Minits-AllOcc: An Efficient Algorithm for Mining Timed Sequential Patterns

Somayah Karsoum[1], Clark Barrus[1], Le Gruenwald[1], and Eleazar Leal[2]

[1]University of Oklahoma, Norman, OK 73019, USA

{somayah.karsoum, clark.barrus, ggruenwald}@ou.edu

[2] University of Minnesota Duluth, Duluth, MN 55812, USA

eleal@d.umn.edu

Abstract. Sequential pattern mining aims to find the subsequences in a sequence database that appear together in the order of timestamps. Although there exist sequential pattern mining techniques, they ignore the temporal relationship information between the itemsets in the subsequences. This information is important in many real-world applications. For example, even if healthcare providers know that symptom Y frequently occurs after symptom X, it is also valuable for them to be able to estimate when Y would occur after X so that they can provide treatment at the right time. Considering temporal relationship information for sequential pattern mining raises new issues to be solved such as designing a new data structure to save this information and traversing this structure efficiently to discover patterns without re-scanning the database. In this paper, we propose an algorithm called Minits-AllOcc (MINIng Timed Sequential pattern for All-time Occurrences) to find sequential patterns and the transition time between itemsets based on all possible occurrences of a pattern in the database. We also propose a parallel multicore CPU version of this algorithm, called MMinits-AllOcc (Multicore Minits-AllOcc), to deal with Big Data. Extensive experiments on real and synthetic datasets show the advantages of this approach over the brute-force method. Also, the multicore CPU version of the algorithm is shown to outperform the single-core version on Big Data by 2.5X.

Keywords: Sequential pattern mining, Timed sequential pattern, Multicore, Parallel Sequential Pattern Mining.

1 Introduction

Sequential pattern mining [1] is a data mining task that discovers frequent subsequences in a sequence database of time-ordered transactional data. Finding interesting, useful, and unexpected patterns is beneficial for a wide range of real-world applications such as illness symptom pattern prediction [12], network intrusion detection [18], and customer shopping behaviors [1]. Existing sequential pattern mining algorithms such as [22],[17], and [9] use implicit timestamps to order the itemsets within a sequential pattern but the transition time between these itemsets is not kept. In many applications, it is important to know the time to move from one itemset to another in the pattern. For example, in healthcare applications, knowing when the next symptom of heart attack will occur assists healthcare providers in forming diagnoses, providing treatments at the right time, and intervening earlier in critical cases. For monitoring weather forecasts in Oklahoma during the tornado season, we want to be able to track the transition time range between cities when a tornado hits multiple cities in the timestamp order. With sequential patterns that also contain the temporal relationship about the transition time, which indicates when the next symptoms will appear, we are answering not only a question like in which order the symptoms for heart attack frequently occur, but also questions like when the symptoms for heart attack frequently occur.

Let us suppose that we have the historical health information of temperature (T) and blood pressure (BP) for patients who have had a heart attack. The time is recorded when each of the measurements is taken for each patient. Since sequential pattern mining algorithms do not deal with continuous data, we need to apply a discretization technique in order to segment the data into classes that have similar features or fall within the same group. For instance, the blood pressure (BP) has five levels [4]: (1) Normal (BP < 120), (2) Elevated (120 ⩽ BP ⩽ 129), (3) High Stage 1 (131 ⩽ BP ⩽ 139), (4) High Stage 2 (140 ⩽ BP ⩽ 180), and (5) Crisis (BP > 181). Therefore, we refer to the blood pressure with the abbreviation BP followed by the class number in which the blood pressure falls. Since the temporal information is available in time-ordered transactional data, we can discover more informative sequential patterns that not only show the symptoms that frequently occur among patients but also include the typical transition times between symptoms (in terms of number of weeks in our example). We call this special type of sequential patterns Timed Sequential Patterns (TSP). For example, < {T1, BP3} [2,7] {T2}> is a TSP that has two itemsets: itemset 1 consisting of two items T1 and BP3, and itemset 2 consisting of item T2, Itemset 2 occurs within 2 to 7 weeks after itemset 1. In our notations, all items enclosed within braces { } occur at the same time and constitute an itemset, and the square brackets [ ]

indicates the time duration to move from one itemset to the next itemset. Thus, the previous example of TSP shows that when patients have a temperature in class 1 (T1) and a blood pressure in class 3 (BP3), then within 2 to 7 weeks, the patients will have a temperature in class 2 (T2). If we apply traditional sequential mining, then this pattern would only be < {T1, BP3} {T2}> which does not include the transition time [2, 7].

Incorporating the temporal information in a sequential pattern raises additional challenges when compared to the regular sequential pattern mining. First, while both sequential pattern mining and timed sequential pattern mining need to find out whether a pattern occurs in a sequence database tuple, timed sequential pattern mining also needs to find out how many times the pattern occurs in that tuple to compute the temporal relationship between the itemsets in the pattern. So, to find all possible occurrence of the pattern; the naïve mechanism is required to scan each tuple in the database from the beginning until the end. Unlike sequential pattern mining, an algorithm will stop checking the rest of the tuple in the database as soon as the pattern is found. In other words, in the best case, sequential pattern mining techniques do not need to check until the end of each sequence in the database. However, timed sequential pattern mining requires checking all the sequences in the database. For example, if we have a tuple of a patient P1 in the database that has all measurements within six months and usually the following symptoms occur many times in the patient's tuple: high temperature followed by low blood pressure after some time. Since the timed sequential patterns mining problem wants to know when the low blood pressure occurs, it is not sufficient to find only the first position of this pattern and report the temporal relation. Instead, it is necessary to consider all possible occurrences of that pattern and also all the different timestamps of each occurrence and find the temporal relation. Second, the temporal information must be updated for each pattern that is discovered based on the timestamps of the tuples that contain the pattern. For example, after we discover the pattern from the patient P1 and calculate the temporal relations, we also find the same pattern, high temperature followed by low blood pressure, in another tuple for another patient Pi in the database. That means the temporal relations need to be updated to represent the actual time duration. So, we need to capture all the occurrences of that pattern for Pi and re-calculate the temporal relations. On top of that, we need to keep track of timestamps of all occurrences of the patterns for both patients P1 and Pi to finalize the temporal relationships. The brute force technique needs to scan the database again to retrieve or store the required information for P1. Thus, for every pattern, we need to scan the whole database many times to make sure that we have the correct temporal relations. Of course, this requests more space and time that obviously impacts the performance of any algorithm.

The existing techniques [3],[8], and [21], which will be discussed in detail in Section III, do not address these challenges. To fill this gap, in this paper we propose a timed sequential pattern mining algorithm, called Minits-AllOcc (MINIng Timed Sequential pattern for All-time Occurrences), that addresses all these challenges. We also validate the proposed algorithm through a set of empirical experiments. The contributions of this paper are the following:

1. The idea of incorporating transition time between itemsets in a sequential pattern, which indicates all possible time occurrences of the pattern within whole timed sequence database. The temporal relations required time to move from one itemset to the next itemset in the timestamp order. The time can be any descriptive statistic based on the user's preference, such as range, average, etc.
2. The parallel implementation of the Minits-AllOcc algorithm and the extensive experiments comparing the single-core vs. multi-core algorithms on real and synthetic datasets.

The remainder of this paper is organized as follows. Section 2 defines the timed sequential pattern mining problem. Section 3 reviews the related works. Section 4 explains the proposed techniques for the Minits-AllOcc algorithm. Section 5 presents the results of the experiments on the dataset. Finally, the conclusion and future work are presented in Section 6.

2 Problem Definitions

In this section, we review the definitions of the sequential pattern mining problem and introduce new definitions for the timed sequential pattern mining problem. Recalling the traditional sequential pattern mining problem [1], an **itemset** I is a set of **items** such that $I \subseteq X$, where $X = \{x_1, x_2, \ldots x_l\}$ is a set of items. A **sequence** (tuple) s is an ordered list (based on timestamps) of itemsets. A sequence A = < {a_1}, {a_2}, …{a_n}> is **contained in** another sequence B = < {b_1}, {b_2}, …{b_m}> and B is **super-sequence** of A, if there exists a set of integers, $1 \leq j_1 < j_2 < \ldots < j_n \leq m$ such that $a_1 \subseteq b_{j_1}$, $a_2 \subseteq b_{j_2}$, ... , $a_n \subseteq b_{jn}$.

A **sequence database** S is a set of sequences (tuples) <sid, s_i> where sid is a sequence identifier and s_i is a sequence. A tuple <sid, s_i> is said to contain a sequence α, if α is a subsequence of s_i. Since our problem considers the temporal data too, we incorporate timestamps explicitly in the database and introduce new definitions.

Definition 1. A timed ***event*** is a pair $e = (I, t)$ where I is an itemset that occurs at the timestamp t. We use e.I and e.t to indicate, respectively, the itemset I and the timestamp t associated with the event e. The list of events that is sorted

in the timestamp order is called a ***timed sequence*** *TS* = $<\{e_1\}, \{e_2\}, ..., \{e_n\}>$, such that $e_i.x \subseteq I$ ($1 \leq i \leq n$). A ***timed sequence database TSDB*** is a set of sequences < *TS_id, TS*>, where *TS_id* is a timed-sequence identifier and *TS* is a timed sequence.

Example 1. (**Running example**) The timed sequence database in Fig. 1 is used as an illustrative example in this paper. For simplicity, we will use letters to refer to items which represent different properties of objects in the database (e.g., temperature and blood pressure for patients), and integer numbers to refer to timestamps, which represent the times when those properties are collected. In this example, there are four timed sequences with IDs from TS1 to TS4. Each timed sequence consists of a set of events ordered in the events' timestamps. For example, TS1 consists of two events: the first event {a, b, 5}, which occurred at timestamp 5, followed by the second event {d, g, 12}, which occurred at time stamp 12.

Definition 2. Given a sequence A = $<\{I_1\}, \{I_2\}, \ldots \{I_n\}>$ and a timed sequence TS = $<\{e_1\}, \{e_2\}, \ldots, \{e_m\}>$, the ***All-time Occurrences*** of A in TS in the timed sequence database TSDB is defined as an ordered list of indices $1 \leq j_1 < j_2 < \ldots < j_n \leq m$ such that: $I_1 \subseteq e_{j_1}.I$, $I_2 \subseteq e_{j_2}.I$, ... $I_n \subseteq e_{j_n}.I$. The ***deltas*** $\Delta$are defined as $\Delta = e_{p.j_{i-1}}.t - e_{p.j_i}.t$.

Example 2. Let sequence A = <{a}{b}> and timed sequence TS4 = <{a,10},{b,f,19},{d,20},{b,30}>, as shown in Fig. 1. The indices of the events for the first occurrence of sequence A in TS4 are $\{e_1, e_2\}$ as the solid arrow is shown in Fig. 2. The delta Δ is the difference between the timestamps of these two consecutive events, which is $e_1.t_1 = 10$ and $e_2.t_2 = 19$. Thus, the $\Delta = 19 - 10 = 9$. Then, the second occurrence of sequence A in TS4, as the dotted arrow is shown in Fig. 2, has the following events' indices $\{e_1, e_4\}$. The delta Δ is the difference between the timestamps of these two consecutive events, which is $e_1.t_1 = 10$ and $e_4.t_2 = 30$. Thus, the $\Delta = 30 - 10 = 20$. Similarly, we can find the rest of the All-time Occurrence. The **support** of a sequence A in a sequence database, or a timed sequence database, is the percentage of the number of sequences in the database that contains A such that sup(A) = (#sequences that contains A / #sequences in DB)*100. If the support of sequence A is greater than or equal to a user defined threshold called minimum support (min_sup), then it is called a **sequential pattern** [1].

Definition 3. A sequence A is called a **timed sequential pattern TSP**, if and only if it is a sequential pattern and accompanied by **temporal relationships** τ_i between itemsets where it represents any descriptive statistic, such as average of transition time or range, calculated based on values of delta Δ. TSP denoted as: TSP = $< \{I_0\} [\tau_1] \{I_1\} [\tau_2] \{I_2\} \ldots\ldots [\tau_n] \{I_n\}>$. For brevity, when we mention a pattern, we refer to a timed sequential pattern.

Example 3. Let us assume the min-sup =50%, since the support of sequence A= <{a}{b}> is 50%, the sequence is a sequential pattern. In this paper, we assume that a user chooses the temporal relation to be presented as a range of time [min, max]. Thus, the timed sequential pattern version is: <{a} [9,20] {b}>. Thus, the timed sequential patterns are originally sequential patterns that satisfy the min_sup condition and clearly state the transition time between itemsets.

Timed Sequence ID	Timed Sequences TS
TS1	< {a,b,5} , {d,g,12} >
TS2	< {e,g,21} >
TS3	< {a, 2} , {a,b,19} , {d,25} >
TS4	< {a, 10} , {b,f,19} , {d,20} , {b,30} >

Fig. 1. An example of Timed Sequence Database

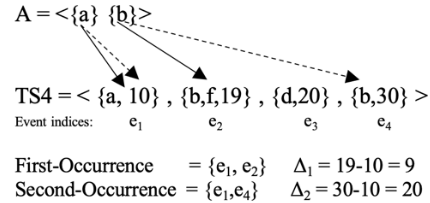

Fig. 2. All time Occurrence of A in TS2

3 Related Works

The concept of sequential pattern mining was first introduced in [1], where three algorithms were proposed: AprioriSome, DynamicSome, and AprioriAll algorithms for discovering sequential patterns. AprioriAll was the basis of many other efficient algorithms that have been proposed to improve its performance. Those algorithms inspired [19] to propose a technique to generate fewer candidates called GSP. Since all algorithms were based on the Apriori algorithm, therefore, they were classified as Apriori-based algorithms. Other algorithms such as SPADE [22] adopted a vertical id-list database format that reduced the number of database scans. In contrast, pattern-growth based algorithms, such as FreeSpan [9] and PrefixSpan [17], used the concept of database projection, which made them more efficient than other Apriori-based algorithms, especially when they dealt with a large database. These algorithms generated a smaller database for their next pass because the sequence database was projected into a set of smaller databases and then sequential patterns in each of them were explored. Thus, they were more efficient. More literature reviews about the state-of-art sequential pattern mining algorithms can be found in [5].

Recently, with the existence of a large volume of data available in many applications, several sequential pattern mining algorithms had been proposed to handle large databases consisting of huge amounts of sequences efficiently using

different platforms. For example, [11] used the multi-core processor architecture for implementing pDBV-SPM to improve processing speed for mining sequential patterns. Ha-GSP [16] adopted the principles of GSP and implemented them on the Hadoop platform for solving the limited computing capacity and insufficient performance with massive data of the traditional GSP. MR-PrefixSpan [20] used the MapReduce platform to implement the parallel version of PrefixSpan to mine sequential patterns on a large database. More literature reviews about the state-of-art parallel sequence mining algorithms are in [6].

In a sequential pattern, objects have an ordinal correlation based on the timestamp precedence. We can obtain a sequence by sorting all these objects based on the order of their timestamps. However, the time between itemsets is discarded. This kind of sequential pattern that incorporates temporal relations is more informative for some applications. Some techniques were proposed to specify some timing constraints, such as the time gaps between adjacent itemsets in sequential patterns. For example, [3] modified the Apriori [1] and PrefixSpan [17] algorithms to discover the time-interval sequential patterns that satisfied the interval duration boundaries. The I-PrefixSpan algorithm in [3] has another input which is called a set of time-intervals TI, where each time-interval has a range. [10] extend the work of [3] and proposed two algorithms: MI-Apriori and MI-Prefix. The time- intervals incorporated in the patterns revealed the time between all pairs of items in a pattern, which is called multi-time-interval sequential patterns. A list of intervals (ti_3, ti_2, ti_1) before item d in a pattern like <a, ti_1,b,(ti_2,ti_1),c,(ti_3,ti_2,ti_1),d> means the intervals between items a, b, and c and item d are ti_3, ti_2 and ti_1, respectively. [2] also extracted the sequential patterns of diseases from medical dataset within user-specified time intervals. The drawback of these methods is that their results will miss some frequent patterns that do not fulfill the time constraint. To decide if a pattern is frequent, two conditions must be satisfied: the support of the pattern must be greater than or equal to min_sup, and the time range between itemsets must lie within the defined time intervals. Therefore, if a pattern fulfills the first condition, which means it is frequent but does not fulfill the second condition, the algorithm will not report it.

[7] incorporated the temporal dimension in the sequential pattern by defining temporally annotated sequences (TAS), and [8] proposed the Trajectory Pattern algorithm (T-pattern) to extract a set of TAS to produce trajectory patterns with a fixed amount of time to travel between places. The algorithm only worked with one-dimensional data that did not represent the real cases in real life such as healthcare applications. Also, the time between events in trajectory patterns is strict, which does not consider different cases of traveling between locations such as transportation types. [21] relaxed the travel time to be a realistic range for traveling time. Nevertheless, the algorithm still cannot deal with multidimensional data because it is dealing only with locations in trajectory data. Also, all the previous techniques did not consider all possible occurrences of a pattern in an individual sequence in the database, which means the temporal relations are calculated based on the first occurrence of a pattern. The issue of calculating the time-intervals of the first occurrence of a pattern and ignore other occurrences was address in [15]. However, this approach is beneficial for a limited number of applications. For example, if a developer wants to evaluate the ease of use a navigation system, the time of moving from A to B is tested when the users visit those location for the first time. In contrast, in other applications such the healthcare, which is mentioned above, we must consider all possible occurrences to provide an accurate time-intervals. To the best of our knowledge, there is no existing algorithm that can find the complete set of timed sequential patterns in which each pattern represents the transition time between successive itemsets in a pattern and consider all possible occurrences.

4 The Proposed Algorithm: Minits-AllOcc

To discover timed sequential patterns, we propose a timed sequential pattern mining algorithm called Minits-AllOcc. We first describe the occurrence tree, which is the core data structure of the algorithm, in Section 4.1. Then, in Section 4.2 and 4.3, we introduce an overview of Minits-AllOcc and how it works in detail. In Section 4.4, we propose some enhancements to improve the efficiency of the algorithm's performance.

4.1 Occurrence Tree (O-Tree)

The occurrence tree *O-tree* is a data structure used to represent the different possible occurrences of a pattern in a timed sequence in TSDB. This tree is the seed of the algorithm because it helps to generate timed sequence patterns without scanning the timed sequence database many times. The O-tree has two types of nodes: root node, which contains a timed sequence ID, and regular nodes. A regular node has the following information (1) the event ID e_{ID} and (2) its timestamp $e_{ID}.t$. The edge between two regular nodes represents the difference Δ between the timestamps associated with the two nodes. For example, sequence <*{a}*> appears twice in TS3, thus, its O-tree in Fig. 3 has two nodes connected to the root. However, sequence <*{a} {a}*>appears once in TS3 that has two nodes too, but one is connected to the root and the other one is connected to a regular node via an edge Δ, which is 19-2=17. Since each

sequence has an O-tree for each timed sequence in TSDB that contained it, the sequence will have a collection of O-trees that identify its occurrence in the whole TSDB. Thus, we give the following definition:
Definition 4. Given a sequence A and timed sequence database TSDB, ***A- forest*** is a collection of all O-trees that identify all possible occurrences of the sequence A in TSDB. Fig 5. Demonstrates the forests of four sequences <{a}>,<{b}>,<{a}[9,20]{b}>, and <{a,b}>. Each forest is surrounded by dotted rectangle, which has group of O-trees that indicates all time-occurrences of a sequence in TSDB.

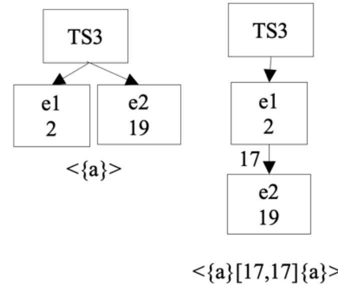

Fig. 3. An O-tree for sequences <{a}> and <{a} {a}> in TS3

4.2 Overview

The main goal of Minits-AllOcc is to find the complete set of the timed sequential patterns that satisfy the min_sup threshold condition from a given TSDB. To achieve this goal, Minits-AllOcc utilizes the forests to store all required information from timed sequences in TSDB. The following steps are performed: (1) Scan TSDB to build a I_j-forest for each distinct item I_j. (2) Find frequent items by counting the number of O-trees in each forest, compare it against the min_sup threshold, and remove the infrequent items. (3) Merge all O-trees that have the same root from different forests to build a new forest for a candidate sequence. It should be mentioned that there are two different relations between itemsets considered while merging step: *event-relation* and *sequence-relation* which are defined as:
Definition 5. Given two items X and Y, it is said X and Y have an ***Event-relation*** e-relation between them denoted as <{X,Y}> if X and Y occur in the same event.
Definition 6. Given two items X and Y, it is said X and Y have a ***Sequence-relation*** s-relation between them denoted as <{X} {Y}> if X and Y occur in two different events and the event of X occurs before the event of Y.
(4) Find the timed sequential patterns among candidate sequences by counting the number of O-trees in each forest, compare it against the min_sup threshold, and ignore the infrequent sequences. By doing step 4, Minits-AllOcc avoids scanning TSDB for each candidate to calculate the support. (5) Compute the temporal relation of the suffix, the new appending part to the pattern, and update the temporal relation of the prefix, the previous part of the pattern. (6) Repeat steps 3, 4, and 5 until the algorithm cannot identify any new timed sequential pattern. Minits-All Occ's pseudocode is presented in Fig. 4.

4.3 The Proposed Algorithm: Minits-AllOcc

In this section, we describe the above steps in detail using the running example shown in Fig. 1. At first, the algorithm starts reading the TSDB row by row and builds the associated O-tree for each distinct item until all forests are completed (line 1). As shown in Fig.5 for instance ,after the algorithm finishes scanning TSDB, <{a}>-forest has three O-trees because sequence <{a}> appears in three timed sequences TS1, TS3, and TS4. Then, the algorithm excludes the infrequent sequences by calculating their supports using the number of O-trees in each forest. The two sequences <{e}> and <{f}> are not frequent because their forests have only one tree, which means they appear in one TS, therefore, their support is 25%. The following is the set of 1-timed sequential patterns = <{a},{b},{d},{g}> (line 2). The third step is generating candidates by merging the O-trees of all 1-timed sequential patterns, so the algorithm calls function *find-TSPs* (line 3). The mechanism of merging is the follows: if the relation is s-relation, the appended node must have an event ID e_i that is greater than the parent(line 11-14). Then, the edge holds the difference between the timestamps of the parent and its child(line 15). In contrast, if the relation is e-relation, the appended node must have the same event ID e_i of its parent(line 23-26). For instance, the forest of the two candidates <{a}[ ]{b}>, which represents the s-relation, and <{a,b}>, which represents the e-relation, is shown in Fig. 5. The first <{a}[ ]{b}>-forest has two O-trees that are generated by combining the <{a}>-forest and <{b}> -forest. Even though both forests have an O-tree that has a root TS1, the O-tree of <{b}> does not contain a node that has an event ID greater than e_1, thus, it was removed from the <{a}[ ]{b}>-forest. In contrast, the node that has e_2 from <{b}>-forest is attached to the node that has e_1 from <{a}>-

```
Algorithm: (Minits-AllOcc)
Input: Timed Sequence Database TSDB,
       minimum support threshold min-sup
Output: The complete set of Timed sequential patterns TSPs
 1   Scan TSDB and build 1-sequence forest for each distinct item
 2   Add frequent 1-seq into TSPs
 3    for each frequent 1-seq
 4      Call find-TSPs (1-seq forest, 1-seq forests)
 5    end for
 6   function find-TSPs (k-seq forest, 1-seq forests)
 7     for each possible combination between k-seq and 1-seq
 8       if (s-relation)
 9         for each common root (TSi)
10           if (O-tree(k-seq). node.EventId < O-tree(1-seq).node.EventId)
11             Append node(1-seq) to O. tree(k-seq)
12             Delta = O-tree(k-seq).node.timestamp -
                       O- tree(1seq).node.timestamp
13           end if
14         end for
15         if (# O-trees (k+1-seq) in forest ≥ min-sup)
16             Traverse k+1-seq forest to update temporal stamp
17             Add k+1-seq into TSPs
18             Call find-TSPs (k+1-seq forest, 1-seq forests)
19         else
20             return
21         end if
22       else if (e-relation)
23         for each common root (TSi)
24           if (O-tree(k-seq).node.EventId == O-tree(1-seq).node.EventId)
25              Append node(1-seq) to O.tree(k-seq)
26              Delta = 0
27           end if
28         end for
29         if (# O-trees (k+1-seq) in forest ≥ min-sup)
30             Add k+1-seq into TSPs
31             Call find-TSPs (k+1-seq forest, 1-seq forests)
32         else
33             return
34         end if
35       end if
36     end for
37   end function
```

Fig. 4. Pseudo-code of the Minits-AllOcc Algorithm

forest, and the Δ is calculated between those nodes, which is 19-2 =17. However, the node that has e_2 from <{a}>-forest does not connect to any node. Since the algorithm is looking for all possible occurrences, the node in TS4 that has e_1 is connected to the two nodes, which has event ID e_2 and e_4, from <{b}> O-tree and each link carries the difference between the timestamps of the two connected nodes. Because in this example we consider the temporal relations as a range of [min, max], the algorithm chooses the minimum and maximum values among all O-trees in <{a}[ ]{b}>-forest, which is [9,20]. The second <{a,b}>-forest, has two O-trees that are generated by combining the <{a}>-forest and <{b}> -forest. The difference between the technique of merging the trees from the previous case and this one is the condition of appending nodes. Since this is an e-relation, all added nodes must have the same event ID e_i as their parents. Also, the Δ is always 0 because the nodes have the same timestamps. Both patterns <{a}[9,20]{b}> and <{a,b}> are considered to be timed sequential patterns and they are added to TSP set because their supports are 50% (line18-20, 30-31). The supports are calculated as following: # O-trees in the forest / #timed sequences in TSDB*100,(2/4)*100=50%. The algorithm now repeats the same steps, by calling function *find-TSPs* recursively in line 21 and 32, to extend the pattern by merging O-trees, extracting TSPs, and computing temporal relations until no more TSPs can be found. As shown in Fig. 6, pattern <{a} [9,17] {b} [1,6] {d }> is a result of merging between <{a} [9,20] {b}>-forest and <{d}>-forest. The forest displays only the O-trees that represent the pattern, then, the time between the prefix <{a} [ ] {b}> and suffix <{d}> is calculated as defined before (the range). Also, it should emphasize that the time of prefix <{a} [ ] {b}> is updated based on the current forest. Before, it was <{a} [9,20] {b}> but now it is <{a} [9,17] {b} ...>. Again, the <{a} [9,17] {b} [1,6] {d}> is TSPs because its support is 50%. Minits-AllOcc continues repeating the steps until

the complete set of TSPs is discovered. The reader can verify that the TSPs in this example is = {<{a} [9,20] {b}>, <{a} [6,23] {d}>, <{b} [1,7] {d}>, <{a,b} [6,7] {d}>, <{a} [9,17] {b} [1,6] {d }>}.

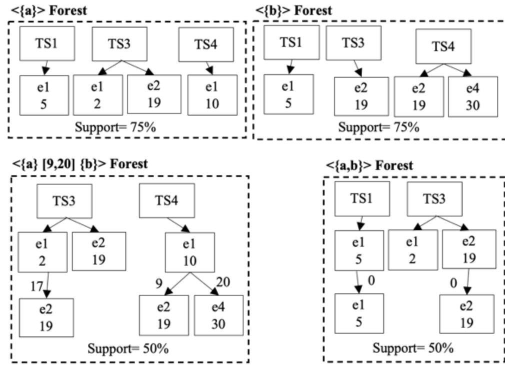

Fig. 5. Merging O-trees of <{a}> and <{b}> to generate <{a,b}>-*forest* and <{a}[9,20]{b}>-*forest*

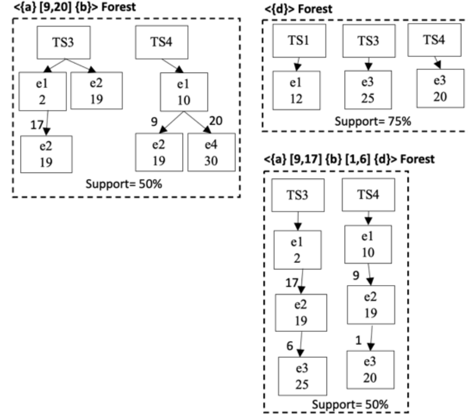

Fig. 6. Merging <{a}[9,20]{b}>-*forest* and <{d}> to generate <{a}[9,17]{b}[1,6] {d}>-*forest*

4.4 The Proposed Enhancement

In this section, we describe some effective mechanisms that help to improve the efficiency of Minits-AllOcc.

1. Pruning the Forests

The idea of this technique is to refine a sequence's forest after merging the O-trees. So, when those O-trees are used in the next step for generating candidates, they carry only the necessary information and therefore save space by removing some nodes and save time by avoiding traversing needless branches in trees. Any branch in an O-tree that does not have a new appended node will be removed after the merging step is executed. Fig. 7 represents the idea by showing the deleted branch of O-trees using the cross symbol. For example, the O-tree that has TS3 root is a result of merging TS3 O-tree from <{a}> and <{b}>-forests. Since there is no appended node to the right branch of <{a}>-forest, this node is removed from <{a} [9,20]{b}>-forest. Those branches do not exist anymore in the O-trees.

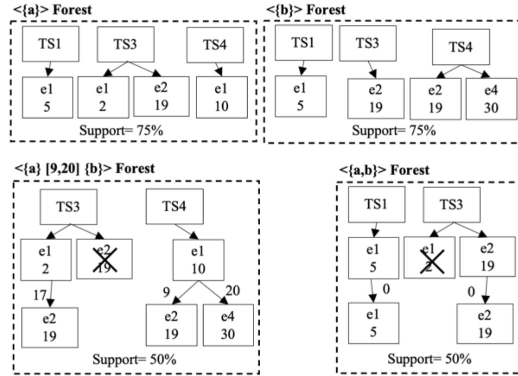

Fig. 7. Pruning the original <{a}[9,20]{b}>-*forest* and <{a,b}>-*forest* in Fig. 5

2. Using frequency matrix

With this technique, we avoid generating unnecessary candidates, which reduces the number of forests. For example, the algorithm uses the 1-sequence-forests to generate 2-sequence candidates, then keeps frequent patterns and removes infrequent ones. Since all required information is already available in the forest, we build a frequency matrix for each sequence to indicate the candidates that are frequent. For example, the frequency matrix of <{a}> pattern is shown in Fig.8. The two different relations: event and sequence (the rows) and all 1-timed sequential patterns that can be combined with {a} (the columns) are considered. The cells under <{b}> column represent the frequency of the two relations between <{a}> and <{b}>. This frequency is calculated from the forests of those patterns as shown in Fig.7. For s-relation, there are two O-trees (TS3 and TS4) in which the <{a}> and <{b}> occur at the different timestamps

within the same timed sequence. For e-relation, there are two O-trees (TS1 and TS3) in which the <{a}> and <{b}> occur at the same timestamps within the same timed sequence. From the matrix, we can infer that <{g}> is not frequent either with s-relation or e-relation, thus, we do not need to build the forest of sequence <{a}[ ]{g}> or <{a,g}>.

<{a}>	a	b	d	g
s-relation	25%	50%	75%	25%
e-relation	0%	50%	0%	0%

Fig. 8. Frequency matrix for <{a}>

3. Using Multicore CPUs

Another enhancement is using multicore CPUs for implementing Minits-AllOcc, which we call it MMinits-AllOcc. A queue is created to hold all jobs of the algorithm and as soon as one thread becomes idle, the next job in the queue is assigned to it. The first mechanism of parallelism is all threads work in parallel when the algorithm recursively generates the patterns. In the serial version, the algorithm starts with the pattern <{a}> and keeps extending it until no more patterns can be found that have prefix <{a}>, for example. Then, it starts with the pattern <{b}> and so on. With the multi-core version, the algorithm works on all patterns <{a}>, <{b}., ..etc at the same time.

5 Performance Analysis

In this section, we describe the environment of experiments and report our evaluation results on the performance of Minits-AllOcc and MMinits-AllOcc considering the impact of different parameters.

5.1 Experimental Setup

All experiments were performed on a computer with a 2.10 GHz Intel Xeon(R) processor with 64 gigabytes of RAM, running Ubuntu 18.04.1 LTS CPU with 12 cores. The Minits-AllOcc and MMinits-AllOcc algorithms are implemented in Java 1.8.

5.2 Datasets and Experimental Parameters

We use real-life and synthetic datasets. The real dataset is T-Drive [13] [14] and the synthetic dataset was generated by using a tool provided by the SPMF Library [4]. Also, we set several parameters to conduct the experiments on the dataset. There are two types of parameters: static and dynamic parameters. The values of the static parameters are not changed in experiments. In contrast, the values of the dynamic parameters are changed from one experiment to another. In this experiment, we have four dynamic parameters. The first one is the minimum support threshold (min_sup). It is a user-defined threshold that is applied to find all timed sequential patterns in a timed sequence database TSDB. The second parameter is the number of timed sequences TS in TSDB (#Seq). The third parameter is the length of TS in TSDB, which is can also be be represented as the number of events per TS (# Events). The last parameter is the number of items in each event (#items). It should be mentioned the timestamp is a fixed attribute in all events. When it said the number of items per event is 3, for instance, it means three items plus the timestamp. We study the impacts of all four parameters shown in Table 1 on the synthetic dataset. However, for the T-Drive dataset, the only valid dynamic parameter is the min-sup. Thus, all other three parameters are considered static. Now, we explain the range of the parameters and their default values of this analysis as they are summarized in Tables 1. When an experiment was conducted, we chose various values of one parameter within its range and assigned the default value to the other parameters. The min-sup parameter has a range from 20% to 80% with the default value = 50%, which is the median of the interval. The range of number of timed sequences parameter is chosen to be from 1 to 100,000 and its median value of 50,000 to be the default value. For the number of events per sequence, the default value is 25 because the range is from 5 to 50. The number of items in the last parameter range has been chosen to be from 1 to 10 items per event, thus, the default value is 5, which is the median.

5.3 Competing Algorithms

Since no existing algorithm can discover the timed sequential patterns and consider All-time Occurrences, we cannot compare Minits-AllOcc against any technique. We will compare it against MMinits-AllOcc.

5.4 Evaluation Metrics

The evaluation metrics include two measurements: (1) Execution Time (ET) of algorithms (Minits-AllOcc, and MMintis-AllOcc) (2) Number of Patterns (#patterns) that are generated by these algorithms.

5.5 Experimental Results

In this section, we present the performance of the two algorithms, Minits-AllOcc and MMinits-AllOcc, in terms of execution time (ET) and the number of discovered patterns (#patterns) for the real and synthetic datasets.

1. Accuracy

In order to validate that Minits-AllOcc always gives the same sequential patterns in terms of the numbers and contents excluding the temporal relation as those produced by PrefixSpan [17]. First, all temporal relations were removed from the patterns that were generated by Minits-AllOcc. Then, these patterns were compared to the patterns that were generated by PrefixSpan to make sure that each sequential pattern generated by PrefixSpan has a matching one generated by Minits-AllOcc and MMinits-AllOcc. For example, a sequential pattern X= < {a} {b} {a,b}> was generated by PrefixSpan and a timed sequential pattern Y= < {a} [2,5] {b} [3,7] {a,b}>, was generated by Minits-AllOcc and MMinits-AllOcc. We took away the temporal relations from Y and compared it with the pattern X. In case the order of at least one itemset was different, the pattern X was not matching the pattern Y. For instance, Z= <{b} [2,5]{a} [3,7] {b,a}> was not matching pattern X because the item <{b}> occurred before <{a}>. However, within the last itemset {a,b} the order does not matter because all the items appear at the same timestamp. At the end of this experiment, we found that the two algorithms: Minits-AllOcc and MMinits-AllOcc discovered the exact patterns that were produced by PrefixSpan. In other words, all algorithms produced the complete and correct set of sequential patterns.

2. Execution Time

The execution time was recorded starting from the moment that a dataset had been read to the moment that an algorithm produced the timed sequential patterns. Table 2 shows the average performance of the two algorithms: Minits-AllOcc and MMinits-AllOcc. The execution time (ET) of MMinits-AllOcc decreases by about 50% and 60% for T-Drive and synthetic datasets respectively compared to the execution time of Minits-AllOcc.

Table 1. Parameter list for the synthetic dataset

Parameter Name	Range of values	Default value
min_sup	20% - 80%	50%
# sequences	1-100,000	50,000
# events per sequence	1-50	25
# items per event	1-10	5

Table 2. Average Execution Time ET and #patterns

Datasets	Minits-AllOcc		MMintis-AllOcc	
	ET	# patt	ET	# patt
T-Drive	12.05 (hour)	126	5.97 (hour)	126
Synthetic data	27.319 (min)	3780	10.825 (min)	3780

3. Impact of Minimum Support

In these set of experiments, we compared execution time (ET) and the number of patterns (#patterns) for different values of minimum support threshold (min_sup) for both datasets: T-Drive and synthetic. From Fig. 9 (a) and Fig. 10 (a), we can see that when the minimum support increased, the execution time of all algorithms decreased. The reason is the algorithms generate fewer timed sequential patterns when the min-sup is high because the number of candidate sequences that satisfy the min-sup condition became fewer. With a large amount of data and a huge number of discovered timed sequential patterns, MMinits-AllOcc outperformed Minits-AllOcc as shown in Fig. 9(a) and Fig. 10(a). Therefore, using multicore CPUs is more useful when the size of the timed sequence database is huge.

The multicore CPU version was also efficient when we have low min-sup. As we observed from Fig. 10 (b), the ET of both Minits-AllOcc and MMinits-AllOcc were very close when the min-sup is greater than 60%. The reason is the number of candidate sequences, and thus the number of timed sequential patterns, was getting smaller, so most of the threads were idle. Therefore, MMinits-AllOcc did not need to use all the available threads and behaved almost like a single-core version Minits-AllOcc. Another observation was based on the number of timed sequential patterns that were generated by these algorithms. All algorithms discovered the same number of patterns; thus, their curves were overlapping in Fig. 9(b),10(b),10(d),11(b), and 11(d). When the min-sup increases, the number of timed sequential patterns decreased because the patterns that satisfy the min-sup condition became fewer. By increasing the threshold min_sup, the percentage of timed sequences in the timed sequence database that was supposed to contain a candidate sequence decreased as shown in Fig. 9(b) and Fig. 10(b).

4. Impact of the Number of Sequences in the Database

In these set of experiments, we compared the execution time (ET) and the number of discovered timed sequential patterns (#patterns) for different number of the timed sequences (#Seq). From Fig. 10(c), we can see that when the number of timed sequences increased; the execution times of all algorithms increased. The reason is that the algorithms needed more time to check the extra timed sequences that were added in the timed sequence database to decide if they contained a timed sequential pattern or not. We observed that number of timed sequential patterns, which were

generated by these algorithms, increased when the number of timed sequences as shown in Fig. 10(d), the number of timed sequential patterns that were discovered by the algorithms also increased because of the possibility of finding more patterns in the new timed sequences that satisfy the min-sup (50% as the default value) condition also increased. By increasing the number of timed sequences in the database, the algorithms needed to check if some new patterns can occur and did not exist in the old timed sequences. Next, it checked their support against the threshold (min-sup). It is possible the support of some old patterns in the database before new sequences were added did not satisfy the min-sup condition because they were not supported by a sufficient number of timed sequences but with a new timed sequence database, these patterns became to be timed sequential pattern. Thus, the number of newly discovered timed sequential patterns would increase. For example, if a database had 1000 sequences in the synthetic dataset, the number of timed sequential patterns was 3720, while the number of timed sequential patterns was 3780 when the timed sequence database had 10,000 timed sequences.

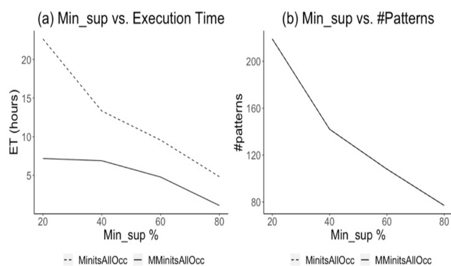

Fig. 9. Parameter study for T-Drive dataset

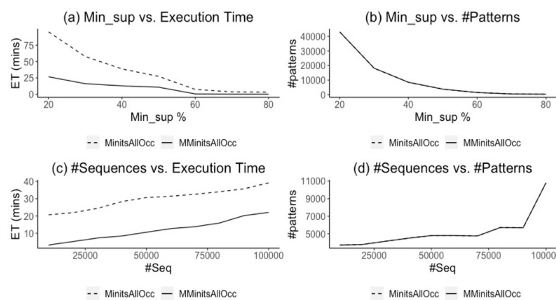

Fig. 10. Parameter study for synthetic dataset

5. Impact of the Number of Events per Sequence

Fig. 11(a) and (b) show the impact of the number of events (#Events) per timed sequence on the execution time (ET) and the number of discovered sequential patterns (#patterns). There was a strong relationship between the length of a timed sequence and the number of discovered patterns. Increasing the length of timed sequences (#Events) drove discovering more patterns because the algorithm can extend a pattern up to the length of the timed sequence. In other words, if we have a timed sequence that contains *n* events, we can discover a set of timed sequential patterns that their length varies from 1 to *n*. Subsequently, the required time of discovering those patterns would be increased.

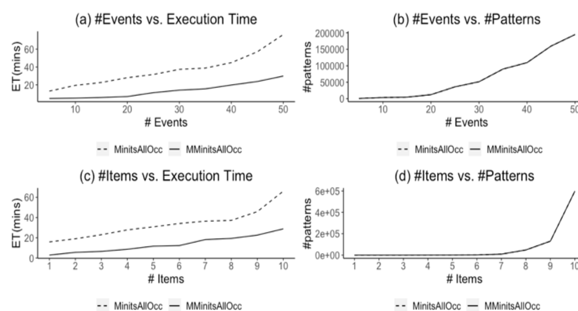

Fig. 11. Parameter study for synthetic dataset

6. Impact of the Number of Items per Event

In the last experiment, we increased the number of unique items in each event. That means may new items appear in timed sequence database TSDB that lead to detecting more new timed sequential patterns. When the number of items increases, the number of possible combinations between those items to generate candidates also increases. Thus, the number of patterns

increased, as shown in Fig. 11(d). Growing the length of events led to the growth of the number of candidates, which means the algorithms needed more time, as shown in Fig. 11(c), to check those events, generate candidates, and determine if they were timed sequential patterns and reported the temporal relations.

6 Conclusion and Future Work

In this paper, we presented an algorithm, called Minits-AllOcc, to discover timed sequential patterns TSP, which are sequential patterns that include the transition times between all possible occurrences in events across the timed sequence database TSDB. We implemented two versions of Minits-AllOcc: (1) Minits-AllOcc using single-core

CPUs, and (2) MMinits-AllOcc on multi-core CPUs. We conducted experiments to compare the accuracy and execution time of the algorithms. The experiments showed that the algorithms produced accurate patterns but MMinits-AllOcc outperformed Minits-AllOcc when the dataset is large in terms of the size of TSDB, length of timed sequences, or the number of items per event. For future work, we plan to improve Minits-AllOcc to be able to account for both very long timed sequences and Dynamic Timed Sequence Database DTSDB, such that the algorithm will be able to mine TSP without re-executing everything from scratch.

References

1. Agrawal, R., Srikant, R. Mining sequential patterns. In: Proceedings of the 11th IEEE International Conference, pp. 3-14, IEEE, Taiwan (1995).
2. AlZahrani, M. Y., & Mazarbhuiya, F. A. (2019). Discovering Constraint-based Sequential Patterns from Medical Datasets. International Journal of Recent Technology and Engineering (IJRTE), ISSN, 2277-3878.
3. Chen, Y. L., Chiang, M. C., & Ko, M. T. Discovering time-interval sequential patterns in sequence databases. Expert Systems with Applications, Vol.25, No.3, pp.343-354, (2003).
4. Fournier-Viger, P., Lin, C.W., Gomariz, A., Gueniche, T., Soltani, A., Deng, Z., Lam, H. T. (2016). The SPMF Open-Source Data Mining Library Version 2. Proc. 19th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2016) Part III, Springer LNCS 9853, Â pp. 36-40.
5. Fournier-Viger, P., Lin, J. C. W., Kiran, R. U., Koh, Y. S., & Thomas, R. A survey of sequential pattern mining. In Data Science and Pattern Recognition, pp. 54-77, (2017).
6. Gan, W., Lin, J. C. W., Fournier-Viger, P., Chao, H. C., & Yu, P. S. (2019). A survey of parallel sequential pattern mining. ACM Transactions on Knowledge Discovery from Data (TKDD), 13(3), 1-34.
7. Giannotti, F., Nanni, M., & Pedreschi, D. Efficient mining of temporally annotated sequences. In Proceedings of the 2006 SIAM International Conference on Data Mining, pp. 348-359, (2006).
8. Giannotti, F., Nanni, M., Pinelli, F., & Pedreschi, D. Trajectory pattern mining. In Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 330-339, (2007).
9. Han, Jiawei, Jian Pei, Behzad Mortazavi-Asl, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. "FreeSpan: frequent pattern-projected sequential pattern mining." In Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 355-359, (2000).
10. Hu, Y. H., Huang, T. C. K., Yang, H. R., & Chen, Y. L. (2009). On mining multi-time-interval sequential patterns. Data & Knowledge Engineering, 68(10), 1112-1127.
11. Huynh, B., Vo, B., & Snasel, V. (2017). An efficient method for mining frequent sequential patterns using multi-core processors. Applied Intelligence, 46(3), 703-716.
12. Jay, N., Herengt, G., Albuisson, E., Kohler, F., & Napoli, A. (2004). Sequential pattern mining and classification of patient path. MEDINFO, 1667.
13. Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. T-drive: driving directions based on taxi trajectories. In Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS '10, pages 99-108, New York, NY, USA,2010. ACM.
14. Jing Yuan, Yu Zheng, Xing Xie, and Guangzhong Sun. Driving with knowledge from the physical world. In The 17th ACM SIGKDD international conference on Knowledge Discovery and Data mining, KDD'11, New York, NY, USA, 2011. ACM
15. Karsoum, S., Gruenwald, L., Barrus, C., & Leal, E. (2019, December). Using Timed Sequential Patterns in the Transportation Industry. In 2019 IEEE International Conference on Big Data (Big Data) (pp. 3573-3582). IEEE.
16. Li, Huanhuan, Xiaofeng Zhou, and Chaojun Pan. "Study on GSP algorithm based on Hadoop." In Electronics Information and Emergency Communication (ICEIEC),5th IEEE International Conference, pp. 321-324, (2015).
17. Pei, Jian, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. "Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth." In icccn, p. 0215, (2001).
18. Pramono, Y. W. T. (2014, August). Anomaly-based intrusion detection and prevention system on website usage using rule-growth sequential pattern analysis: Case study: Statistics of Indonesia (bps) website. In 2014 International Conference of Advanced Informatics: Concept, Theory and Application (ICAICTA) (pp. 203-208). IEEE.
19. Srikant, R., Agrawal, R. Mining sequential patterns: Generalizations and performance improvements. In: International Conference on Extending Database Technology, pp.1-17, Springer, Berlin, Heidelberg (1996).
20. Wei, Yong-qing, Dong Liu, and Lin-shan Duan. Distributed PrefixSpan algorithm based on MapReduce. In IEEE Information Technology in Medicine and Education (ITME), International Symposium on, vol. 2, pp. 901-904, (2012).
21. Yang, H., Gruenwald, L., & Boulanger, M. A novel real-time framework for extracting patterns from trajectory data streams. In Proceedings of the 4th ACM SIGSPATIAL International Workshop on GeoStreaming, pp. 26-32, (2013).
22. Zaki, Mohammed J. SPADE: An efficient algorithm for mining frequent sequences. Machine learning 42, no. 1-2, (2001).