a bit map represent the items i1, i1+1,…, i1+4. If j + d < n, number of items in the system, then the bit map will represent the items i1, i1+1,…, in.
- If the last bit information in the Parent(r) provides information for the item a1 = i1 and an > ai, then the bit map for node r will represent the items i1, i1+1,…, i1+n. If j + k > n then the bit will represent the items i1, i1+1,…, in.

Fig. 4. Visited nodes when counting the support of {7, 8} using the GIB.

![Fig. 4. Visited nodes when counting the support of {7, 8} using the GIB.](image)

**Example:** In fig. 4, the number of items n=8, d=4 bits. In order to count the support of itemset {7, 8}, we need to visit all the tree nodes. However, by making use of the GIB, only black nodes are visited.

### 3.4 Support Counting of a Set of Itemsets Using the Itemset Tree

In the process of mining the frequent itemsets, support counting of the itemsets one-by-one will slow down the process dramatically. Therefore, we are going to discuss an efficient method for counting the support of a set of itemsets. The set will be in the form C = {P | a1 ∪ a2,…, ai > last_item(P)}, since this will be the form of the generated candidates set to find the frequent itemset later in the next section.

**Lemma 1:** For an itemset tree T and two itemsets s1 = {P | a1} and s2 = {P | a2}, where P is an itemset, such that s1 ∈ P and s2 ∈ P, we must stop traversing a subtree r for s2 then r will not be traversed for s1.

Proof:

1. First part: Since s1 satisfies IsInsideSubTree check, so each item a, where a ∈ s1 and a < last_item(r), then a ∉ r. Since s1, s2 are the same except in the last item then for the same item a also a ∈ s2, so s2 will satisfy IsInsideSubTree.
2. Second Part: we stop traversing a subtree r to count the support of s2 if:
   a. s2 can’t exist in r, i.e. s2 doesn’t satisfy IsInsideSubTree function: In this case, at least an item a ∈ s2 and a < Last_Item(r), such that a ∉ r. Since s1, s2 share the same itemset P and also a ∈ s1, then s1 can’t exist in r’s subtree. If a = a2, the last item, then if ai ∈ r then s1 ⊂ r and if a1 ∉ r then s2 ⊂ r and in both cases we will stop traversing r’s subtree.
   b. s2 ⊂ r: then if ai ∈ r then s1 ⊂ r since P ⊂ r. Else if a1 ∉ r and because a2 ∈ r and a1 < a2 then s1 will not appear in r’s subtree. In both cases r will not be traversed for s1.

Based on Lemma 1, the algorithm CountSet is proposed in fig. 5 to count the support of a group of itemsets in the form {P | ai ∪ a2,…, ai > last_item(P)}. The group can be represented by two lists P and C, where C={a1, a2,…, ak} and a1 < a2 < … < ak. The algorithm will traverse the tree as if we are counting the support of the itemset P|ai. CountSet stops traversing a subtree r only if P | ai does not exist in P | ai ⊂ r.

**CountSet** starts by storing the NextValid in NV local variable, which is an index of an item in C’s list. NextValid means that all the sets P | ai ∪ a2,…, ai | ai ∈ C can be found in the subtree T. First, in line 2, a check for P existence in r is done followed by a check for each item in C (a1v1, a2v1, …, akv1). If ai is found in r then the support counters are updated. NV is updated with the index...
of the item found in \( r \), which is \( i \), to indicate that the items \( P \cup a_i, P \cup a_2, \ldots, P \cup a_{|\text{NextValid}|} \) are not valid for support counting by further traversing \( r \)’s subtree. Line 10 checks whether there are still itemsets valid for support counting in this subtree. Line 11 checks whether \( P \cup a_i \) can exist in the current subtree. If so, a recursive call for each child of \( r \) is done with the NextValid is NV.

Algorithm CountSet(C,P,T, NextValid )
P: Leading itemset of the group
C: Set of extensions
T: Itemset tree node
NextValid: index of the next valid itemset for counting in T subtree. (i.e. PU C[NextValid+1] can be found in subtree T)
Begin
1: NV = NextValid
2: If \( P \subset \text{items}(T) \) Then
3: For \( i = NV+1 \) to \( |C| \)
4: If \( c_i \in \text{items}(T) \) Then
5: NV = i
6: Count(c) = Count(c) + freq(r)
7: End If
8: Loop
9: End if
10: If (NV < |C| ) Then
11: If (IsInsideSubTree( T, P U c_i )) Then
12: \( T = \text{child}(T) \)
13: while \( (T' \subseteq P U c_i) \)
14: Call CountSet( C, P, T, NV)
15: \( T' = \text{Sibling}(T') \)
16: End loop
17: End If
18: End If
End

Fig. 5. Support counting algorithm of a set of itemsets that share a common leading itemset using the IST.

Example: Using the IST in fig. 1, we want to count the support of \{1,2\}, \{1,3\}, \{1,4\}. Using the Count algorithm, to count the support of \{1,2\}, we need to visit the IST nodes \{1\}, \{1,2\}. To count the support of \{1,3\} we need to visit \{1\}, \{1,2\}, \{1,2,3\}, \{1,3\}. To count the support of \{1,4\} we need to visit \{1\}, \{1,2\}, \{1,2,3\}, \{1,3\}, \{1,3,4\}. It is noted that by traversing the IST to count the support of \{1,4\} we will visit all the necessary nodes to count the support of \{1,2\} and \{1,3\}.

Using the CountSet algorithm where \( P = \{1\} \) and \( C = \{2,3,4\} \) and \( P \cup \{4\} = \{1,4\} \):

- Starting from subtree \{1\}; \( P = \{1\} \), no item of \( C \) exist is node \{1\}, \{1,4\} satisfies IsInsideSubTree condition.
- The subtree of \{1\} is orderly traversed; starting with node \{1,2\}. \( P \subset \{1,2\} \). \( c_2 \in \{1,2\} \). So \( NV = 1 \) and \( freq(c) = 2 \). \( NV < 3 \) and \{1,4\} satisfies IsInsideSubTree condition.
- The subtree of \{1,2\} is orderly traversed; starting with node \{1,2,3\}. \( P \subset \{1,2,3\} \). \( c_2 \in \{1,2,3\} \) and \( c_3 \in \{1,2,3\} \). In this case, \( NV = 3 \) and \( freq(c_3) = 1 \) and \( freq(c_1) = 1 \). Since \( VN = 3 \) \( = |C| \) then no further traversing in this subtree.

- Go back to next subtree of \{1\} where \( NV = 0 \), node \{1,3\}. \( P \subset \{1,3\} \) and \( c_3 \in \{1,3\} \). So \( NV = 2 \) and \( freq(c_3) = 1 \). \( NV < 3 \) and \{1,4\} satisfies IsInsideSubTree condition.
- The subtree of \{1,3\} is orderly traversed; starting with node \{1,3,4\}. \( P \subset \{1,3,4\} \). \( c_4 \in \{1,3,4\} \). So \( NV = 3 \) and \( freq(c_4) = 1 \). Since \( NV = 3 \) then no further traversing in this subtree.
- Go back to next subtree \{1\} where \( NV = 0 \), node \{2,3\}. \( P \subset \{2,3\} \), so no further traversing in this subtree.

4 Finding Frequent Itemsets Using Itemset Tree

In this section, we are presenting a proposal for making use of the itemset tree to find all the frequent itemsets given a minimum support. The frequent itemsets search space, which forms a lexicographic tree, will be explored in depth-first order. The lexicographic tree will help in generating the candidates itemsets in the form \( C = \{P \cup a_i \ | \ a_i \succ last\_item(P)\} \).

![Image](image_url)

Fig. 6. Frequent extensions and prospective extensions of the itemset \( P = \{2,3,5\} \).

The lexicographic tree, or the set enumeration tree, can be defined for a set of items \( I \). Assume there is a total ordering over the items \( I \). This ordering can be used to enumerate the subset lattice (search space). A node exists corresponds to a frequent itemset. A frequent 1-extension of node \( P \), \( E(P) = \{e_1, e_2, \ldots, e_l\} \), is the set of tree branches \( P \times e \), where \( P \times e \) is frequent for \( i = 1..k \). Let \( Q \) be the immediate ancestor of the itemset \( P \), which is a frequent itemset, in the lexicographic tree. The prospective extension of \( P \), \( F(P) = \{q_1, q_2, \ldots, q_l\} \subseteq E(Q) \) and \( q_i \succ last\_item(P) \). These are the possible frequent extensions of \( P \). Thus, \( E(P) \subseteq F(P) \subseteq E(Q) \). The prospective extensions set \( F(P) = \{q_1, q_2, \ldots, q_l\} \) for a frequent itemset \( P \) is used to produce the candidate itemsets \( C = \{P \cup q_i \ | \ i = 1..1\} \). From these candidates, we can get \( E(P) = \{q_i \ | \ P \cup q_i \ is \ frequent \ AND \ i = 1..1\} \).
The prospective extensions set $F(p) = \{q_1, \ldots, q_k\}$ for a frequent itemset $P$ is used to produce the candidate itemsets $C = \{P \cup q_i | 1 \leq i \leq k\}$. From these candidates we can get $E(P) = \{q_i | (P \cup q_i) \text{ is frequent AND } 1 \leq i \leq k\}$.

The TreeDepth-First (TDF) Algorithm: The TDF, in fig. 7, examines the lexicographic tree of the frequent itemsets in Depth-First order. The process of examination of a node refers to the counting of the support of the candidate extensions of the itemset. In other words, the support of all descendent itemsets of a node is determined before determining the frequent extensions of other nodes of the lexicographic tree.

![Algorithm TDF](image)

There are two parameters for the algorithm TDF: $P$, a frequent itemset and $F_p$, which is $F(P)$ the prospective extension of $P$.

The TDF algorithm starts by passing $P$ with its prospective extensions $F_p$, attached with counters, to the CountSet function, which counts the support of each itemsets $\{P \cup a_i | a_i \in F_p\}$ using the IST. Then, the set of frequent itemsets extensions of $P$ are discovered, and then $E(P) = \{a_i | P \cup a_i \text{ is frequent, } a_i \in F_p\}$. Once $E(P)$ is found, we recursively call TDF for each $e_k \in E(P)$ with its prospective extensions $F(e_k) = \{e_i | i = k + 1 \ldots |E(P)|\}$.

5 Experimental Study

This section presents the different experiments conducted to study the performance of the TDF algorithm, using the IST with its various shapes and reduced size; and compares it with the FP-growth algorithm.

The experiments have been conducted on a PC with 2.8 GHz speed with 512 MB main memory, running Microsoft Windows XP. The proposed and the competitive algorithms have been written in C++ and compiled using MS Visual C++ 6.0. Through our experiments, we are using synthetic data generator. A program developed by IBM Almaden research center was used to generate the dataset. The program is available from the IBM QUEST website (http://www.almaden.ibm.com/cs/quest). The program generates a synthetic transactional database using several parameters (see [9] for an explanation of every parameter and the rational behind the method of generating the database).

We decided to compare TDF performance with FP-growth as a proof of competency with recent mining algorithms. FP-growth algorithm is an efficient algorithm recently proposed. The novel idea is to build up frequent pattern trees to store data and mine frequent patterns using the trees. This results in: 1) FP-trees substantially smaller than the original data and saving costly database scans; 2) avoiding candidate set generation and testing.

The main costs in FP-tree-based mining involve recursively building conditional FP-trees. The number of conditional FP-trees can be enormous and the algorithm may run out of virtual memory. Further, the complicated data structure of FP-tree requires large number of pointers. In order to build the conditional FP-tree efficiently, each node needs three pointers. An FP-tree node contains an item, a counter and 3 pointers; the storage overhead of the pointers is 60% of the data storage.

The most interesting parameters that control the shape of the dataset are $(T)$, the avg. transaction size and $(I)$, avg. size of the maximal potential frequent itemsets. Through the performance study, we will change the values of these parameters. For the other parameters we will use the most popular values used in the previous publications, where $D=100K$, $L=2000$, and $N=1000$.

Since the IST is built based on the ordering that exists between the items, if we change this order, a different tree will be produced. If we re-ordered the items based on its support in descending, the produced trees, most probably, will have smaller number of levels and higher number of nodes in a tree level or Short-Wide tree. On the other hand, for the ascending ordered items, the trees, most probably, will have higher number of levels and smaller number of tree nodes in a tree level or Long-Narrow tree.

In the following, we will use the dataset generator to produce the experiments’ datasets. We will study the performance when both $(T)$ and $(I)$ change.

The processing time to find all the frequent itemsets for min-support values 1.5% will be reported for 5 simulations:

- Fpg: The FP-growth algorithm.
- SCT: TDF using the SCT.
- SCTShort: TDF using SCT Short-Wide.
- SCTLong: TDF using SCT Long-Narrow.
- IST: TDF using original IST.
We chose to plot the results on 2D-graphs, where we reported the total process time with the change in \((I)\) and there is a graph for each \((T)\) value \(T=5, 10, 15,\) and \(25\).

In the fig. 8, a comparison between Fp-growth and TDF when \(\text{MinSupport} = 1.5\%\) is presented. Each graph represent the reported processing times for Fp-growth and TDF when using each of the SCT, SCTShort, SCTLong, and IST. In each graph, a fixed value for \((T)\) was used and the reported values were plotted with the change in \((I)\).

In fig. 8.(a), \(T=5\), which is considered a dataset with relatively small transaction size. Fp-growth and TDF - when using the SCTShort and SCTLong - performed almost the same. The use of the IST and SCT results in a poor performance. When \(T=10\) in fig. 8.(b), the TDF using the SCTShort is the best. However, Fp-growth outperforms the TDF when using the SCT, SCTLong and the IST. The TDF using the IST produced the worst performance.

In fig. 8.(c), \(T=15\). When \(I=2\), Fp-growth’s performance is dramatically decreased. The use of the SCTShort produces the best performance - twice as good as Fp-growth. In fig. 8.(d), \(T=25\), which is considered a dataset with long transaction size. TDF generally outperforms the Fp-growth with a factor of about 5 to 6 times for very low values of \((I)\). Moreover, the TDF using the SCTShort performs the best and using the IST performs the worst. In both fig. 8.(c) and (d), it is noted that Fp-growth is highly affected by changing \((I)\), while the TDF performance is slightly affected.

Our performance study showed that TDF outperforms Fp-growth for higher values of \((T)\), average transaction size, and lower values for \((I)\), average size of the maximal potentially frequent itemsets:

- For higher values of \((T)\), the size of the produced trees is relatively high. The TDF builds only one tree and traverse it as necessary, while Fp-growth builds a large number of conditional FP-trees and traverse them to build other trees.

- For lower values of \((I)\), the dataset becomes scattered and so the number of supported items increases. In this case, the performance of both the Fp-growth and TDF is reduced; the Fp-growth builds more conditional trees and the TDF trees size increases. However, TDF outperforms the Fp-growth because of using a single tree without extending it and also because of the support counting algorithm, which tries to minimize the number of visited nodes.

It was also noted that changing the order of the items to produce Short-Wide tree has improved the performance of the TDF. The use of short-wide itemset tree helped the TDF to identify the support of the candidates faster. The reason for that: First, since the tree is wide, the CountSet algorithm prunes a lot of horizontal branches in the tree while counting the support. Second, since the tree is short, the CountSet does not traverse a lot of tree level to count the support of the candidates.
6 Conclusion and Future Work

In this paper, we are trying to overcome a drawback in the currently presented algorithms for mining the frequent itemsets based on the dataset-transformation approach, where we can not reuse the constructed structure in ad-hoc mining queries. We selected the ItemSet Tree, which was originally proposed for ad-hoc mining queries, to use it in the process of mining the frequent itemsets in the TDF algorithm.

To speedup the support counting in the IST: first; a suggestion to reduce its size was presented by removing the non-supported items. Second, the GIB information was presented to reduce the False Sub-Tree checking. To find the frequent itemsets, the proposed algorithm TDF depends on generating the candidates in groups that share the leading itemset. Exploring the frequent itemset search space in depth-first will help in getting the candidates in this shape.

Several experiments have been conducted to compare the performance of the TDF with the FP-growth, which is a recent algorithm for finding the frequent itemsets. The study showed that TDF outperforms FP-growth for higher values of (T) and lower values for (I). It was also noted that sorting the items in descending order based on their support to produce SCTShort, improves effectively the performance of TDF because searching in shorter trees minimizes the number of tree nodes visited.

More research can be done to find another grouping scheme that can improve the process of mining the frequent itemsets. Also, employing a breadth-first traversal for the frequent itemsets search space could improve the performance, but we should use another grouping scheme for the candidates.

Since we used the IST in the process of support counting, we can also use it in other data mining fields that require support counting like dynamic data mining, constrained mining …etc. The problem is to find the best way to group the itemsets to find its support.

7 References


