Interactive Discovery of Statistically Significant Itemsets

Philippe Fournier-Viger[1(✉)], Xiang Li[2], Jie Yao[1], and Jerry Chun-Wei Lin[2]

[1] School of Humanities and Social Sciences, Harbin Institute of Technology (Shenzhen), Shenzhen, Guangdong, China
philfv8@yahoo.com, julie_j_yao@163.com

[2] School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen, Guangdong, China
leeideal93@gmail.com, jerrylin@ieee.org

Abstract. Frequent Itemset Mining (FIM) is a fundamental data mining task, which consists of finding frequent sets of items in transaction databases. However, traditional FIM algorithms can find lot of spurious patterns. To address this issue, the OPUS-Miner algorithm was proposed to find statistically significant patterns, called productive itemsets. Though, this algorithm is useful, it cannot be used for interactive data mining, that is the user cannot guide the search toward items of interest using queries, and the database is assumed to be static. This paper addresses this issue by proposing a novel approach to process targeted queries to check if some itemsets of interest to the user are non redundant and productive. The approach relies on a novel structure called Query-Tree to efficiently process queries. An experimental evaluation on several datasets of various types shows that thousands of queries are processed per second on a desktop computer, making it suitable for interactive data mining, and that it is up to 22 times faster than a baseline approach.

Keywords: Itemset mining · Productive itemsets · Query-Tree Pattern

1 Introduction

Frequent Itemset Mining (FIM) [6], consists of finding frequently occurring patterns in databases to understand the data, and support decision-making. The input of FIM is a customer transaction database, where each transaction is a set of items purchased by a customer. An itemset (set of items) is said to be frequent if its support (number of transactions where it appears) is no less than a predefined *minsup* threshold, set by the user. The task of FIM is to enumerate all frequent itemsets in a transaction database. Although FIM is useful in many domains [8], it can find a large number of patterns, depending on how the *minsup* threshold is set. If the *minsup* threshold is set too high, no patterns are found. But if it is set too low, millions of patterns are found, and algorithms may become

© Springer International Publishing AG, part of Springer Nature 2018
M. Mouhoub et al. (Eds.): IEA/AIE 2018, LNAI 10868, pp. 101–113, 2018.
https://doi.org/10.1007/978-3-319-92058-0_10

very slow and consume a large amount of memory. To set the *minsup* threshold, a user typically run a FIM algorithm several times with different parameter values to find enough but not too many patterns. It was shown that traditional FIM algorithms can find a lot of spurious patterns that are frequent but are uninteresting to the user because their support (frequency) can be explained by the support of their subsets [2,7]. In other words, a pattern can be frequent just because the items that it contains are frequent, while items in that pattern may not be correlated. Analyzing a set of patterns containing many spurious patterns is both inconvenient and time-consuming for the user. To address this problem, an emerging topic is to find patterns that are statistically significant [2,7]. One of the most popular algorithms to find statistically significant frequent itemsets is OPUS-Miner [2]. It discovers a set of patterns called *non-redundant productive itemsets* by applying the Fisher test to determine if the bipartitions of each pattern are significantly correlated. In recent years, the concept of productive itemsets has been adapted for several applications such as discovering periodic patterns [9] and sequential patterns [10]. Although mining productive itemsets is useful as it only shows itemsets that are significant to the user, it has several limitations. First, OPUS-Miner outputs the k patterns that are productive and have the highest lift or leverage, where k is a user-specified parameter. But if k is set to a small value, patterns that are not top-k patterns will not be found, and if k is set to a very large value, the algorithm may find these patterns but become very slow and consume a huge amount of memory. Second, OPUS-Miner is not designed for interactive data mining, as the user cannot guide the search of patterns. If the user wants to know if a specific itemset is productive, he may have to run OPUS-Miner with a large k value, hoping to find this itemset among the top-k patterns. If it is not a top-k pattern, the user may then have to run the algorithm againwith a different value of k, which is inconvenient and time consuming. In fact, OPUS-Miner is unable to process queries to determine if some specific itemsets are non-redundant and productive.

Supporting targeted queries is key to the development of interactive data mining systems as it allows the user to perform queries to search for specific patterns, look at the results, and then send refined queries to search for more interesting patterns [1]. For example, targeted queries can let users quickly search for patterns containing only some items, instead of considering all items [3–5]. To efficiently process targeted queries for FIM in the context of static or incremental databases, the Itemset Tree (IT) data structure was proposed [3], as well as improved versions such as the Min-Max Itemset-Tree [4] and Memory Efficient Itemset-Tree [5]. The IT is a tree structure, which can be incrementally updated and efficiently queried. The IT structure allows processing several types of targeted queries such as (1) calculating the frequency of a given itemset, and (2) finding all frequent itemsets subsuming a set of items and their support. The IT structure has various applications such as predicting missing items in shopping carts in real-time [9]. However, it is not designed to find significant patterns, and thus can also find many spurious patterns.

This paper addresses these limitations of previous work by proposing a novel approach called IDPI (Interactive Discovery of Productive Itemsets) to support targeted queries about non redundant productive itemsets in dynamic databases. An efficient algorithm is proposed to answer queries to check if some itemsets are non redundant and productive in a database. This algorithm relies on a novel structure called Query-Tree. To evaluate the proposed approach, experiments have been carried on multiple real-life datasets used in the FIM litterature.

The rest of this paper is organized as follows. Section 2 introduces preliminaries and defines the problem. Section 3 presents the proposed approach. Section 4 presents the experimental evaluation. Section 5 draws the conclusion.

2 Preliminaries and Problem Statement

The problem of frequent itemset mining is defined as follows [6,8]. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of items (symbols). A *transaction database* is a set of transactions $D = \{T_1, T_2, \ldots, T_m\}$, where each transaction T_z $(1 \leq z \leq m)$ is a subset of items purchased by a customer $(T \subseteq I)$, and z is a unique Transaction IDentifier (TID). An unordered set of items $X \subseteq I$ is said to be an *itemset*. An itemset X is said to be of length r or a r-itemset if it contains r items. The cover of an itemset X in a database D is the set of transaction containing the itemset X, that is $cov(X, D) = \{T | T \in D \wedge X \subseteq T\}$. The *support* of an itemset X in a database D is the number of transactions that contain X, that is $sup(X, D) = |cov(X, D)|$, denoted as $sup(X)$ when the context is clear. For example, consider the transaction database D of Fig. 1, which contains five items (a, b, c, d, e) and five transactions $(T_1, T_2, \ldots, T_5)$. The first transaction represents the set of items a and d. The cover of the itemset $\{a, b\}$ is $cov(\{a, b\}, D) = \{T_3, T_4\}$, and the support of $\{a, b\}$ is $sup(\{a, b\}) = \{T_3, T_4\}$.

TID	Items
T_1	{a, d}
T_2	{b, e}
T_3	{a, b, c}
T_4	{a, b, d}
T_5	{b, e}

Fig. 1. A transaction database

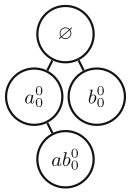

Fig. 2. The query tree of $\{a, b\}$

The traditional problem of frequent itemset mining [6,8] is to find all frequent itemsets in a transaction database, that is all itemsets having a support that is no less than a user-defined *minsup* threshold. To find patterns that are not spurious, an emerging problem in data mining is to find patterns that are statistically significant. For this purpose, Webb et al. proposed to discover the set of *productive itemsets* in a database, defined as follows [2].

Definition 1 (productive itemset). *Let there be an itemset X. Two itemsets $\{Y, Z\}$ are said to be a bipartition of X if $Y \cup Z = X \wedge Y \cap Z = \emptyset \wedge Y \neq \emptyset \wedge Z \neq \emptyset$. Let $bipart(X)$ be the set of all bipartitions of X. Let $P(X \subseteq R)$ be the probability that an itemset X is drawn from the same distribution as the database D. An itemset X of length $k \geq 2$ is said to be productive if $P(X \subseteq R) > \max_{\{Y,Z\} \in bipart(X)} P(Y \subseteq R) \times P(Z \subseteq R)$.*

The constraint of productivity is useful as it ensures that all items within a productive itemset contribute to the support of the itemset. For example, the itemset $\{alchool, liver\_cancer\}$ is productive as the probability (support) of drinking alchool and having liver cancer is higher than what would be expected if those items were not correlated. On the other hand, the itemset $\{alchool$, $liver\_cancer$, $black\_hair\}$ is not productive because although this itemset may be frequent the bipartition $\{\{alchool, liver\_cancer\}, \{black\_hair\}\}$ is not correlated (the support of that bipartition can explain the support of the itemset). Thus, mining productive itemsets can filter many spurious patterns. To ensure that productive itemsets are also statistically significant, Opus-Miner applies the Fisher exact test [2]. Besides, OPUS-Miner filters redundant itemsets to show a small set of non redundant productive patterns to the user [2].

Definition 2. *An itemset X is said to be non redundant if there does not exists a proper subset Y of X having the same support, i.e. $\nexists Y \subset X | sup(X) = sup(Y)$.*

The concept of non redundant patterns (also called *generators* or *key patterns*) [11] is interesting according to the Minimum Description Length principle since it represents the smallest sets of items that are common to sets of transactions. For example, in market basket analysis, generator itemsets represent the smallest sets of items common to group of customers.

Discovering non redundant productive itemsets in a database is a very time-consuming task. The reason is that to determine if an itemset X is non redundant and productive, it is necessary to compute the support of all its bipartitions, that is the support of all its non empty subsets. Generally, an itemset X has $2^{|X|} - 1$ non empty subsets. Thus, to determine if a 6-itemset is productive, it is necessary to compute the support of $2^{|6|} - 1 = 63$ itemsets. Besides, as mentioned in the introduction, another important issue is that the state-of-the-art OPUS-miner algorithm is a batch algorithm, which can only be applied to find the top-k productive itemsets in a static database. Thus, if one wants to determine if an itemset X is productive, the user must run the algorithm with a value of k that is large enough to ensure that the itemset X will be among the top-k itemsets, which is very inconvenient as it may require to run the algorithm multiple times and can cause the algorithm to have long execution times. To address this problem, this paper defines the problem of processing queries to determine if an itemset is non-redundant and productive.

Definition 3 (Problem statement). *Given a database D, the problem of interactive discovery of non-redundant productive itemsets is to efficiently answer queries of the form "Is an itemset X productive and non redundant?".*

3 The Proposed IDPI Approach

To efficiently process queries, this paper proposes the IDPI approach. It consists of three components: (1) a variation of the Itemset-Tree structure [3] to compress the database, (2) a novel structure called Query Tree to accumulate information about the support of itemsets to answer queries, and (3) an algorithm that efficiently answer queries by comparing the two aforementioned structure.

3.1 Compressing the Database Using the Itemset-Tree Structure

The proposed approach compresses the database using a variation of the Itemset-Tree structure called Memory Efficient Itemset-Tree (MEIT) [5]. The Itemset-Tree structure was designed for interactive frequent itemset mining an can be updated incrementally to support dynamic databases.

The Itemset-Tree structure. An IT is built for a database D by inserting each transaction T of D into the IT. An IT node g has three fields: (1) $g.itemset$ stores an itemset, (2) $g.sup$ stores its support and (3) $g.childs$ stores pointers to the node's childs (if it is not a leaf). Each itemset stored in an IT node is a transaction or the intersection of one or more transactions. An IT initially only contains a root node denoted as $IT.root$, which stores the empty set, i.e. $IT.root.itemset = \emptyset$. Each itemset stored in an IT node is sorted according to a total order such as the lexicographical order. Based on that order, an itemset $X = \{a_1, a_2, \ldots a_k\}$ is said to share the leading items with an itemset $Y = \{b_1, b_2, \ldots b_l\}$ if there exists an integer $1 \leq v \leq argmin(\{k, l\})$ such that $a_1 = b_1, a_2 = b_2, \ldots a_v = b_v$.

Constructing an Itemset-Tree. Initially, an IT only contains the root node. The algorithm *Insert-Transaction* is applied for inserting each transaction of D in the IT (Algorithm 1). It was shown that the expected cost of this algorithm is approximately O(1) [3]. As example, Fig. 3 illustrates the construction of an IT by successively inserting each transaction of the database of Fig. 1. Figure 3 (A) shows the tree after the insertion of transaction $\{a, d\}$. A child node has been added to the root to store the itemset $\{a, d\}$ with a support of 1. Figure 3 (B) shows the tree after the insertion of transaction $\{b, e\}$. A child node has been added to the root, representing itemset $\{b, e\}$, with a support of 1. Figure 3 (C) shows the tree after the insertion of transaction $\{a, b, c\}$. Since the itemset $\{a, b, c\}$ shares the leading item $\{a\}$ with the node $\{a, d\}$, a new node $\{a\}$ has been inserted with a support of 2, having $\{a, d\}$ and $\{a, b, c\}$ as childs. The same process is repeated for the other transactions. Figure 3(D), (E) and (F) show the tree after the insertion of transactions $\{a, b, d\}$, $\{b, e\}$ and $\{b, d\}$, respectively.

In the proposed IDPI approach, a variation of the IT called MEIT [5] is used. This data structure is designed to reduce the memory usage of the IT. The difference between the MEIT and IT is that in a MEIT node, items from the parent node are not stored. For example, Fig. 3(F) shows the MEIT corresponding to the IT of Fig. 3(E). It was shown that using a MEIT instead of an IT can reduce memory usage by up to 50% [5]. A reason for using a MEIT in the proposed approach is that once it is constructed, it can be used to efficiently

Algorithm 1: Insert-Transaction

input: T: a transaction, IT: an itemset-tree

1 $IT.root.sup \leftarrow IT.root.sup + 1$;
2 **if** $T = IT.root.it$ **then** exit;
3 Let ITT be a sub-tree of $IT.root$ such that $ITT.root.it$ and T share some leading items;
4 **if** ITT *does not exist* **then** Add a child node g to $IT.root$ such that $g.itemset = T$ and $g.sup = 1$;
5 **else if** $ITT.root \subset T$ **then** $Construct(T, ITT)$;
6 **else if** $T \subset ITT.root.it$ **then** Create a new node g as a son of $IT.root$ and a father of $ITT.root$ where $g.itemset = T$ and $g.sup = ITT.root.sup + 1$;
7 **else** Create a node g as a father of $ITT.root$ such that $g.itemset = T \cap ITT.root$, $g.sup = ITT.root.sup + 1$. Moreover, create a node h as a son of g, such that $h.itemset = T$ and $h.sup = 1$;

find the support of any itemset X, which is required to determine if an itemset is productive and non-redundant. Moreover, a MEIT can be updated in real-time by inserting new transactions if needed, thus to support interactive data mining. Due to space limitation, the reader is referred to [5] for the algorithm for calculating the support of an itemset using a MEIT. Another reason for using a MEIT instead of an IT is that the MEIT facilitates query answering using the proposed Query Tree structure, described in the next subsection.

3.2 Representing Queries Using the Query-Tree Structure

The second component of the proposed IDPI approach is a novel structure called *Query Tree (QT)*. It is designed to increase the performance of support counting using a MEIT. Let there be a query to check if an itemset X is productive and non redundant. To answer this query, it is necessary to compute the support of all its non empty subsets. Using the traditional approach to count support using a MEIT, a query for support counting would need to be performed for each of those itemsets, that is the MEIT would need to be traversed multiple times, which is inefficient. A better approach proposed in this paper is to store X and all its subsets in a Query Tree. Then, this structure is used to quickly calculate the support of all these itemsets by traversing the MEIT only once using a novel query processing algorithm (described in the next sub-section).

The Query Tree structure. A QT is a tree where each node g has four fields: (1) $g.itemset$ stores an itemset, (2) $g.sup$ stores its support, (3) $g.pos$ stores a position (an integer initialized to zero), and (4) $g.childs$ stores a list of pointers to child nodes of g (if g is not a leaf node). It is to be noted that $g.child$ is sorted according to a total $\succ$ order such as the lexicographical order. Initially, a Query-Tree contains a single node, which is the empty set.

Constructing a Query Tree. The QT of an itemset X is built by inserting X and each non-empty subset of X in a QT. This is done by applying a modified

version of Algorithm 1 for each non empty subset of X. The modified algorithm does not update the support field of each node (it remains equal to 0). Moreover, the algorithm sorts the child nodes of each node according to the $\succ$ order. The structure of a QT is similar to that of an IT. The differences are that (1) a QT stores itemsets instead of transactions, (2) the *childs* field is sorted, (3) the *sup* field is used differently, and (4) the *pos* field is introduced for matching a query to an MEIT to answer queries (described in the next subsection). For example, the QT, constructed to determine if the itemset $X = \{a, b\}$ is productive and non redundant, is shown in Fig. 2. In that figure, each node g contains an item, where its subscript and superscript indicate $g.sup$ and $g.pos$, respectively.

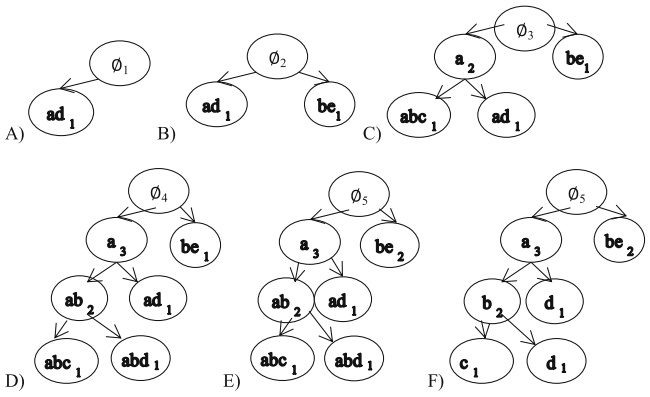

Fig. 3. Construction of the Itemset-Tree for the database of Fig. 1.

3.3 Processing Queries Efficiently Using a Query-Tree

The third component of the proposed approach is a novel algorithm to process queries (Algorithm 2), which takes as input a transaction database D and a set of queries to be processed. Let there be a query to check if an itemset X is productive and non redundant. To process the query, the first step is to build the Query Tree, as described in the previous subsection (a QT can store multiple queries) (line 2). This process creates a list QL initially containing the root of the Query Tree. Then, the MEIT is compared with the Query Tree to calculate the support of X and its subsets (line 3) by calling the $GetSupportUsingQueryTree$ procedure. This procedure stores these support values in the Query Tree nodes. After collecting all the support values, the Fisher test is applied to each bipartition of X to determine if the support of X is significantly different from the expected support of its bipartitions, and the support of X is compared with that of its subsets to determine if X is non redundant (line 5). For each application of the Fisher test, a p value is generated. The algorithm then shows the p values of the itemset X to the user, and indicates if the itemset is non redundant and productive for $p \leq 0.05$ (line 6).

Algorithm 2: IDPI

input: D: a transaction database, QD: a set of queries

```
1 rootMEIT = buildMEIT(D);        // Build MEIT (if not previously built)
2 QL = buildQueryTree(QD);                          // Build the Query Tree
3 GetSupportUsingQueryTree(QL,rootMEIT);      // Get support of itemsets
4 foreach query tree node QTN ∈ QL do
5 |   Check if QTN.itemset is productive and non redundant using QT;
6 |   Output the result for QTN;                                // Output itemset
7 |   Insert all child nodes of QTN in QL;
8 end
```

The $GetSupportUsingQueryTree$ procedure (Algorithm 3) takes as input (1) a list of query tree nodes QL sorted according to the $\succ$ order (initially containing the root node), and (2) a MEIT node (initially the root). The procedure compares the MEIT with the Query Tree by performing a depth-first search on the MEIT to update the support values of all itemsets in the Query Tree. Each node in the MEIT is traversed at most once.

To compare a QT node QTN with a MEIT node ITN, a challenge is that an itemset stored in a MEIT node is not completely stored (for example, the leftmost node of Fig. 3(F) represents the itemset $\{a, b, c\}$ but only $\{c\}$ is stored in that node), while itemsets in QT nodes are completely stored. To be able to compare these two representations of itemsets, we introduce the concept of *suffix* of an itemset $X = \{a_1, a_2, \ldots a_k\}$ w.r.t a position pos, which is defined as $suf(X, pos) = \{a_{pos}, a_{pos+1}, \ldots a_k\}$. For a node QTN, the pos field stored in QTN indicates that only the items $suf(QTN.itemset, QTN.pos)$ should be compared with the items in ITN. For example, if $QTN.itemset = \{a, b, c\}$ and $QTN.pos = 2$, and $ITN = \{b\}$, it indicates that only the items $\{b, c\}$ of $QTN.itemset$ should be compared with ITN. For the sake of brevity, let $QTNsuffix$ denotes $suf(QTN.itemset, QTN.pos)$. When comparing a QT node QTN and a MEIT node ITN, five distinct cases are encountered:

Case 1. If $suf(QTN.itemset, QTN.pos) \subseteq ITN.itemset$, then it means that $QTN.itemset$ is included in the itemset represented by ITN. In that case, the support of QTN is incremented by the support of ITN. Moreover, each child node $QTNC$ of QTN is added to the list QL (while preserving the $\succ$ order) with pos equal to the number of items in QTN, so that it will be processed later. Moreover, QTN is removed from QL.

Case 2. If $QTNSuffix$ has some items in common with $ITN.itemset$, and all other items of $QTNSuffix$ are greater than the largest item in $ITN.itemset$ according to the $\succ$ order, it means that $QTN.itemset$ is not included in the itemset represented by ITN but that it may be included in those represented by ITN's childs. In this case, the pos value of QTN is saved in a map, and then pos is incremented by the number of items that QT and ITN have in common.

Case 3. If there exists an item i in $QTNsuffix$ that is not in $ITN.itemset$ and i is smaller than the last item in $ITN.itemset$ according to the $\succ$ order, it means that QTN is not included in the itemset represented by ITN and those represented by its childs. Hence, QTN is removed from the QL list.
Case 4. If the first item in $QTN.itemset$ is greater than the last item in ITN according to the $\succ$ order, it means that the itemsets represented by QTN and its siblings in QL may be included those represented by ITN and its childs. In that case, QTN and its siblings must remain in QL to be processed next when considering ITN's childs.
Case 5. Otherwise, it is necessary to compare the siblings that succeed QTN according to the $\succ$ order with ITN.

It can be proven that the *GetSupportUsingQueryTree* procedure is correct to calculate the support of itemsets stored in a Query Tree, although the proof is omitted due to space limitation.

Algorithm 3: GetSupportUsingQueryTree

```
input: QL: a QT node list (initially containing only the QT root node) ITN: a MEIT node
       (initially the root node)
1  if QL is empty then exit;
2  QTNsuffix = suf(QTN.itemset, QTN.pos);
3  foreach QTN ∈ QL do
4      if QTNsuffix ⊆ ITN.itemset                                          // Case 1
5      then
6          QTN.sup+=ITN.sup; foreach child QTNC of QTN do
7              QL.add(QTNC);                  // while preserving the ≻ order in QL
8              tjmap[QTNC] = QTNC.pos;
9              QTNC.pos = |QTN.itemset|;
10         end
11         QL.delete(QTN);
12     end
13     else if
       QTNsuffix∩ITN.itemset ≠ ∅∧∀i ∈ QTNsuffix\ITN.itemset, i > ITN.itemset.last
       // Case 2
14     then
15         tjmap[QTNC.itemset] = QTNC.pos;
16         QTN.pos += |QTNsuffix ∩ ITN.itemset|;
17     end
18     else if ∃i ∈ QTNsuffix \ ITN.itemset ∧ i < ITN.itemset.last then
19         QL.delete(QTN);                                                 // Case 3
20     end
21     else if QTN.itemset.first > ITN.itemset.last then break;            // Case 4
22     else continue;                                                      // Case 5
23 end
24 foreach ITNC ∈ ITN do GetSupportUsingQueryTree(QL, ITNC);
25 foreach QTN ∈ tjmap do QTN.pos = tjmap[QTN];
```

The process of calculating the support of itemsets using a Query Tree is illustrated with an example. Consider Fig. 4. It shows how the Query Tree of $\{a, b\}$ is updated by traversing the MEIT using a depth-first search. Five steps ((A), (B), (C), (D), (E)) are illustrated corresponding to the comparison of the QT with the five nodes of the MEIT, respectively. The first and second lines show

the content of the QL and Query Tree before the comparison, respectively. The third line shows the MEIT, where the node marked in light gray is the current node ITN used for the comparison. Initially (Fig. 4(A)), all values in the QT are equal to zero. The list QL contains only the root of the QT, representing the empty itemset. This current node, called QTN, is compared with the root of the MEIT, called ITN. Since $QTN.itemset \subseteq ITN.itemset$ ($\varnothing \subseteq \varnothing$), case 1 is applied. Thus, the support of ITN is added to the support of QTN, the childs of QTN are inserted in QL with *pos* equal to the number of items in QTN, and QTN is removed from QL. After this, QL contains two nodes: a_0^0 and b_0^0. The next node from QL to be considered as QTN is a_0^0. Because the first item of $\{a\}$ is greater than the last item of ITN, case 4 is applied (the main loop is stopped). The current state of QL and the QT are shown in Fig. 4(B). Next, the $GetSupportUsingQueryTree$ is recursively called to compare nodes in QL with the first child of ITN. Thus, the node a_3 becomes the node ITN. This current node of QL, $QTN = a_0^0$, is compared with ITN. Because $QTN.itemset \subseteq ITN.itemset$ ($\{a\} \subseteq \{a\}$), case 1 is applied. Thus, the support of ITN is added to the support of QTN, the childs of QTN are inserted in QL with *pos* equal to the number of items in QTN, and QTN is removed from QL. After this, QL contains two nodes: ab_0^1 and b_0^0. The next node from QL to be considered as QTN is ab_0^1. Case 5 is applied, and thus the next node in QL is processed, that is b_0^0. Because the first item of $\{b\}$ is greater than the last item of $ITN = \{a\}$, case 4 is applied (the main loop is stopped). The current state of QL and the QT are shown in Fig. 4(C). The same process is repeated for the other nodes of the MEIT following the depth-first search. The final QT is shown in Fig. 4(F). From this tree, it is found that the support of itemsets $\{a\}$, $\{b\}$ and $\{a, b\}$ are 3, 2, and 4, respectively. Thus, $\{a, b\}$ is non redundant, and the Fisher test can be applied using these support values to check if $\{a, b\}$ is productive. Note that in this example, the database is too small to determine if an itemset is productive.

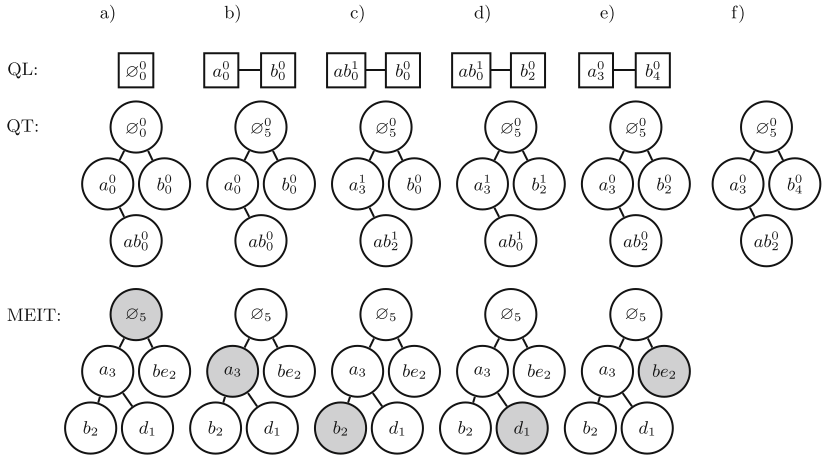

Fig. 4. Updating the query tree of $\{a, b\}$ by traversing the MEIT

4 Experimental Evaluation

To evaluate the proposed IDPI approach, an experiment was performed to compare its performance with that of a baseline approach using an IT to count the support of itemsets (as presented in Sect. 3.1). The goal is to check if the novel QT structure reduces the time to process queries, and generally how many queries may be processed per second on a desktop computer on benchmark datasets. Both approaches were implemented in C++. The experiment was performed on a computer equipped with a Xeon E3-1270 processor running Windows 10 and 64 GB of free RAM. The experiment was carried on real-life and synthetic datasets commonly used in the frequent itemset mining literature, namely Accidents, Chess, Connect, Mushrooms, Pumsb and Retail. In the experiment, n random queries were generated for each dataset, where n was varied from 1000 to 3000, and the length of itemsets in queries was varied from 2 to 10. IDPI and the baseline approach were applied on each set of random queries and the total execution time was measured in seconds. Results are shown in Table 1.

Table 1. Time for processing n queries using IDPI and the baseline approach

Dataset	Itemset length	Time for processing n queries (s)					
		IDPI			Baseline (IT)		
		1000	2000	3000	1000	2000	3000
Mushroom	2–4	1	2	3	9	17	36
	5–7	4	8	14	33	66	121
	8–10	15	35	57	86	194	260
Connect	2–4	11	19	30	77	162	247
	5–7	56	107	150	374	743	1192
	8–10	297	698	926	1501	3361	5503
Accident	2–4	13	18	23	184	358	525
	5–7	35	59	82	511	1063	1507
	8–10	83	160	230	1007	2240	3345
Pumsb	2–4	11	23	35	111	220	328
	5–7	35	86	139	262	522	777
	8–10	168	448	782	461	922	1398
Retail	2–4	27	58	110	72	162	248
	5–7	71	217	434	213	451	704
	8–10	244	861	1402	435	968	1214
Chess	2–4	1	2	2	4	9	48
	5–7	7	15	20	29	56	86
	8–10	50	93	132	159	307	450

It can be observed that the proposed IDPI approach is up to 22 times faster than the baseline approach. IDPI is faster in all cases except on Retail when $n = 3000$ and the itemset length is between 8 to 10. The reason is that in that case the QT becomes very large since for example, a 10-itemset has $2^{10} = 1024$ subsets. However, it can be argued that itemsets of length 8 to 10 are unlikely to be productive (since all their bipartitions must be positively correlated), and long itemsets are rarely useful in practice as they represent very specific cases. It was also found that the proposed approach can process up to 1000 queries per second, which makes it suitable for interactive pattern mining. Generally, the speed of processing queries is influenced by the length of itemsets, whether they share subsets, and the nature of the database. Note that if a QT is reused multiple times on the same database (e.g. on different days), the time for query processing is reduced because the QT is not rebuilt. Overall, the performance of the proposed approach is found to be very satisfying.

5 Conclusion

This paper has defined the problem of interactively discovering non redundant and productive itemsets in transaction databases. To efficiently process targeted queries performed by users to check if some patterns are non redundant and productive, an approach called IDPI was proposed. The approach relies on the MEIT and a novel Query Tree structure to efficiently process queries. Experimental results show that the IDPI approach is up to 22 times faster than a baseline approach and can process up to 1000 queries per second on a desktop computer, making it suitable for interactive pattern mining. In future work, the IDPI approach will be improved to support additional query types.

References

1. Han, J., Pei, J., Kamber, M.: Data Mining: Concepts and Techniques. Elsevier, Waltham (2011)
2. Webb, G.I., Vreeken, J.: Efficient discovery of the most interesting associations. ACM Trans. Knowl. Discov. Data **8**(3), 15 (2014)
3. Kubat, M., Hafez, A., Raghavan, V.V., Lekkala, J.R., Chen, W.K.: Itemset trees for targeted association querying. IEEE Trans. Knowl. Data Eng. **15**(6), 1522–1534 (2003)
4. Lavergne, J., Benton, R., Raghavan, V.V.: Min-max itemset trees for dense and categorical datasets. In: Chen, L., Felfernig, A., Liu, J., Raś, Z.W. (eds.) ISMIS 2012. LNCS (LNAI), vol. 7661, pp. 51–60. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34624-8_6
5. Fournier-Viger, P., Mwamikazi, E., Gueniche, T., Faghihi, U.: MEIT: memory efficient itemset tree for targeted association rule mining. In: Motoda, H., Wu, Z., Cao, L., Zaiane, O., Yao, M., Wang, W. (eds.) ADMA 2013, Part II. LNCS (LNAI), vol. 8347, pp. 95–106. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-53917-6_9

6. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings of 20th International Conference on Very Large Databases, pp. 487–499. Morgan Kaufmann, Santiago de Chile (1994)
7. Llinares-López, F., Sugiyama, M., Papaxanthos, L., Borgwardt, K.: Fast and memory-efficient significant pattern mining via permutation testing. In: Proceedings of 21th ACM International Conference on Knowledgs Discovery and Data Mining, pp. 725–734. ACM (2015)
8. Fournier-Viger, P., Lin, J.C.-W., Vo, B., Chi, T.T., Zhang, J., Le, H.B.: A survey of itemset mining. WIREs Data Mining Knowl. Discov. **7**(4), e1207 (2017). https://doi.org/10.1002/widm
9. Nofong, V.M.: Discovering productive periodic frequent patterns in transactional databases. Ann. Data Sci. **3**(3), 235–249 (2016)
10. Petitjean, F., Li, T., Tatti, N., Webb, G.I.: Skopus: mining top-k sequential patterns under leverage. Data Mining Knowl. Discov. **30**(5), 1086–1111 (2016)
11. Fournier-Viger, P., Wu, C.-W., Tseng, V.S.: Novel concise representations of high utility itemsets using generator patterns. In: Luo, X., Yu, J.X., Li, Z. (eds.) ADMA 2014. LNCS (LNAI), vol. 8933, pp. 30–43. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-14717-8_3