Finding Frequent Closed Itemsets in Sliding Window in Linear Time

Junbo CHEN, Student Member, Bo ZHOUa), Lu CHEN†, Nonmembers, Xinyu WANG†, Member, and Yiqun DING†, Nonmember

SUMMARY One of the most well-studied problems in data mining is computing the collection of frequent itemsets in large transactional databases. Since the introduction of the famous Apriori algorithm [14], many others have been proposed to find the frequent itemsets. Among such algorithms, the approach of mining closed itemsets has raised much interest in data mining community. The algorithms taking this approach include TITANIC [8], CLOSET+ [6], DCI-Closed [4], FCI-Stream [3], GC-Tree [15], TGC-Tree [16] etc. Among these algorithms, FCI-Stream, GC-Tree and TGC-Tree are online algorithms work under sliding window environments. By the performance evaluation in [16], GC-Tree [15] is the fastest one. In this paper, an improved algorithm based on GC-Tree is proposed, the computational complexity of which is proved to be a linear combination of the average transaction size and the average closed itemset size. The algorithm is based on the essential theorem presented in Sect.4.2. Empirically, the new algorithm is several orders of magnitude faster than the state of art algorithm, GC-Tree.

key words: frequent itemsets, closed itemsets, online algorithm, association rules, data mining

1. Introduction

Frequent closed itemsets is a complete and condensed representation for all the frequent itemsets. Therefore, the study of the frequent closed itemsets has raised much interest in the data mining community. Extensive research has been carried out in this area, in the following they are split to four categories:

Both A-Close [12] and TITANIC [8] exploit a level-wise process to discover closed itemsets through a breadth-first search strategy. In each iteration, they try to search for candidates of MGs (Minimum Generators) with the help of search space pruning technique, and then verify them. Finally the MGs are used to generate all the closed itemsets. Usually these algorithms are required to scan the whole dataset many times.

CLOSET [13] and CLOSET+ [6] try to project the global extraction context to some smaller sub-contexts with the help of a high compact data structure, termed FP-Tree, and then apply frequent closed itemsets mining process recursively on these sub-context in a depth-first manner. Better performance can be achieved than the adoption of A-close and TITANIC algorithms. FP-Close [17] is an improvement of CLOSET+, which proposed a novel search space pruning technique, termed “array”. LCM [18] also traverses the search space in a depth-first manner, and it could find all frequent closed itemsets in polynomial time per itemset.

CHARM [7] and DCI-Closed [4] exploit hybrid techniques which try to use the properties of both previous mentioned techniques. Due to a data structure called IT-Tree, CHARM simultaneously explores both the closed itemset space and transaction space, with the depth-first search strategy, and generates one candidate each time, then with tidset intersection and subsumption checking, it will see whether the candidate is closed. DCI-Closed could be considered an improvement of CHARM.

FCI-Stream [3], GC-Tree [15] and TGC-Tree [16] are online algorithms which perform the closure checking over data stream sliding windows. FCI-Stream uses a data structure called DIU-tree to store all the closed itemsets discovered so far. With a specific search space pruning technique, it tries to perform the time consuming closure checking operation only when it is really needed. GC-Tree maintains a tree like structure in the memory, which is also called GC‘Tree’. Each closed itemset is represented as a node in the tree, and every path from the root to a specific node in the tree is an order preserving sequence (the definition of “order preserving” could be found in DCI-Closed [4]) of closed itemsets. TGC-Tree [16] is a new algorithm based on GC-Tree which tries to trace the closed itemsets and closed transaction sets simultaneously. Actually, it maintains two GC-Trees, one of them keeps all the closed itemsets, called I-Tree, and the other keeps all the closed transaction sets, called T-Tree.

Although GC-Tree and its variant algorithm, TGC-Tree, could handle transactions which have relatively small number of items quite efficiently, their super-linear scaling behavior with $\beta$ makes them inefficient or even intractable on very large datasets. In this paper, we propose a new algorithm, the computational complexity of which is proved to be a linear combination of the average transaction size and the average closed itemset size. Obviously, this is very attractive in the real world datasets.

*Actually, the algorithm is named after the data structure it used.

**Let $\beta$ denotes the average size of the transactions. Detailed computational complexity analysis is listed in Sect. 4.1.
The rest of this paper is organized as follows. Section 2 formally defines the concept of closed itemsets and describes the notations to be used throughout the paper. Section 3 introduces the GC-Tree algorithm. Section 4 describes our method to improve it. Section 5 introduces some optimizations. The performance evaluation is depicted in Sect. 6. Finally, comes conclusion of this paper, Sect. 7.

2. Problem Definition

Let \( I = \{i_1, i_2, \ldots, i_n\} \) be a set of items, \( D = \{t_1, t_2, \ldots, t_n, \ldots\} \) be a set of infinite stream of transactions, and \( L = \{t_0, t_1, \ldots, t_M\} \) be a sliding window of \( D \) which contains the recent \( M+1 \) transactions. A subset \( I \subseteq I \) is called an itemset. Each transaction \( t \in D \) is a set of items in \( I \). There’s an unique transaction id for every \( t \). Given a set of transactions \( T \subseteq D \), it can be represented by a tid_list, and the support of an itemset \( I \) in \( T \) is the number of transactions that contain \( I \), denoted as \( s(I) \).

The concept of closed itemsets is based on the following functions:

\[
\begin{align*}
  f(T) &= \{i \in I \mid \forall t \in T, i \in t\} \\
  g(I) &= \{t \in D \mid \forall i \in I, i \in t\}
\end{align*}
\]

Definition 1 An itemset \( I \) is said to be closed if and only if \( C(I) = f(g(I)) = f \circ g(I) = I \) where the composite function \( C = f \circ g \) is called a Galois operator or a closure operator.

As a closure operator, \( C \) has the following properties:

Property 1 \( C(X) \supseteq X \)

Property 2 \( Y \subseteq X \Rightarrow C(Y) \subseteq C(X) \)

Property 3 \( C(C(X)) = C(X) \)

Mining all the frequent closed itemsets from the current sliding window of data stream \( D \) requires to discover all the closed itemsets which have higher support than a given threshold \( \text{min supp} \) in the sliding window.

The closure operator, \( C \), defines a set of equivalence classes over all the itemsets. Every element belongs to the same equivalence class has the same support (actually, has the same corresponding transaction set). Each equivalence class has one and only one closed itemset, which is the biggest one in the equivalence class. In this way, the closed itemsets represent all the equivalence classes, and in turn, represent all the itemsets. Our algorithm tries to find exactly one generator in every equivalence class, then calculates the closure of them. There’re 2 popular lemmas which are wildly exploited.

Lemma 1 Given two itemsets \( X \) and \( Y \), if \( X \subseteq Y \) and \( s(X) = s(Y) \), then \( C(X) = C(Y) \).

Lemma 2 Given an itemset \( X \) and an item \( i \), then \( g(X) \subseteq g(i) \Rightarrow i \in C(X) \).

3. GC-Tree

GC-Tree is a tree like structure that maintains all the closed itemsets in the current sliding window as nodes. The idea of GC-Tree is based on the following theorem which is introduced by Claudio Lucchese et al. [4]:

**Theorem 1** For each closed itemset \( Y \neq C(\emptyset) \), there exists one and only one sequence of \( n \) \( (n \geq 1) \) items \( i_0 < i_1 < \ldots < i_{n-1} \) such that \( \{\text{gen}_0, \text{gen}_1, \ldots, \text{gen}_{n-1}\} = \{Y_0 \cup i_0, Y_1 \cup i_1, \ldots, Y_{n-1} \cup i_{n-1}\} \) where the various \( \text{gen}_i \) are order preserving generators, with \( Y_0 = C(\emptyset) \), \( Y_{j+1} = C(Y_j \cup i_j), j \in [0, n-1] \) and \( Y = Y_n \).

Here, we assume that all the items are ordered according to a total order, \( < \). And when we say a generator is order preserving, we mean:

**Definition 2** A generator \( X = Y \cup i \), where \( Y \) is a closed itemset and \( i \not\in Y \), is said to be order preserving iff \( i < (C(X) - X) \).

The concepts “preset” and “posset” are defined as:

**Definition 3** Let \( \text{gen} = Y \cup i \) be a generator of a closed itemset where \( Y \) is a closed itemset and \( i \not\in Y \). \( \text{preset}(\text{gen}) \) is defined as \( \{j < i \mid j \not\in \text{gen}\} \), \( \text{posset}(\text{gen}) \) is defined as \( \{j \in I \mid j \not\in \text{preset}(\text{gen}) \text{ and } j \not\in C(\text{gen})\} \).

And the following rule is a shortcut used to check whether a generator is order preserving:

**Lemma 3** Let \( \text{gen} = Y \cup i \) be a generator of a closed itemset where \( Y \) is a closed itemset and \( i \not\in Y \). \( \text{gen} \) is not order preserving iff \( \exists j \in \text{preset}(\text{gen}), \text{ such that } g(\text{gen}) \supseteq g(j) \).

**Theorenm 1, Definition 1-3, Lemma 1-3, Property 1-3** are cited from [4].

GC-Tree works under data stream sliding window environments. This algorithm uses a data structure also called GC-Tree (Generator and frequent Closed itemsets Tree) to store all the frequent closed itemsets in the current sliding window. Each element in the GC-Tree has the following format: \( \langle \text{gen}, \text{eitem}, \text{clo} \rangle \). GC-Tree Node may be written in the compact way in the rest of this paper: \( \langle \text{gen}, \text{clo} \rangle \).

The generator is an itemset in the same equivalence class as a specific closed itemset. By applying the function \( C \), we could obtain the closed itemset of the equivalence class from the generator.

\( Y_0 \cup i_0 \) is an abbreviation of \( Y_0 \cup \{i_0\} \), we may use similar abbreviations in the rest of the paper without declaration.

For the concise of representation, we omit the links between the parents and children, and the support of the closed itemsets.

\( \text{eitem} \) is the item in Definition 2.

\( \langle \text{gen}, \text{clo} \rangle \) is the compact way in the rest of this paper: \( \langle \text{gen}, \text{clo} \rangle \).
elements in this path compose the order preserving generator sequence \{\text{gen}_0, \text{gen}_1, \ldots, \text{gen}_{n-1}\} mentioned in Theorem 1.

GC-Tree also introduces a series of Stopping Rules, Node Generation Rules and Node Pruning Rules to avoid redundant computations, so that the tree could be maintained efficiently.

4. The Improved GC-Tree Algorithm

4.1 The Computational Complexity of GC-Tree

In this section, we will calculate the computational complexity of GC-Tree. Let us take a look at the main block of the GC-Tree algorithm in Fig. 1.

Explanations:

- \text{preset}(\text{node}) \text{ is defined as the preset of node}.\text{gen}, \text{posset}(\text{node}) \text{ is the posset of node.}\text{gen}, \text{posset}^*(\text{gen}) \text{ is defined as posset}(\text{gen}) \cup t.
- \text{T}_1 \text{ is the set of transactions holding in the sliding window before } t \text{ comes.}
- \text{moveChild} \text{ (child,old-parent,new-parent) detaches child from old-parent and add it as a child of new-parent.}
- \text{checkChild}(\text{curr.clo} \cup i, \text{curr}, t) \text{ searches the children of curr node: if any of them satisfies the condition } \text{gen} = \text{curr.clo} \cup i, \text{ the algorithm invokes add()} recursively; if there’s no such child found, the algorithm involves closureCheck() function\footnote{Similar to the function closureCheck() in DCI-Closed, devised by Claudio Lucchese et al. [4].}. The pseudo code of the procedure checkChild() is listed in Fig. 2. In line 2 of it, \exists \text{child} \in \text{parent} \text{ means that "there exists a child node of parent".}
- \text{The pseudo code of closureCheck()} \text{ is listed in Fig. 3.}

The procedure is\_dup(\text{gen}) \text{ tries to check whether the gen} \text{ is an order-preserving generator. According to Lemma 3, it could be judged by the following condition: } \exists i \in \text{preset}, g(i) \supseteq g(\text{gen}). \text{ This requires that the algorithm caches all the g(i) in the memory, and g(\text{gen}) could be calculated by } g(\text{gen}) = \bigcap_{i \in \text{preset}} g(i). \text{ After } g(\text{gen}) \text{ is obtained and verified, closureCheck()} \text{ still need to find } \forall k \in \text{posset}^*(\text{gen}), \text{subject to } g(k) \supseteq g(\text{gen}), \text{to generate the closure of gen.}

Now, we can calculate the computational complexity of the GC-Tree algorithm.

Let \( \alpha \) be the number of nodes in GC-Tree which satisfies \( \text{node}.\text{gen} \subseteq t, \beta \) be the average size of the transactions in the data stream, \( \gamma \) be the average size of g(i). So, it is provable that the computational complexity of adding a transaction to the current sliding window is \( O(\alpha \cdot \beta^2 \cdot \gamma) \). Briefly, the computational complexity could calculated in the following way:

1. Line 6 in Fig. 1 filter out the unqualified nodes, only \( \alpha \) nodes pass this filter.
2. In the worst case, the procedure checkChild() will be invoked for each qualified node in line 10-11 or in line 24-25. Because the average size of posset\_curr is \( \beta \), the procedure checkChild() will be invoked at most \( \alpha \cdot \beta \) times.
3. closureCheck() is invoked in checkChild(). In line 3 in closureCheck(), the procedure is\_dup() is invoked. In the procedure is\_dup(), g(\text{gen}) is calculated and it is compared with every \( i, i \in \text{preset}(\text{gen}) \). Hence, the computational complexity of the procedure is\_dup() is \( \beta \cdot \gamma \).

Hence, the computational complexity of add() is \( O(\alpha \cdot \beta^2 \cdot \gamma) \).

The pseudo code of the procedure delete() which is used to remove an expired transaction from the sliding window is listed in Fig. 4.

Explanations:

\begin{verbatim}
1: procedure add(t, curr)
2: if(t \in T_1)
3: \{Y \subseteq t, Y is closed : s(Y)++;\}
4: return;
5: else
6: if (curr.gen) return;
7: newclo = curr.clo \cup t;
8: if(newclo == curr.clo)
9: s(curr)++;\}
10: \forall i \in \text{posset}^*(\text{curr})
11: checkChild(i, curr, t)
12: else
13: m = \min_{i < curr.clo - newclo}
14: newnode = newclo \cup m, curr.clo >
15: s(newnode) = s(curr);
16: \forall \text{child of curr}
17: if(child.eitem \notin \text{preset}(newnode))
18: moveChild(child, curr, newnode);
19: else
20: child.gen = newclo \cup child.eitem;
21: addChild(curr, newnode);
22: curr.clo = newclo;
23: s(curr)++;\}
24: \forall i \in \text{posset}(\text{curr})
25: checkChild(i, curr, t)
\end{verbatim}

Fig. 1 add().
1: procedure delete(t, curr)
2:   if (t \notin T_2)
3:     \text{Y is closed: } s(Y) --;
4:   return;
5: else
6:     if (curr.clot) return;
7:     \text{(child of curr)}
8:     delete(t, child)
9:     if (isClosed(curr, t))
10:    return;
11: else
12:     if (curr has more than one child)
13:       newClo = \{j | j is child.clo \};
14:       \text{curr.gen = newClo - child.eitem;}
15:       \text{moveChild(mchl, mchl.children, mchl, curr);}
16:       curr.deleteChild(mchild);
17:     else if (curr has only one child)
18:       child.gen = curr.gen;
19:       \text{moveChild(child, curr, curr.parent);}
20:     else
21:       removeChild(curr.parent, curr)
22:     removeChild(curr.parent, curr)
23:   return;
24: end procedure delete()

(1) $t$ is the transaction which is leaving the sliding window. $T_2$ is the sliding window after $t$ left.

(2) If $t \notin T_2$, that is, if there is a transaction in $T_2$ which is the same as $t$, all the closed itemsets remain closed. The algorithm only needs to adjust the support of some of them. See line 2-4 in Fig. 4.

(3) If curr.clo \notin t, then all the nodes in the subtree based on curr will remain closed. See line 6 in Fig. 4.

(4) The procedure is called recursively in a depth-first manner. See line 7-8 in Fig. 4.

(5) If the curr node is still closed, then return directly, since all the children of it have already been checked. See line 9-11 in Fig. 4.

(6) Line 13-20 in Fig. 4 deals with the situation when there’re more than one child under curr. Since curr is not closed in $T_2$, the new closed itemset for it, newClo is recalculated. It could be proved that newClo is a superset of the old closed itemset of curr. For each child of curr, the generator should be recalculated, and if there exists a node, the generator of which is equals to the newClo, the child must be a duplicated one and should be removed from the tree.

(7) Line 21-26 in Fig. 4 deals with the situation when there’s only one child under curr or there’s no child under curr. In both cases, the algorithm only needs to remove curr from the tree.

(8) The procedure isClosed() is listed in Fig. 5.

Explanations:

(1) To check whether $t.id \in g(curr.clo)$ in line 2, the algorithm need to check $\forall i \in curr.clo, t.id \in g(i)$. It requires $O(k(\beta))$. ¹

(2) Line 5 checks whether there exists a node in GC-Tree which has the same support as curr, while satisfies that node.clo \supset curr. According to Lemma 2, if there exists such a node, curr.clo is not a closed itemset, else, curr.clo is still closed.

To find such a node efficiently, the algorithm keeps a hashtable in the memory. The hashtable keeps all the reference of the nodes in GC-Tree. The key of the hashtable is the support of the nodes. If there are too many conflicts in the hashtable, the algorithm could be further improved to have a two level hashtable, the key of the first level is the support of the nodes, the key of the second level is the summary of the transaction ids of the nodes. In this way, the node with the same support (and the same summary of the transaction ids) could be found in $O(1)$. To check whether node.clo \supset curr.clo requires $O(k(\beta))$. Hence, the final computational complexity is $O(k(\beta))$.

So, the computational complexity of isClose() is $O(k(\beta))$. Now, we could carry out the computational complexity analysis of the procedure delete(). In line 6 in Fig. 4, there’re only $\alpha$ nodes pass the filtering. The procedure isClose() requires $O(k(\beta))$. Line 13 in Fig. 4 requires $O(s \cdot k(\beta))$. Hence the computational complexity of the procedure, delete(), is $O(\alpha \cdot s \cdot k(\beta))$. Note the fact that $s \ll \beta$ and $k(\beta) \ll \beta$. Hence we could conclude that the procedure delete() is much faster than add(). So the improvement algorithm is focused on the add() procedure and leave the delete() procedure the same as it is.

4.2 The Improvement

In this section, we represent the new method to improve the GC-Tree algorithm and prove the correctness of this method.

**Theorem 2** Let $S$ be the set of all the closed itemsets, then $\forall C \in S$, it must satisfy one of the 2 conditions: (1) $C$ is a transaction in the data stream, (2) $\exists C_1, C_2 \in \{S - \{C\}, C = C_1 \cap C_2$.

**Proof:** We use mathematical induction to prove this theorem. Let $D$ be the set of transactions received from data stream already. $D_n$ denotes the known data set with the first ¹Let $k(\beta)$ be the average size of the closed itemsets. For detailed information, refer to Sect. 4.4.

²Let $s$ be average number of children of each node.

³In one of our performance experiments, $s \approx 3.3$ while $\beta \approx 28.4$. For detailed information, refer to Sect. 4.4.
m transactions, and $S_m$ denotes the set of all the closed itemsets in $D_m$.

**The basic step:**

For $D_1$, there’s only one transaction in the data set. So the only closed itemset is the transaction itself. It meets the condition (1).

**The inductive step:**

Let theorem 2 holds for $D_n$, that is, $\forall C \in S_n$, it satisfies condition (1) or (2) in theorem 2. $t_{n+1}$ denotes the $n+1$th transaction, $tid_{n+1}$ is the id of it, $g_n()$ be the function $g()$ works on itemsets in $D_n$. $g_{n+1}()$ be the function $g()$ works on itemsets in $D_{n+1}$.

Then, let us take a look at $D_{n+1}$ and $S_{n+1}$.

Assumption: If $\exists C \in S_{n+1}$ which satisfies neither condition (1) nor (2). Then, we have $tid_{n+1} \in g_{n+1}(C)^1$

and,

$$C \notin S_n \Rightarrow f_n(g_n(C)) \neq C$$

Hence,

(1) If $g_n(C) = \emptyset$, then,

$$f(g_{n+1}(C)) = f(g_n(C) \cup tid_{n+1})$$

$$= f(\emptyset \cup tid_{n+1})$$

$$= tid_{n+1}$$

In this case, $C$ satisfies condition (1) of theorem 2.

(2) If $g_n(C) \neq \emptyset$, then, $\exists C_x \in S_n, C_x = f(g_n(C)) \text{ which satisfies } C_x \supset C$. Hence,

$$f(g_{n+1}(C)) = f(g_n(C) \cup tid_{n+1})$$

$$= f(g_n(C)) \cap t_{n+1}$$

$$= C_x \cap t_{n+1}$$

In this case, $C$ satisfies condition (2) of theorem 2.

Hence, $C$ satisfies either condition (1) or (2) in theorem. This is a contradiction of the assumption.

**Lemma 4** Let $\delta S_{n+1} = \{ C | C \in S_{n+1}, C \notin S_n \}$, then we have, $\forall C \in \delta S_{n+1}, C = t_{n+1}$ or $\exists C_x \in S_n, C = t_{n+1} \cap C_x$

**Proof:** It could be directly inferred from the proof of Theorem 2.

Here comes the idea based on Theorem 2 and Lemma 4: whenever a new transaction $t_{n+1}$ comes, we simply calculate the intersection between $t_{n+1}$ and every node in GC-Tree. If the intersection already exists in the GC-Tree, increase the support. Else, a new closed itemset is found, and the new algorithm inserts it to the tree. Finally, the $t_{n+1}$ is inserted to the GC-Tree as a new closed itemset if there’s no node in the tree equals to it. In this way, we could reduce the computational complexity of the algorithm.

The pseudo code in Fig. 6 is based on this idea, which is adapted from GC-Tree [15].

Figure 6 represents the improved procedure add(), the changes between the new procedure and the original one is marked with underlines. Note that, if $t \notin T_1$, then after add() is invoked, $t$ should be added to the tree as a new closed itemset. It’s because in the original version of add(), $t$ would be found as a closed itemset by closureCheck(), while in the improved version of add(), closureCheck() is replaced by closureCheckForSubTree(), which can’t find $t$ as a closed itemset.

The new procedure processChildren is described in Fig. 7.

Figure 7 represents the new procedure processChildren. For all the children of parent, if the child.gen equals to the new generator parent.clo $\cup i$, then add() is invoked, $t$ should be added to the tree as a new closed itemset. It’s because in the original version of add(), it would be found as a closed itemset by closureCheck(), while in the improved version of add(), closureCheck() is replaced by closureCheckForSubTree(), which can’t find $t$ as a closed itemset.

The improved algorithm makes sure that $\forall C \in$ GC-Tree, the intersection $C \cap t$ is performed, and the result is either an exist closed itemset or a new added closed itemset. In the Sect. 4.4, we prove the new algorithm has much better performance than the original GC-Tree. In the following section, we gives a running example which could demonstrate how GC-Tree is build.

---

1: procedure add(t, curr)
2: if ($t \in T_1$)
3: $Y \subseteq t$, Yis closed: $s(Y) +$;
4: return;
5: else
6: if ($curr.gen \not\subseteq t$)
7: closureCheckForSubTree(curr, t);
8: return;
9: newclo = curr.clo $\cap t$
10: if (newclo == curr.clo)
11: $s(curr) +$;
12: processChildren(curr, t);
13: else
14: $m = \min_{\{curr.clo - newclo\}}$
15: newnode = newclo $\cup curr.clo >$
16: $s(newnode) = s(curr)$;
17: $\forall (child of curr)$
18: if (child.eitem $\not\subseteq$ preset(newnode))
19: moveChild(child, curr, newnode);
20: else
21: child.gen = newclo $\cup$ child.eitem;
22: addChild(curr, newnode);
23: curr.clo = newclo;
24: $s(curr) +$;
25: processChildren(curr, t);

Fig. 6 Improved-add().

1: procedure processChildren(parent, t)
2: $\forall i \in posset(parent)$
3: if ($\exists child \in parent, child.gen == parent.clo \cup i$)
4: add(t, child);
5: $\forall$ other child
6: closureCheckForSubTree(child, t);

Fig. 7 processChildren().
1: procedure closureCheckForSubTree(subRoot, t)
2:  ∀ node ∈ subTree(subRoot)
3:    tempClo = node.clo ∩ t;
4:  if (∃ node ∈ GC-Tree, node.clo == tempClo)
5:    newNode = createNode(tempClo);
6:    relocate(newNode);
7:  else
8:    if (s(tempClo) not updated);
9:      s(tempClo)++;

Fig. 8 Closure Check For Sub Tree().

Table 1 Running example.

<table>
<thead>
<tr>
<th>T.id</th>
<th>Set of items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>f, c, a, m, p</td>
</tr>
<tr>
<td>2</td>
<td>f, c, a, m, p</td>
</tr>
<tr>
<td>3</td>
<td>f, c, a, b, m</td>
</tr>
<tr>
<td>4</td>
<td>f, b</td>
</tr>
<tr>
<td>5</td>
<td>c, b, p</td>
</tr>
</tbody>
</table>

Fig. 9 GC-Tree with transactions 1, 2, 3.

4.3 Running Example

Table 1 gives a running example to demonstrate how the GC-Tree is build. We assume that the total order in the example is $f < c < a < b < m < p$ and the size of the sliding window is 3.

The demonstration begins at the time point when there are 3 transactions (1, 2, 3) in the sliding window, and the corresponding GC-Tree is shown in Fig. 9.

In each node, the generator, the closed itemset and the support is shown, and the extend item in the generator is marked with underlines.

Then transaction 4, denoted as $t_4$, comes, and the procedure $add(t, root)$ is invoked. Let $T_1$ denote the sliding window before $t_4$ comes. Since $t_4 ∉ T_1$ and $curr.gen = 0 ⊆ t_4$, the algorithm goes to line 9 in Fig. 6, where the new closed itemset is calculated: $newclo = curr.clo ∩ t_4 = fcam ∩ fb = f$. The new closed itemset is not equal to $curr.clo$, hence, the algorithm goes to line 14 to create a new node, $newnode = <fc,fcam>$. Because there’s no child satisfies that $child.item ∈ preset(newnode)$, all the children of current node are moved to under the $newnode$.

After that, the $newnode$ is added to $curr$ as a child, and the closed itemset of $curr$ is set as $f$. Now the GC-Tree would be like the one shown in Fig. 10.

Then, the procedure $processChild(root, t_4)$ is invoked in line 25 to process all the children of $root$. In this example, the new node, $<fc,fcam>$, is checked, and it satisfies that $\exists i ∈ posset(parent), newnode.gen = parent.clo ∪ i$. Hence $add(t_4, newnode)$ is invoked recursively. Since $newnode.gen ⊈ t$, the algorithms goes to line 7 in Fig. 6, and the procedure $closureCheckForSubTree(newnode, t_4)$ is invoked. In this procedure, two intersection operations are performed. One of them, $fcabm ∩ fb = fb$, finds a new closed itemset $fb$, and relocate it in the GC-Tree, the other one, $fcamp ∩ fb = f$, finds an existing closed itemset, $f$. Since the support of $f$ has already been updated, the algorithm simply returns. The final result of GC-Tree is shown in Fig. 11.

Since the size of sliding window is 3, now the transaction 1, denoted as $t_1$, should leave the sliding window. Let $T_2$ denote the sliding window after $t_1$ leaved. Because there is a transaction $t_2 ∈ T_2$, which satisfies $t_2 ⊈ t_1$, the algorithm goes to line 3 in Fig. 4. Hence, for each node in the GC-Tree, which satisfies that $node.clo ⊈ t_1$, the support is minused by 1. The final result of GC-Tree is shown in Fig. 12.
4.4 The Computational Complexity

Let \( n \) be the number of the nodes in the GC-Tree, \( p \) be the number of new added closed itemsets, \( h \) be the hight of the GC-Tree, \( s \) be the average number of children of each node, \( k(\beta) \) be the average size of the closed itemsets. Note that \( k(\beta) \) depends on \( \beta \) and \( k(\beta) < \beta \).

Firstly, let us consider the process of the nodes already exist: For all nodes in GC-Tree, the new algorithm will perform the intersection operation once and only once. So, the computational complex is \( O(n \cdot k(\beta)) \).

Secondly, let us consider the process of the new added closed itemsets. Let \( \text{clo} \) denote a new added itemset, the algorithm will perform two kinds of operations for it,

1. \( \text{createNode}(\text{clo}) \), which in turn need to set the support of the \( \text{clo} \). The support of \( \text{clo} \) will be calculated as, \( s(\text{clo}) = \bigcap_{i \in \text{clo}} g(i) \). This is \( O(\beta \cdot \gamma) \).
2. \( \text{relocate}(\text{newNode}) \). The procedure \( \text{relocate}() \) works in the following way. Initially, it set the root node of GC-Tree as the current node, \( \text{curr} \). Let \( i_m = \min\{\text{newNode.clo} \setminus \text{curr.clo}\} \), there could be only one child of \( \text{curr} \) satisfies that \( \text{child.eitem} = i_m \). If there exists such a child, then set \( \text{curr} = \text{child} \) and recheck the \( \text{curr} \) node. It there is no such child found, then, \( \text{curr} \) is the parent node of \( \text{newNode} \).

For example, in the running example in Sect. 4.3, the algorithm calls \( \text{relocate}() \) to add the node \( <fb, fb, 3> \) to the tree in Fig. 11. Initially, \( \text{curr} = \text{root} \), and the algorithm tries to find a child of \( \text{curr} \) which satisfies that \( \text{child.eitem} = i_m \). There is no such a child found, so \( \text{root} \) is the parent node of \( <fb, fb, 3> \).

The computational complexity of the procedure \( \text{relocate}() \) is composed of two parts. Firstly, in each step in the iteration, each child of the \( \text{curr} \) node is checked whether \( \text{child.eitem} = i_m \). The average number of children of each node is \( s \). Secondly, the number of steps of the iteration is at most the hight of the tree, \( h \). Hence, the final computational complexity of \( \text{relocate}() \) is \( O(h \cdot s) \).

So, finally, the computational complexity of the improved algorithm is \( O(n \cdot k(\beta) + p \cdot \beta \cdot \gamma + p \cdot h \cdot s) \). Hence, the computational complicity is a linear combination of \( \beta \) and \( k(\beta) \), that is, the computational complexity scales linearly according to the average transaction size and the average itemset size.

Note that the first tow parts of it are dominant. For example, Table 3 lists the size of GC-Tree in one of the performance experiments. If the number of movies is 2600, the size of the GC-Tree would be 26709, that is \( n=26709 \). And the statistical analysis shows that \( k(\beta) \approx 6.8, h = 68, s \approx 3.3, \beta \approx 28.4, \gamma \approx 17.3, p \approx 118.9 \). Hence, we could conclude that \( n \cdot k(\beta) \gg p \cdot h \cdot s, \) and \( p \cdot \beta \cdot \gamma \gg p \cdot h \cdot s \).

5. Some Optimizations

In this section, we provide two methods to optimize the algorithm in Sect. 4. Each method tries to optimize one item in the computational complexity of the improved algorithm, \( O(n \cdot k(\beta) + p \cdot \beta \cdot \gamma + p \cdot h \cdot s) \).

- Method 1: Tries to optimize the item \( O(n \cdot k(\beta)) \).
- Method 2: Tries to optimize the item \( O(p \cdot \beta \cdot \gamma + p \cdot h \cdot s) \).

5.1 Optimization Method 1

**Lemma 5** Let \( A, B, C \) be 3 sets, if \( A \subseteq C \), and \( C \cap A \subseteq B \), then we have, \( C \cap A = C \cap B \).

**Proof:**

1. For \( \supseteq \)
   \[ A \supseteq B \Rightarrow C \cap A \supseteq C \cap B \]
2. For \( \subseteq \)
   \[ C \cap A \subseteq B \Rightarrow C \cap (C \cap A) \subseteq C \cap B \Rightarrow C \cap A \subseteq C \cap B \]

So, we have \( C \cap A = C \cap B \).

The following lemma is cited from [15]

**Lemma 6** Let \( X \) and \( Y \) be 2 closed itemsets, and \( Z = X \cap Y \neq \emptyset \), then \( Z \) is closed itemset too.

**Theorem 3** If \( \exists C_x \in S_n, C_x \cap t_{n+1} = I \) and \( I \in S_n \), then \( \forall \{C \in S_n : C \subseteq C_x\} \), we have, if \( C \cap t_{n+1} \neq \emptyset \), then \( C \in S_n \).\]

**Proof:** Since \( C \subseteq C_x \), then \( C \cap t_{n+1} \subseteq C_x \cap t_{n+1} = I \), meanwhile, \( I \subseteq t_{n+1} \). According to Lemma 5, we have \( C \cap t_{n+1} = C \cap I \). Because \( C \in S_n \), and \( I \in S_n \), hence according to Lemma 6, we have \( C \cap I = C \cap t_{n+1} \in S_n \).

Here comes the idea based on Theorem 3: If we find a node \( C_x \in S_n \), which has \( I = C_x \cap t_{n+1} \neq \emptyset \) and \( I \in S_n \), then \( \forall \{C \in S_n : C \subseteq C_x\} \), \( C \) will not generate new closed itemsets for \( t_{n+1} \). If this kind of nodes are termed Invalid Nodes, then, we could let the algorithm maintain a hash table, named \( \text{skipSet} \), which contains all the invalid nodes found so far. When traverse the subtree in the procedure \( \text{closureCheckForSubTree}() \), the algorithm should skip the nodes hold by \( \text{skipSet} \). The pseudo codes based on this idea is represented in Fig. 13.

**Explanations:**

- In this procedure, the algorithm traverses the sub-tree with the node "subRoot" as the root. If the current node is contained by the hash table \( \text{skipSet} \), the algorithm returns directly. If the current node is not contained in \( \text{skipSet} \) and the intersection set \( \text{tempClo} \) is
1: procedure closureCheckForSubTree(subRoot, t)
2:   iterator = subTree(subRoot).postOrderIterator();
3:   while(iterator.hasNext())
4:     node = iterator.next();
5:     if(!skipSet.contains(node))
6:       tempClo = node.clo ∩ t;
7:     if(nodeGC-Tree, node.clo = tempClo)
8:       newNode = createNode(tempClo);
9:       relocate(newNode);
10:  else
11:    ancestor of node
12:    skipSet.add(ancestor);

Fig. 13 ClosureCheckForSubTree().

not contained by the GC-Tree, then it does the same thing as described in Sect. 4.2. Else, the algorithm finds a "invalid node" according to theorem 3, so it puts the current "invalid node" and all its ancestors to the skipSet (Because the closed itemsets of the ancestors are subsets of the itemset of the current "invalid node").

- The code "subTree(subRoot).postOrderIterator()" is used to obtain an iterator which traverses the subtree in postorder. The algorithm uses the postorder traversal because in this way, the nodes will be iterated from bottom to top. So, if the algorithm finds a node n, which satisfies n.clo ∩ t_{n+1} ≠ ∅ and t ∈ S, then all the ancestors of the node n are "invalid nodes" too and could be skipped in the following traversal.

- Let S = \{subNode ∈ GC-Tree : subNode.clo ⊂ node.clo\}. Then in line 11, we could have put the following code: "for all n, n ∈ S" (Because all the n ∈ S are invalid nodes, and the ancestors of node is only a subset of S). However, to find every element in S itself requires O(n) which the algorithm can’t afford, so it just puts the ancestors of node to the skipSet because the ancestor nodes could be found in O(1).

Let n₀ be the number of nodes in the GC-Tree which is not contained by skipSet, then the computational complexity of the algorithm could be reduced to O(n₀ · k(β) + p · β · γ + p · h · s)).

5.2 Optimization Method 2

In this sub-section, we propose a method to cut down the second item of the computational complexity, O(p · β · γ + p · h · s). The first part of the item, O(p · β · γ), is caused by the calculations of the supports of the new added closed itemsets, which can’t be improved because we believe there’s no simpler way to calculate the supports.

The second part of this item, O(p · h · s), is caused by the relocations of the new nodes. This part could be eliminated if the algorithm does not do the relocation operation at all. If the algorithm does not do the relocation operation, then the tree structure is broken, hence, the algorithm can’t use the procedure add() in table 3 to find new closed itemsets. Instead, we put all the closed itemsets in a list, and use the following algorithm in Fig. 14 to find all the new closed itemsets.

Explanations:
- list is the list which contains all the closed itemsets.
- Whenever a new transaction t comes, invoke closureCheck(list, t) to handle it.
- There’s a hash table keeps all the closed itemsets in the list, so the check of cᵢ ∉ list could be done in O(1).
- The optimization method described in Sect. 5.1 could not be applied here because there’s no "ancestors" any longer, so no "invalid nodes" could be found.
- Since the new algorithm does not use GC-Tree to store the closed itemsets, the original delete() procedure could not work any longer. The new version of delete() checks every node in the list, if node is not closed, delete it. That is, the procedure isClosed() in Fig. 5 will be invoked n times. Hence, the computational complexity of the new version of delete() is O(n · k(β)), which is much less than the computational complexity of the procedure add().

So, the computational complexity of the algorithm could be reduced to O(n · k(β) + p · β · γ).

6. Performance Evaluation

In this section, we will empirically analyze the scaling behavior of the four algorithms in the following experiments according to the average transaction size β and the average sliding window size t respectively.

We use two kinds of datasets in our experiments, the synthetic datasets and the real world datasets:
- A series of synthetic datasets are used. Each dataset is generated by the same method as described in [14]. An example synthetic dataset is T10.I6.D100K, where the three numbers denote the average transaction size (T), the average maximal potential frequent itemset size (I) and the total number of transactions (D), respectively.
- The real world training dataset comes from KDD-Cup 2007. The dataset provides a list of 100,000...
user\textsubscript{id}, movie\textsubscript{id} pairs where the users and movies are drawn from the Netflix Prize training data set which consists of more than 100 million ratings from over 480 thousand randomly-chosen, anonymous customers on nearly 18 thousand movie titles. In this dataset, a transaction corresponds to a user, an item corresponds to a movie.

In order to analyze the scaling behavior of the algorithms according to the average transaction size $\beta$, 7 datasets are sampled from the KDD-Cup dataset. They are generated by randomly choosing 800, 1100, 1400, 1700, 2000, 2300, 2600 movies from the whole dataset respectively.

### 6.1 Experiments on Synthetic Dataset

Figure 15 shows the average processing time of the original GC-Tree algorithm for every transaction. In this figure, the x-axis represents the Average Size of the Transactions, the y-axis represents the Sliding Window Size, the z-axis represents the Running Time required to process each transaction. According to the analysis of the original GC-Tree algorithm, the computational complexity is $O(\alpha \cdot \beta^2 \cdot \gamma)$. It could be validated intuitively by the result of the experiment, since it could be inferred from the picture that the cpu time increases quadratically according to the average size of the transactions($\beta$) and almost linearly according to the sliding window size(which determines the value of $\gamma$).

Figure 16 shows the average processing time of the improved GC-Tree algorithm for every transaction. According to the analysis of the improved GC-Tree algorithm, the computational complexity is $O(n \cdot k(\beta) + p \cdot \beta \cdot \gamma + p \cdot h \cdot s)$). It could be validated intuitively by the result of the experiment too, since for the improved algorithm, the cpu time scales both linearly according to the average transaction size($\beta$) and the the sliding window size(which determines the size of $\gamma$).

From Fig. 15 and Fig. 16, We could conclude that when the average transaction size and(or) the sliding window size is big, the new algorithm is several orders of magnitude faster than the original one.

Figure 17 shows the average processing time of the improved GC-Tree algorithm which is optimized with Method 1. It could be seen that the optimization method further improves the performance.

Figure 18 shows the average processing time of the improved GC-Tree algorithm which is optimized with Method 2. It could be seen that this optimization method also further improves the performance.

Figure 19 shows the average processing time of all the algorithms by fixing the size of the sliding window(200).

Figure 20 shows the average processing time of all the algorithms by fixing the average size of the transactions(20).

By putting all the algorithms together, Fig. 19 and Fig. 20 give us more clear pictures about the scaling behavior of each algorithm. In Fig. 19, it could be demonstrated visually that the running time of the Original GC-Tree algorithm for every transaction increases quadratically according to the average transaction size. Meanwhile, the improved algorithms scales almost linearly. In Fig. 20, it could be inferred that all the algorithms scale linearly according to the sliding window size. However, the slope of the original GC-Tree Algorithm is much more greater than the rest algorithms, this observation consists with the computational complexity analysis, which tells that the compu-
According to Fig. 19 and Fig. 20, it seems that the optimization method 2 is better than the optimization method 1, however, we believe it is not necessary true since according to Sect. 5.1, the implementation of the optimization method 1 only filters out a small subset of the "invalid nodes". This fact opens several areas for research, one of them is to figure a way to filter out all the "invalid nodes", another is trying to make optimization method 1 and optimization method 2 compatible. Each way could make the algorithm more faster.

6.2 Experiments on KDD-CUP 2007 Dataset

Figure 21 shows the average processing time of the original GC-Tree algorithm for every transaction on the real world datasets.

Figure 22 shows the average processing time of the improved GC-Tree algorithm for every transaction on the real world datasets.

Figure 23 shows the average processing time of the improved GC-Tree algorithm which is optimized with Method 1 on the real world datasets.

Figure 24 shows the average processing time of the improved GC-Tree algorithm which is optimized with Method 2 on the real world datasets.
Fig. 23  Runtime performance of the improved GC-Tree optimized with Method 1.

Fig. 24  Runtime performance of the improved GC-Tree optimized with Method 2.

Fig. 25  Runtime performance of all the algorithms based on the number of movies (The size of the sliding window is fixed as 230).

Figure 25 shows the average processing time of all the algorithms based on the number of movies (The size of the sliding window is fixed as 230).

The results of the experiments based on the real world datasets are almost the same as those based on the synthetical datasets. So similar conclusions could be drawn.

Nevertheless, there are still some small differences. For example, it could be inferred that compared to the improvement of the performance made by the new algorithms based on the synthetical datasets, the performance improvement could be even better based on the real world data sets. It is simply because the performance of the original GC-Tree algorithm is getting worse. The reason of it is the real world data sets have more noises. For example, the sizes of the transactions vary considerably between each other, that means while the average transaction size remains the same, there’re more long transactions in the real world datasets than the synthetical datasets. Hence, the performance of the original GC-Tree algorithm is getting worse since it depends on the long transactions more than on the short transactions.

6.3 Runtime Comparison with the Batch Algorithm

In order to show how much the incremental algorithm proposed in this paper outperforms the traditional batch algorithms in the sliding window environment, a performance study is carried out in this section.

FP-Close [17] is a state of arts batch algorithm which dedicated to find the frequent closed itemsets. The implementation of FP-Close is available at http://fimi.cs.helsinki.fi/fimi04/. Whenever a new transaction comes to the sliding window, the batch algorithm is required to be executed to calculate all the closed itemsets in the sliding window.

Figure 27 shows the average processing time of all the algorithms for every transaction on the synthetic datasets by

\[ \text{Average processing time} = \frac{1}{n} \sum_{i=1}^{n} C \beta_i \gamma, \]

where \( C \) is a constant and \( \beta_i \) is the length of the \( i \)-th transaction. It is obvious that this expression is dominated by the transactions with bigger \( \beta_i \).
6.4 The Size of GC-Tree

In Table 2, we list the sizes of the GC-Trees on the synthetic datasets. The first row lists the average transaction sizes of the datasets. The second row lists the corresponding sizes of the GC-Trees.

In Table 3, we list the sizes of the GC-Trees on the real datasets. The first row lists the numbers of movies in the datasets. The second row lists the corresponding sizes of the GC-Trees.

7. Conclusions

In this paper we presented an improved algorithm working under sliding window environment which scales linearly according to the average transaction size and the average closed itemset size. This algorithm is a variant of the state-of-arts algorithm, GC-Tree [15]. The improvement is based on the core theorem in Sect. 4.2. In addition, we proposed two optimization methods which could improve the performance even better. The performance evaluation shows that this new algorithm is empirically very fast.

This algorithm also opens several areas for research:

- Figuring out a way to maintain the “includes” relationship between closed itemsets in the tree. It has two kinds of benefits, one benefit is that the optimization method described in Sect. 5.1 could filter out all the “invalid nodes” instead of only a small subset of them. The other benefit is that it could let the two optimization methods work together to further improve the performance.
- The improvement methods proposed in this paper could be adapted to work with TGC-Tree [16], which tries to trace the frequent closed itemsets and frequent transaction sets simultaneously and incrementally.

References


