Mining Frequent Closed Structures in Streaming Melody Sequences

Hua-Fu Li  
Department of Computer Science  
and Information Engineering  
National Chiao-Tung University  
Hsinchu, Taiwan, R.O.C.  
hfli@csie.nctu.edu.tw

Suh-Yin Lee  
Department of Computer Science  
and Information Engineering  
National Chiao-Tung University  
Hsinchu, Taiwan, R.O.C.  
sylee@csie.nctu.edu.tw

Man-Kwan Shan  
Department of Computer Science  
National Cheng Chi University  
Taipei, Taiwan, R.O.C.  
mkshan@cs.nccu.edu.tw

Abstract

In this paper, we study the problem of mining frequent closed structures in a continuous, infinite-sized, and fast changing music melody stream. By modeling a music melody as a sequence of chord-sets, we propose an efficient algorithm FCS-stream (Frequent Closed Structures of streaming melody sequences) for incremental mining of frequent closed structures in one scan of the continuous stream of chord-set sequences. An extended prefix-tree structure called TCS-tree (Temporal Closed Structure tree) is developed for storing compact, essential information about the frequent closed structures of the stream. Results from our theoretical analysis and experimental studies with synthetic data show that algorithm FCS-stream satisfies the main performance requirements, namely, single-pass, bounded memory, and real-time, for data stream mining.

1. Introduction

In recent years, database and data mining communities have focus on a new data model, where data arrives in the form of rapid, infinite-sized, fast changing, and continuous streams. Such applications include network traffic monitoring, sensor data, financial tickers, online transaction flows in retail chains, Web record and click stream in Web applications, etc. In such streaming model, data does not take the form of persistent relations but arrives sequentially, and is processed by a single-pass algorithm whose workspace is insufficient to store all the data, so the algorithm must maintain an in-memory summary data structure for storing essential information of the streaming data. Therefore, the outputs generated by this algorithm are approximate answers [2, 7]. In this paper, we called an algorithm is stream-efficient if it satisfies the following performance requirements: one streaming data scan, limited memory requirement for constructing the in-memory summary structure, and fast processing time of each record in the stream and short response time for query processing.

Several efficient techniques have been proposed recently for discovering and analyzing the content of music data [3, 5, 6, 8, 9, 10]. However, no work has been done on mining streaming music data. Thus, in this paper, we study the problem of how to mine all temporal frequent closed structures in streaming chord-set sequences. The problem comes from the context of online music downloading services, such as Kuro (www.music.com.tw), where the streams in questions are streams of queries, i.e., music downloading requests, sent to the query processing server, and we are interested in finding the frequent closed melody structures requested by most customers in some period of time. With the computation model of music melody streams presented in Figure 1, the melody stream processor and the in-memory summary structure are two major components in such streaming environment. The user query processor receives user queries in the form of <Timestamp, Query-ID, Music-ID>, and transforms these queries into music melody sequences in the form of <Timestamp, Query-ID, Melody-Sequence> by accessing the music database. Notice that the buffer component can be optionally set for temporary storage of recent melodies sequences of the stream.

In this work, we focus on the dotted rectangle of Figure 1. While providing a general framework of streaming music mining, FCS-stream has two major features, namely, one scan of streaming melody sequences for online closed information collection, and extended prefix-tree-based compact pattern representation. With these two important features, FCS-stream is provided with the adaptability to work continuously on infinite stream for arbitrary long time with bounded resources, such as CPU time and main memory, and to quickly answer users’ queries at any time.

Figure 1. Computation model for streaming music melody mining

2. Preliminaries

2.1 Music Terminologies

In this section, we describe several features of music data used in this paper. For the basic terminologies on music, we refer to, e.g. [5]. A chord is a sounding combination of three or more notes at the same time. A note is a single symbol on
a musical score, indicating the pitch and duration of what is to be sung and played. A chord-set is a set of chord. More details about how to use the chord-sets to represent the music melody can be found in [8].

### 2.2 Problem Statement

Let \( \Psi = \{t_1, t_2, \ldots, t_i\} \) be a set of literals, called chord-sets, or items for simplicity. A melody sequence \( S \) with \( m \) chord-sets is denoted by \( S = (x_1, x_2, \ldots, x_m) \), such that \( S \subseteq \Psi \), and is called \( m \)-gram. A block \( B_i \) is consisted of a timestamp \( T_i \) and a set of melody sequences, i.e., \( B_i = (T_i, S_1, S_2, \ldots, S_k) \), where \( k \geq 0 \).

**Definition 1.** A melody stream \( MS = [B_1, B_2, \ldots, B_N, \ldots] \), is an infinite sequence of blocks, where each block \( B_i \) is associated with a block identifier \( i \), and \( N \) is the identifier of the "latest" block \( B_N \). The current length of the melody stream, i.e., \( |MS| \), is \( |B_1| + |B_2| + \ldots + |B_N| \). The blocks arrive in some order implicitly by arrival time or explicitly by timestamp, and may be seen only once.

**Definition 2.** An itemset, i.e., a set of chord-sets, \( X \subseteq \Psi \) is a non-empty subset of \( \Psi \). For brevity, itemset \( X = \{x_1, x_2, \ldots, x_j\} \) can also be denoted as \( X = x_1 x_2 \ldots x_j \). A \( k \)-itemset is consisted of \( k \) chord-sets and is represented by \( x_1 x_2 \ldots x_k \). The support of an itemset \( X \), denoted as \( \text{Sup}(X) \), is the number of melody sequences within the melody stream \( MS \) in which that itemset occurs as a subset. An itemset \( X \) is frequent temporal (or frequent for short), if its support is no less than \( \text{minsup} \times |MS| \), where \( \text{minsup} \) is a user-specified minimum support threshold (between zero and one) and \( |MS| \) is the current length of the melody stream with respect to itemset \( X \).

For example, assume that an itemset \( ABC \) first appears in block \( B_{10} \) of \( MS \), then \( |MS|_{ABC} = |B_{10}| + |B_{11}| + \ldots + |B_{10}N| \), where \( N \) is the current block identifier in \( MS \).

**Definition 3.** A frequent temporal itemset \( Y \) is called a closed itemset if there exists no itemset \( Y' \) such that \( Y' \) is a proper superset of \( Y \), and (2) every melody sequence containing \( Y \) also contains \( Y' \).

**Lemma 1.** Let \( X \) and \( Y \) be two itemsets, and \( \text{Sup}(X) = \text{Sup}(Y) \). \( Y \) is not a closed itemset, if \( Y \subsetneq X \).

For example, we assume that there are two melody sequences \( S_1 = (a_1, a_2, \ldots, a_{10}) \) and \( S_2 = (a_1, a_2, \ldots, a_{10}) \) in \( B_N \), and assume that \( \text{minsup} = 0.5 \). Therefore, we can find \( 2^{100} \) frequent items (i.e., \( a_1, a_2, \ldots, a_{10} \)) in \( B_N \). For \( 2 \) frequent closed itemsets (i.e., \( a_1 a_2 \ldots a_{10} \) with support 2 and \( a_1 a_2 \ldots a_{10} \) with support 1) in \( B_N \), we can find that closed structure is more suitable for music stream mining since its space requirement is considerably small.

**Problem Definition.** Given a \( MS = [B_1, B_2, \ldots, B_N] \), and a \( \text{minsup} \). We wish to determine the set of frequent temporal closed melody structures (i.e., frequent temporal closed itemsets) that have support no less than \( \text{minsup} \times |MS| \) set of frequent closed structures.

### 3. Mining Streaming Melody Sequences for Frequent Closed Structures

The framework of FCS-stream is derived from the well-known pattern-growth approach \( \text{FP-growth} \) proposed by Han et al. [4], which is a divide-and-conquer method. We briefly review the FP-growth method as follows.

#### 3.1 Pattern Growth Mining

First of all, FP-growth builds a header table and a prefix-tree-based structure \( \text{FP-tree} \) (Frequent Pattern tree) by two database scans for storing all necessary information about frequent itemsets in the database. The \( \text{FP-tree} \) is a compact representation (but not always minimal) of all relevant occurrence information of the database. Each branch of the \( \text{FP-tree} \) represents a frequent itemset, and the nodes along the branches are stored in support decreasing order, with leaves representing the least frequent items. Furthermore, \( \text{FP-tree} \) has a header table associated with it. The entry of an item in this table contains their item identifier, item counts, and the head of a list that links all the corresponding nodes in the \( \text{FP-tree} \).

After the construction of the \( \text{FP-tree} \), FP-growth uses a recursive method of conditional FP-tree construction and frequent pattern testing to find all frequent itemsets in all FP-trees. More precisely, the FP-growth method relies on the following principle: if \( X \) and \( Y \) are two itemsets, the count of itemset \( X \cup Y \) in the database is exactly that of \( Y \) in the restriction of the database to those transactions containing itemset \( X \). The restriction is called the \( \text{conditional pattern base} \) of \( X \), and the FP-tree constructed from the \( \text{conditional pattern base} \) is called \( X \)’s \( \text{conditional FP-tree} \). To generate frequent itemsets, FP-growth constructs each entry \( Z \)’s conditional FP-tree of the header table. Then recursively construct the conditional FP-tree based on \( Z \)’s conditional pattern base, and it stops when the resulting new conditional FP-tree contains only one single path. The complete set of frequent itemsets is generated by combining the results of enumeration of the single-pass FP-tree and previous results of conditional FP-tree. More details about the FP-growth method can be found in [4].

There are several problems in developing pattern growth-based method for mining streaming data. First, it requires two data scans to construct the in-memory summary data structure. Second, the space requirement of conditional FP-trees in recursive finding frequent patterns is probably undetermined. These challenges show that previous pattern-growth-based methods cannot appropriately meet the performance requirements in mining streaming data.

Can we develop a pattern-growth-based method that needs only one streaming data scan and utilize some optimizations to reduce the time and space cost in mining streaming melody sequences for frequent temporal closed structures? The answer is yes by using a stream-efficient pattern growth technology.

#### 3.2 FCS-stream: Stream-Efficient Pattern Growth

Algorithm FCS-stream consists of three modules: FCS-buffer, TCS-tree and FCS-mine. First, FCS-buffer repeatedly reads a block of melody sequences into available main memory. Then, all essential closed information of this block is stored in the TCS-tree by parallel projection of melody sequence. After the TCS-tree construction, FCS-mine finds all frequent temporal closed structures in a “top-down” manner in the current TCS-tree. Hence, the main challenges lie in...
designing a space-efficient representation for the in-memory summary data structure and a fast closed pattern discovery algorithm.

The construction process of TCS-tree is described as follows. First of all, each melody sequence, such as \( X = x_1x_2...x_k \), in the current block \( B_k \) maintained by FCS-buffer should be parallel projected in the TCS-tree by inserting \( k \) item-prefix melody sequences, i.e., \( x_i|X = x_1x_2...x_i, x_i|X = x_2x_3...x_i, ..., x_k|X = x_kx_1x_2...x_k \). The method is called the parallel projection, and denoted as \( PP(x) = \{x_i|X, x_{i+1}|X, ..., x_k|X\} \). Then, we drop \( X \) from FCS-buffer after \( PP(X) \).

The TCS-tree can be designed as follows.

**Definition 4.** A Temporal Closed Structures tree (or TCS-tree for short) is an extended prefix-tree structure defined as follows.

1. **TCS-tree** consists of a Dynamic Header Table (or DHT for short) and a set of item-prefix **TCS-tree** (or iTCS-tree for short).
2. Each entry in the DHT consists of four fields: item-id, support, block-id and head-link, where support registers the number of melody sequences containing the item carrying the item-id, the value of block-id assigned to a new entry is the block identifier of current block, and head-link points to the root node with item-id of the iTCS-tree. In other words, each entry \( i \) in DHT is an item-prefix \( i \), and it is a root node of iTCS-tree.
3. Each node in the iTCS-tree consists of four fields: item-id, support, block-id, and node-link, where item-id is the identifier of the inserting item, support records the number of melody sequences represented by a portion of the path reaching the node with item-id, the value of block-id assigned to a new node is the block identifier of current block, and node-link links to the next node in the iTCS-tree carrying the same item-id. If no such node, the node-link is null.
4. Each iTCS-tree has a specific item-prefix DHT (or iTDHT for short), where iTDHT works as DHT, but only records the id, support, and block-id of the item whose prefix item carrying item-id \( i \) in this iTDHT. Notice that |iTDHT| = |DHT| in worst case.

To construct the TCS-tree, FCS-stream first reads a melody sequence \( X = x_1x_2...x_k \) from the FCS-buffer. Then, it parallel projects this sequence and inserts these item-prefix melody sequences \( x_i|X \) into DHT and iTCS-tree. In details, the set of items in the item-prefix melody sequence \( x_i|X \) are inserted into the iTCS-tree as a branch and updates the iTDHT. If an itemset shares a prefix with an itemset already in the tree, the new itemset will share a prefix of the branch representing that itemset. In addition, a support counter is associated with each node in the tree. The counter is updated when the next item-prefix melody sequence causes the insertion of a new branch. After pruning all infrequent information, the TCS-tree contains all closed information of the stream. Let’s us examine an example as follows.

**Example 1.** Let the first block \( B_1 \) of melody sequence stream be \( \{acdef, abc, cef, acdef, cef, df\} \), and the second block \( B_2 \) be \( \{def, bef, bde, be\} \), and \( \text{minsup} = 30\% \). Figure 2 is the TCS-tree after inserting \( B_1 \), and the TCS-tree after pruning some infrequent patterns is shown in Figure 3. Moreover, the TCS-tree after inserting second block without pruning is shown in Figure 4.

The mining principle of FCS-mine is described as follows. Given an item \( i \) in the current DHT, FCS-mine generate candidates by “top-down” enumerating the combination of frequent items in the iTDHT. Then it checks these candidates whether they are frequent closed ones or not by traversing the iTCS-tree. For example, for an item \( c \) of DHT in Figure 4, FCS-mine first generates a maximal candidate \( cef \) based on the cDHT, and it traverses the cTCS-tree for evaluating its support. At that time, FCS-mine finds that \( cef \) is a frequent structure carrying \( \text{Sup}(c) = 3 \). Notice that now the frequent threshold is 3. Next, FCS-mine generates two candidates \( ce \) and \( cf \) based on \( cef \), and finds their support directly from the cDHT without traversing the cTCS-tree, since they are 2-itemsets. The support of \( ce \) and \( cf \) are 3 and 4, respectively. By lemma 1, we find that itemsets \( c \) and \( cf \) are frequent structures, but they are not frequent closed structures in this case. Therefore, there are only two frequent closed structures, i.e., \( (ce: 4) \) and \( (cef: 3) \), in cTCS-tree.
4. Complexity Analysis

4.1 Space Complexity

Assume that the DHT of FCS-stream contains $k$ chord-sets at any time. Therefore, we know there are at most $C_{k/2}^{[k/2]}$ maximal frequent chord-sets in the current music data stream seen so far. If we construct a TCS-tree for all these maximal frequent itemsets, the tree has height $\lceil k/2 \rceil$. In the first level, there are $C_{k/2}^{[k/2]}$ nodes, in the second level, there are $C_{k/4}^{[k/4]}$ nodes, in the $i$-th level, there are $C_{k/2^i}^{[k/2^i]}$ nodes, and in the last level, the $\lceil k/2 \rceil$ level, there are $C_{k/2}^{[k/2]}$ nodes. Thus, the total number of nodes is

$$C_{k/2}^{[k/2]} + C_{k/4}^{[k/4]} + \ldots + C_{k/2^i}^{[k/2^i]} = \sum_{i=1}^{\lceil k/2 \rceil} C_{k/2^i}^{[k/2^i]}.$$

The space requirement of FCS-stream consists of three parts: the working space needed to create a DHT, and the storage space needed for the iDHT and the set of iTCS-trees. In worst case, the working space for DHT requires $k$ entries. For storage, there are at most $\sum_{i=1}^{\lceil k/2 \rceil} C_{k/2^i}^{[k/2^i]}$ nodes of the set of iTCS-trees, and $(k^2-k)/2$ nodes for all iDHT. Thus, the total space bound of FCS-stream is $\frac{k}{2}(k^2+k)+\sum_{i=1}^{\lceil k/2 \rceil} C_{k/2^i}^{[k/2^i]}$.

4.2 Time Complexity

First, from the construction process of TCS-tree, we can see that one needs exactly one streaming data scan. Second, the cost of inserting a melody sequence $S$ into the TCS-tree is $O(|S|^3)$, where $|S|$ is the number of chord-sets of melody sequence $S$ within the DHT.

5. Performance Analysis

In this section, we report our performance study of algorithm FCS-stream. All the experiments are performed on a 1GHz IBM X24 with 128MB, and the program is written in Microsoft Visual C++6.0. Limited by space, we reported here only the results on two synthetic datasets: S10.15 and S30/15, using a standard procedure described in [1]. The first dataset has average melody sequence size of 10 with average frequent temporal closed structure size of 5. In the second dataset, the average melody sequence size and average frequent temporal closed structure size are set to 30 and 15, respectively. Both synthetic datasets have 1,000,000 melody sequences.

5.1 Time and Space Usage

In this experiment, we examine the two primary factors: execution time and memory, in the problem of mining frequent temporal closed structures in streaming melody sequences, since both should be bounded online as time advances. Therefore, in Figure 5 (a), the execution time grows smoothly as the dataset size increases. We assume that $\text{msup} = 0.01\%$. The memory usage in Figure 5 (b) for both synthetic datasets is stable as time progresses, indicating the scalability and feasibility of algorithm FCS-stream. In this experiment, the synthetic music melody stream is broken into blocks with size 50K.

6. Conclusions

In this paper, we propose a first stream-efficient pattern-growth algorithm FCS-stream with a space-efficient summary structure TCS-tree to exploit the frequent closed structures in streaming melody sequences. Our performance study shows that FCS-stream is efficient and scalable over a continuous stream of melody sequences.

Acknowledgements

The authors thank the reviewers’ precious comments for improving the quality of this paper. The work was supported by the National Science Council of R.O.C. under grants no. NSC92-2213-E009-123.

References