An Efficient Algorithm for Mining Maximal Frequent Sequential Patterns in Large Databases

Qiu-bin SU\textsuperscript{1}, Lu LU\textsuperscript{1,2,*} and Bin CHENG\textsuperscript{1}

\textsuperscript{1}School of Computer Science & Engineering, South China University of Technology, Guangzhou, Guangdong, China
\textsuperscript{2}Modern Industrial Technology Research Institute, South China University of Technology, Zhongshan, China

*Corresponding author

Keywords: Sequential pattern mining, Frequent sequence mining, Pruning strategy; data stream.

Abstract. Frequent sequence mining is one of the important research directions of click stream analysis, this paper studies the problem of mining maximal frequent sequences in mobile app clickstreams. Different from frequent itemsets mining, frequent sequence mining takes the time order of elements into account. In this paper, MFSGrowth (Maximal Frequent Sequence Growth) is proposed for fast discovery of frequent sequence. MFSGrowth is an efficient algorithm based on the FP Tree, combined with the storage characteristics of the TriedTree, experiments show that the algorithm performs well in both mining time and storage efficiency.

Introduction

Frequent sequence mining is to find out the user's general usage process and page browsing interest by mining the user's page access sequence in the process of using App. Because of the common feature of frequent sequence, it is helpful to improve the performance of mobile app and improve the user experience by improving the loading performance and adjusting the structure of the pages.

Click stream\textsuperscript{[1]} refers to the user access to the entire process of App, which is accessed to the page called Page, click stream is formed by a series of ordered Page to form Session. Click stream analysis first through access to information collected for each Page in the Session, such as the name of the page, enter time and departure time, loading time and residence time, and Session's overall information, such as page browsing order, browse page number and access time data, then use the machine learning algorithm for mining patterns contained in a large number of Session.

There are many algorithms for mining frequent sequence (FS), such as GSP\textsuperscript{[2]}, SPAM\textsuperscript{[3]}, PrefixSpan\textsuperscript{[4]} and SPADE\textsuperscript{[5]} etc. But these algorithms mostly performed on the whole data set, and usually have the problems of high time complexity and high spatial complexity; besides, SPAM and SPADE are only suitable for small data sets, because their memory consumption will increase with the size of the data set; the GSP algorithm scans the database by the length of the longest frequent sequence, therefore, when the length of the longest frequent sequence is very large, the I/O consumption of scanning the database will be very large. In this case, we propose MFSGrowth, a new algorithm for mining FS. Considering any sub sequence is also FS, in order to avoid repetitive mining of sequences, if a FS is not a subsequence of any other FS in a frequent sequence set, then the frequent sequence is called a maximal frequent sequence (MFS). In this paper, only MFS is considered, since the most information of FS is contained in MFS.

The rest of the paper is organized as follows: in the next section, a formal description of the problem are given, including some definitions, then describe MFSGrowth in detail. After that, performance evaluation and experimental results are given. The final section contains conclusion and future work.
Problem Description

In order to simplify the introduction of MFSGrowth, some basic concepts and notations involved in the algorithm will be given in this section.

Let \( I = \{P_1, P_2, \ldots, P_n\} \) be a set of distinct items, i.e. all the pages in an app, where \( P_i \) corresponds to a specific page in the app, and \( n \) stands for the length of the itemset \( I \).

Definition 1. Session is a set of continuous page access sequences ordered by timestamp in a usage data record, or generated by a user from the time when he enters the page to the time he leaves, represented as \( \text{Session} = \{S_1, S_2, \ldots, S_m\} \), where \( S_i = <(P_{i_1}, t_{i_1}), (P_{i_2}, t_{i_2}), \ldots, (P_{i_k}, t_{i_k})> \), \( P_{i_j} \in I \) and \( t_{i_j} \) is the loading time of page \( P_{i_j} \). A session also has an associated unique identifier called \( \text{Sid} \).

Definition 2. Dwell Time refers to the time that a user spends on a specific page in a sequence of session. The dwell time of \( P_{i_j} \) can be calculated by subtracting the loading time of the current page from the loading time of the next page. We can use \( T_0 \) to represent the dwell time of \( P_{i_j} \). Meanwhile, two thresholds must be given to constrain \( T_0 \) of every Page, the two thresholds are minimal dwell time \( \lambda_1 \) and maximal dwell time \( \lambda_2 \).

Definition 3. Support is defined as the number of a specific sequence occurred in DB divided by the total number of user access sequence records, that is \( \text{sup}(\text{Seq}_i) = \frac{\text{Session.count}_i}{\text{Session.count}} \).

Definition 4. Frequent Sequence is called when the support of a sequence \( \text{Seq}_i \) is no less than \( \text{Minsup}(\text{sup}(\text{Seq}_i) \geq \text{Minsup}) \) and contains at least two pages, where \( \text{Minsup} \) is a user specified support threshold.

Most of the original data are incomplete, inconsistent or have outliers, these raw data can not be directly used for data analysis, which will lead to lack of accuracy of the analysis results. Therefore, it is necessary to preprocess the original data before mining.

Different from the data in the Web log, the data collected from daily use of mobile app is simple. Only page name, page load time, and page access sequence was collected. Incomplete and inconsistent data is not an issue, so the data preprocessing phase only need to deal with the data outliers, as well as simple data conversion.

We use the following rules to deal with dwell time and Session.count:

1. If the dwell time of a page is less than \( \lambda_1 \), then indicates that the page is not a user interest page or the page loading process error, the page need to be removed from sequence \( S_i \).
2. If the dwell time of a page is more than \( \lambda_2 \), then the page need to be removed from sequence \( S_i \) too. The case implies the user may leave the App or the page load process error.
3. If the total number of pages in the user session (the page can be repeated) exceeds \( N(\text{Session.count} > N) \), it means that the session may be generated by the user without any purpose, and the session does not have a reference value, \( N \) is 30min.

The Proposed Algorithm: MFSGrowth

Constructing the Tree Structures

First let’s introduce two novel tree storage structures: MFSTree(Maximal Frequent Sequence Tree) and InvTree(Inverse Tree), both are based on FP-Tree and draws on the storage features of TriedTree. The use of these two data structures for data compression can help reduce memory consumption.

MFSTree and InvTree have the same tree node (TreeNode) structure and a header table, each TreeNode includes five fields: \( \text{Node.name} \), \( \text{Node.count} \), \( \text{Node.parent} \), \( \text{Node.children} \) and \( \text{Node.next} \). Field \( \text{parent} \) holds a pointer to the parent node, \( \text{null} \) for root. Field \( \text{children} \) are used to save all next Node occurred in the sessions. Field \( \text{next} \) denotes the pointer pointing to a Node carrying the same name with the Page.

Algorithm 1 explains the steps of constructing a MFSTree.
We assume part of the user traversal sequence DB data is shown in Table 1. Among them, the first column represents a unique identification number of the session, the second column represents a conversation page access sequence, the last page of each session are the ending page, in this case, all set to null, the third column correspond to each page loading time occurred in the second columns.

First we need to preprocess the data, the dwell time for any session can’t exceed or less than two defined threshold (decision-maker is free to determine the values of $\lambda_1$ and $\lambda_2$ according to his purpose). In this case let’s set $\lambda_1=0.5\text{ min}$ and $\lambda_2=30\text{ min}$. For example, though page P10 has appeared in session 101, its dwell time is $53.5-10.5=43\text{ min}$, exceeds $\lambda_2$, and the dwell time of page P1 in session 104 is $7.2-7=0.2\text{ min}$, less than $\lambda_1$, these data need to be preprocessed before use.

After data preprocessing and convert the load time of the data into dwell time, Table 2 shows the data after first scan of the user traversal sequence database and count the number of accesses to each page, now use this as an example to explain Algorithm 1. As is shown in Figure 1 and Figure 2, Each node in these figures shows the name of the page (the character before the colon) and the access frequency of the page (the number after the colon), corresponding to the `name` and `count` attributes in TreeNode. In the second DB pass, when adding a session to MFSTree, if the current MFSTree has a branch of the same prefix, use that branch as a public prefix and increase the count of pages in the public prefix. If the current MFSTree do not have the same prefix, then add a new one, and the count of each page node in that branch is counted as 1. The dotted lines in the figure points to the node with the same name, corresponding to the next attribute in the TreeNode, while the starting point are all from the header table.
Algorithm 2: MAKE-InvTree()

Input: MFSTree, Min_sup

1: for each Pi in the header table do
2:     Q = Pi.next;
3:     create new InvTree βi for Q
4:     while (Q != null && Q.next != null) do
5:         while (Q.parent != null) do
6:             if βi has a node N and
7:                 N.name = Q.parent.name Then
8:                 N.count = N.count + 1
9:         else
10:             Insert(βi, N) // insert N to βi
11:         end if
12:     end while
13:     Q = Q.next // find Q’s next homonym
14: end for

Table 2. The user session database after data preprocessing.

<table>
<thead>
<tr>
<th>Sid</th>
<th>User Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>⟨(P₁,3), (P₁,2), (P₂,2.5), (P₃,3.5), (P₇,7), (P₈,8)⟩</td>
</tr>
<tr>
<td>101</td>
<td>⟨(P₅,5), (P₄,2), (P₃,3.5), (P₄,4.5), (P₅,5), (P₆,8)⟩</td>
</tr>
<tr>
<td>102</td>
<td>⟨(P₂,2), (P₁,1), (P₃,3), (P₄,4), (P₅,5), (P₆,6)⟩</td>
</tr>
<tr>
<td>103</td>
<td>⟨(P₇,7), (P₅,5), (P₄,3), (P₄,4)⟩</td>
</tr>
<tr>
<td>104</td>
<td>⟨(P₁,3), (P₁,2), (P₂,2.5), (P₃,3.5), (P₇,7), (P₈,8)⟩</td>
</tr>
</tbody>
</table>

Figure 1. The MFSTree after join session 100. Figure 2. The MFSTree after join session 104.

After generating the full MFSTree, Algorithm 2 are used to create all the InvTrees corresponding to every page in the header table, and then use these InvTrees to mine frequent sequences. For example, the process of generating InvTree with Pᵢ as the root node is shown in Figure 3. First, the \( sup(Pᵢ) \) of Pᵢ must be beyond \( min_sup \), or its InvTree will be null. Then find in the MFSTree the first node with the same name through the pointer of Pᵢ in the header table, denoted by Q, use Q as the starting node and find all its ancestor nodes(Q.parent), add them to the InvTree one by one by the idea of reverse TriedTree, where the count of each node is the current Q node count. If Q.next is not null, means that the next node with the same name is not empty, then continue this step until Q.next is null. Finally, after pruning branches with support less than frequency thresholds, we get an InvTree from node Pᵢ.
Designing the Algorithm

It is necessary for the user to set the threshold of frequent degree before the mining of maximal frequent sequence, the frequent threshold is a quantitative definition of the concept of "frequent". It cannot be measured using the absolute number, because the absolute number of data sets of different sizes have different frequency, it can only be defined by percent frequency. Assuming that the size of the data set is N, when the user sets the threshold, the MFSGrowth algorithm will find the sequence of access frequency greater than N*threshold. We apply the following strategies to mine MFSGrowth from MFSTree and InvTree. At first, the user session database is scanned for the first time and a collection of page elements whose support is not less than the frequency threshold(Minsup) will be calculated to build the header table. Then, according to $\lambda_1$, $\lambda_2$, Minsup specified by user, we can use Algorithm1 to generate MFSTree, this is the second and also the last scan of the user session database. Finally, generating InvTree for every page in the header table from MFSTree by using Algorithm2, do the pruning operations, excluding non-maximal frequent sequences, leaving only MFS. By finding all the paths of the root node to all leaf nodes and reversing these paths, all the frequent sequences of the nodes are obtained. Although some of the non-maximal frequent sequences are cut in the prune process, only the end node of the same maximum frequent sequences. To completely remove the non-maximal frequent sequence, some string matching algorithms such as KMP, are needed for further processing.

Analysis and Performance Evaluation

In the process of frequent sequence mining, the number of scanning session database and the memory consumption are two key factors to determine the efficiency of mining. In this section will analyze the efficiency of MFSGrowth algorithm from these two aspects.

Because the MFSGrowth algorithm is on the basis of the classical frequent itemset mining algorithm FP-Tree to taking the order of the sequence element into account, it has the basic features of FP-Tree, the algorithm execution process only needs to access the database twice. In the MFSGrowth algorithm, the first db access is used to generate header table consisting of pages that are no less than the frequency threshold, which is mainly used for subsequent generation of InvTree. The second scan database is used to load the sessions one by one and generate MFSTree. The follow-up frequent sequence mining operations are completed based on the MFSTree and header table, do not need to access the database, so the algorithm costs less time. In terms of memory usage, since the memory of the MFSGrowth algorithm is mainly consumed in the construction of MFSTree and InvTree, the two tree structures are designed with the idea of TriedTree, it has very low memory consumption feature. And the in-degree and out-degree of each page is not large, the length of each session is limited in an app use process, therefore, when building MFSTree, we can effectively use the public prefix's role in reducing storage space. To sum up, the MFSGrowth algorithm can help improve the efficiency of mining frequent sequences to some extent.

To compare MFSGrowth with other algorithms, we implemented GSP and performed a number of experiments in the same experimental environment. Figure 4 shows the comparison results of the two algorithms under different frequency and different size of data set, each value in the figure is obtained by averaging after ten operations and excluding the outliers.
Figure 4-1 shows the execution time of the MFSGrowth and GSP at different frequency thresholds, with a data set size of 1142. Figure 4-2 shows the execution time under different experimental datasets, with the threshold of 5%. Results show that the execution time of both algorithms decreases with the increase of the frequency threshold or the decrease of the data set size. Among them, the execution time of GSP is more influenced by the frequency threshold and the size of data set, while the execution time of MFSGrowth has little effect from them. The main reason is that GSP is based on the Apriori algorithm, which requires multiple scanning of the database, MFSGrowth only needs to scan the database twice, so the time overhead is very large. On the other hand, in terms of memory consumption, the memory overhead of GSP is mainly used for k order and the set of candidate k-sequences storage, and MFSGrowth is used for MFSTree and InvTree storage.

Both algorithms will have a certain memory overhead in the procedure of implementation. Theoretically the number of different pages in App is limited and the sequence set size obtained by the combination of different pages is infinite, with the increase of the data size, the MFSTree constructed by MFSGrowth will increase the memory consumption as the data size expand, GSP will gradually remain stable. But in the actual situation, the user always want to minimize the number of pages as much as possible to achieve their purpose, and the average page in and out degree of entry is relatively small, so the idea of the TriedTree can greatly reduce the MFSTree storage overhead.

Conclusions

This paper proposed a new I/O efficient algorithm MFSGrowth that mines the set of all MFS over the mobile app access sequence, and two in-memory data structure called MFSTree and InvTree are constructed for storing frequent sequences. In the MFSGrowth algorithm, only two database passes are required, and bidirectional dwell time is used to constrain every page in session sequences, which can efficiently limits the number of the meaningless pages. Experiments showed that MFSGrowth can effectively reduce the execution time and I/O cost. Extensive experiments are also performed to compare with GSP, by using the proper constraint parameters, MFSGrowth can outperform GSP by a wide margin. The next step, we plan to extend this work by parallelizing the algorithm for a shared multiprocessor machine.

Acknowledgement

This research was supported by the National Nature Science Foundation of China (No. 61370103), Guangdong Province Science & Technology Fund (2015B010129007) and Zhongshan Produce & Research Fund (2017A1014).
References


