Journal of Autonomous Intelligence (2023) Volume 6 Issue 1
doi:10.32629/jai.v6i1.593

ORIGINAL RESEARCH ARTICLE

Mining timed sequential patterns: The Minits-AllOcc technique

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

[1] *University of Oklahoma, Norman, OK 73019, USA. E-mail: somayah.karsoum@ou.edu*
[2] *University of Minnesota Duluth, Duluth, MN 55812, USA.*

ABSTRACT

Sequential pattern mining is one of the data mining tasks used to find the subsequences in a sequence dataset that appear together in order based on time. Sequence data can be collected from devices, such as sensors, GPS, or satellites, and ordered based on timestamps, which are the times when they are generated/collected. Mining patterns in such data can be used to support many applications, including transportation recommendation systems, transportation safety, weather forecasting, and disease symptom analysis. Numerous techniques have been proposed to address the problem of how to mine subsequences in a sequence dataset; however, current traditional algorithms ignore the temporal information between the itemset in a sequential pattern. This information is essential in many situations. Though knowing that measurement Y occurs after measurement X is valuable, it is more valuable to know the estimated time before the appearance of measurement Y, for example, to schedule maintenance at the right time to prevent railway damage. Considering temporal relationship information for sequential patterns 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 occurrences of a pattern in the database. We also propose a parallel multi-core CPU version of this algorithm, called MMinits-AllOcc (Multi-core for MINIng Timed Sequential Pattern for All-time Occurrences), 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 multi-core CPU version of the algorithm is shown to outperform the single-core version on Big Data by 2.5X.

Keywords: Data Mining; Sequential Pattern Mining; Timed Sequential Patterns; Singe-core and Multi-core Processor

ARTICLE INFO

Received: 12 April, 2023
Accepted: 1 June, 2023
Available online:19 June, 2023

COPYRIGHT

Copyright © 2023 by author(s).
Journal of Autonomous Intelligence is published by Frontier Scientific Publishing. This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).
https://creativecommons.org/licenses/by-nc/4.0/

1. Introduction

Sequential pattern mining (SPM)[1] analyzes a sequence database, which contains sequences of events that are ordered based on the times when the events occurred or collected, called timestamps, to discover sequential patterns. These sequential patterns are those time-ordered events that frequently occur in the sequence database. SPM has been used in many real-life application areas such as transportation arrival time analysis, weather prediction[2,3], illness symptom pattern prediction[4], network intrusion detection[5], educational data mining[6], and customer shopping behaviors[1]. Knowing frequent sequential patterns, we can answer a question like "In which order does the measurements(s) for train aerodynamic phenomena frequently occur?" For example, high temperature and high wind speed could cause a train crash by warping tracks or overturning of the lightly loaded train[7,8]. Similarly, through transportation arrival time analysis, with sequential patterns discovered from a sequence database recording the movement of taxis, we may expect that the travel range of taxis that move from Broadway Street to Times Square in NYC

is 20 to 40 minutes. Also, in Oklahoma during the hurricane season, we can estimate the transition time range when a tornado hits Oklahoma City, Moore, and Norman in order. However, the existing works in SPM[9–11], tried to improve the efficiency of techniques to discover the frequent sequential patterns but discard the time dimension completely. The timestamps are used to order events within a sequential pattern, but the transition time between these events is not shown in the discovered sequential patterns. In many applications, it is important to know the time interval [min, max] events in a frequent sequential pattern discovered, which we call a timed sequential pattern. For example, with sequential patterns that contain temporal information about the transition time between signs, we can answer a question like "When will the next measurements of train aerodynamics occur?" Similarly, we may want to have a frequent timed sequential pattern that shows that *after a tornado hits Oklahoma City, within 10 to 15 minutes later, the tornado will hit Moore, and then within 3 to 5 minutes later, the tornado will hit Norman*. Knowing the temporal information (the time intervals of event occurrences) in frequent sequential patterns will help in preparing a safety plan to reduce damages and loss.

Sensor ID	Sequences
S1	< {5,T1, W3}, {12,T1, W2}>
S2	< {5,T1, W1}, {24,T1, W3}, {28,T1, W1}>
S3	< {6,T2, W3}, {19,T1, W2}>
S4	< {7,T1, W3}, {9,T1, W3}>

Figure 1. Sensors' historic weather information and discretize data.

As shown in **Figure 1**, we have the historic weather information (temperature and wind speed) of four sensors, each of which denoted as (S) with an ID as shown in the first column. The time is recorded with each measurement taken by each sensor as shown in the second column. Since sequential pattern mining algorithms do not deal with continuous data, we need to apply a partitioning technique to segment the data into classes that have similar features or fall within a same group. Therefore, we add an additional column next to each column that contains the equivalent class ID. For instance, the wind speed (W) has five levels[12] and each level refers to the damage that causes: 1) minimal ($74 \leq W < 95$), 2) moderate ($96 \leq W \leq 110$), 3) extensive ($111 \leq W \leq 129$), 4) extreme ($130 \leq W \leq 156$), and 5) catastrophic ($W \geq 157$). Thus, the first row in the last column has the wind speed class 3 because the value 112 belongs to the class 3 (minimal).

A timed event sequence that we want to discover is a sequence of frequently occurring measurements among sensors (or events) and typical transition times between these measurements (in terms of hours for the example case). The following is the format of a pattern that would be discovered in this study:

$$TSP = < \{T1, W3\}\ [30, 36]\ \{W5\} >$$

This TSP has two itemsets: itemset 1 consisting of two items T1 and W3, and itemset 2 consisting of item W5. Itemset 2 occurs within 30 to 36 hours after itemset 1. In our notations, all items enclosed within braces {} occur at the same time and constitute an itemset, and the square brackets [min, max] indicate the time duration to move from one itemset to the next. In this algorithm, the time duration represents the temporal relation [min, max]. Thus, the given example TSP shows that frequently when measurement has a temperature falling in the class 1 (T1) and a wind's speed falling in the class 3 (W3), then within 30 to 36 hours, the measurement will have a wind's speed in the class 5 (W5). If we apply traditional sequential pattern mining, this sequential pattern will only be < {T1, W3}, {W5} >, which does not include the transition time [30, 36].

Incorporating the temporal information in a sequential pattern raises additional challenges for mining compared to regular sequential pattern mining. First, while both sequential pattern mining and timed sequential pattern mining need to find out whether a pattern occurs in some tuples of a database, timed sequential pattern mining also needs to find out how many times the pattern occurs in each tuple to compute the temporal relationship between the itemsets in the pattern. Suppose we have a tuple that has all the measurements within 6 months and the following measurements occurring many times: low temperature followed by high wind speed after some time. Since the timed sequential pattern mining problem wants to know when the high wind speed occurs, it is not sufficient to find only the first position of this measurement and report the temporal relation. For example, from **Figure 2**, we can observe that the second sensor, S2, has the follow-

Sensor ID	Sequences
S1	< {5,T1, W3}, {12,T1, W2}>
S2	< {5,T1, W1}, {24,T1, W3}, {28,T1, W1}>
S3	< {6,T2, W3}, {19,T1, W2}>
S4	< {7,T1, W3}, {9,T1, W3}>

Figure 2. Sequence records.

ing measurements based on the timestamp order: the temperature from class 1 and the wind speed from class 1 {T1, W1}, followed by the temperature from class 1 and the wind speed from class 3 {T1, W3}, followed by the temperature from class 1 and the wind speed from class 1 {T1, W1}. The tuple for this sensor is: S2 = < {T1, W1}, {T1, W3}, {T1, W1} >. To find the temporal relation between the two measurements {T1} and {W1} (for the pattern denoted as < {T1} [ ] {W1} >), we need to do the following as shown in **Figure 3**:

1) Find the timestamp difference t1 between the first occurrence of T1 and the first occurrence of W1 (solid arrows).
2) Find the timestamp difference t2 between the second occurrence of T1 and the first occurrence of W1 (dotted arrows).
3) Find the timestamp difference t3 between the first occurrence of T1 and the second occurrence of W1 (dashed arrows).
4) Find the minimum timestamp difference and the maximum timestamp difference among t1, t2, and t3.
5) Produce the temporal relation as [min, max].

So, to find all possible occurrences of a pattern, the naïve method is to scan each tuple until the end in the database. However, a sequential pattern mining algorithm will stop checking the rest of a tuple in the database as soon as the pattern is found. In contrast, timed sequential pattern mining requires checking all the tuples in the database. First, it is necessary to consider all possible occurrences of the pattern and all the different timestamps of each occurrence and find the temporal relation. After the temporal relation is found for one sensor, we need

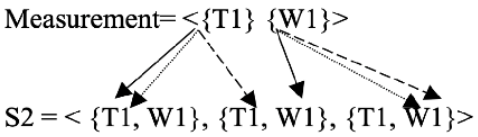

Figure 3. All possible occurrences of the measurement {T1} {W1} in S2.

to check the temporal relation for the same measurements among all sensors. The final interval [min, max] represents the minimum and maximum time difference among all sensors in the database.

This leads to the second challenge of timed sequential pattern mining, which updates the temporal relation between itemsets as soon as a pattern is found. When a timed sequential pattern is defined, it means that the ratio of tuples that contain this pattern is greater than or equal to a user-defined threshold. Then, when we want to extend that pattern to include more measurements; it does not mean that the pattern must appear at the same tuples because some tuples may not carry it anymore. Accordingly, the time relation is not valid anymore, and we need to update that relationship based on the new timestamps of the new tuples. Let us suppose that we have the timed sequence pattern < {T1, W3} [t1, t2] {T1} >. From **Figure 2**, we can observe that S1 and S4 have these measurements. So, t1 and t2 are calculated based on the timestamps associated with these measurements in these tuples. When the pattern is extended to be < {T1, W3} [t1, t2] {T1, W2} >, we can observe that the record of S4 does not carry this pattern and only S1 had these measurements. Therefore, t1 and t2 must be updated based on the timestamps associated with measurements. The brute force technique needs to scan the database again to update the temporal relation of the pattern. Thus, for every pattern, we need to scan the entire database many times to make sure that we have the correct temporal relations.

The contributions of this paper are the following:

1) The idea of incorporating transition time between itemsets in a sequential pattern indicates all possible time occurrences of the pattern within the whole timed sequence database. The time can be any descriptive statistic based on the user's preference, such as range and average.
2) The parallel implementation of the Minits-AllOcc algorithm can help when dealing with Big Data.
3) The extensive experiments compare the single-core algorithm against the multi-core algorithm on real and synthetic datasets.

The remainder of the paper is organized as fol-

lows. Section 2 reviews the definiation of the problem. Section 3 reviews the related work. Section 4 introduces and defines the timed sequential pattern mining problem. Section 5 explains how the algorithm works. The results of performance evaluations on different datasets are given in Section 6. Finally, Section 7 concludes the paper and discusses future work.

2. Problem definition

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], we define an **itemset I** as a set of **items**, such that I ⊆ X, where X = {x_1, x_2, ..., x_l} is a set of items in the database. 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 a **super-sequence** of A if there exists a set of integers, $1 \le j_1 < j_2 < \ldots < j_n \le m$, such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \ldots, a_n \subseteq b_{jn}$.

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 sub-sequence of s_i. Since our problem also considers the temporal data, 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 item set 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 ⊆ I ($1 \le i \le n$). 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 **Figure 4** is used as an illustrative example in this paper. For simplicity, we will use letters to refer to items that represent different properties of objects in the database (e.g., temperature and speed of wind), and integer numbers to refer to timestamps that 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 timestamp 12.

Definition 2. Given a sequence A = < {I_1}, {I_2}, …, {I_n} > and a timed sequence TS = < {e_1}, {e_2}, …, {e_m} >, the **All-time Occurrences** of A in TS in the timed sequence database TSDB is defined as an ordered list of indices $1 \le j_1 < j_2 < \ldots < j_n \le m$, such that: $I_1 \subseteq e_{j_1}.I, I_2 \subseteq e_{j_2}.I, \ldots I_n \subseteq e_{j_n}.I$. The **delta** Δ is 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 **Figure 4**. The indices of the events for the first occurrence of sequence A in TS4 are {e_1, e_2}, as shown by the solid arrow in **Figure 5**. 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 shown by the dotted arrow in **Figure 2**, has the indices of the evenets {e_1, e_4}. The delta 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 contain 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].

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} >

Figure 4. An example of timed sequence database.

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 an average of tran-

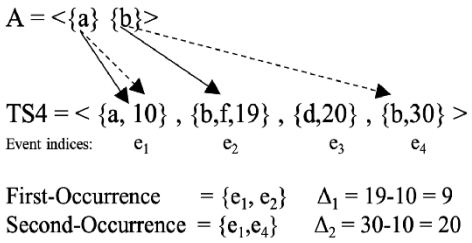

Figure 5. All-time Occurrence of A in TS4.

sition time or range, calculated based on the values of the delta . TSP is denoted as: TSP = < $\{I_0\}$ $[\tau_1]$ $\{I_1\}$ $[\tau_2]$ $\{I_2\}$, …, $[\tau_n]$ $\{I_n\}$ >. For brevity, in the rest of this paper, 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} >. The timed sequential patterns thus are sequential patterns that satisfy the min_sup condition and include the transition times between item sets.

3. Related works

One of the fundamental data mining techniques is pattern mining, which identifies rules to discover interesting correlations in the dataset. There are several approaches of pattern mining, including frequent itemset mining, high utility itemset mining, and sequential pattern mining. The survey paper of Kumar and Mohbey[13] introduced an overview of these unique approaches and discussed dealing with large-scale data from different aspects.

There are many techniques for mining the temporal patterns for frequent itemset[14–16], high utility[17–19], and sequential patterns[20–23]. Our work focuses on sequential pattern mining incorporating temporal information.

Sequential pattern mining was first introduced by Agrawal and Srikant[1], where three algorithms, AprioriSome, DynamicSome, and AprioriAll were proposed to discover sequential patterns. AprioriAll is the basis of many other efficient algorithms that have been proposed to improve its performance. Those algorithms[24] inspired to propose a technique to generate fewer candidates called GSP. Since all algorithms were based on the Apriori algorithm, they were classified as Apriori-based algorithms. Other algorithms, such as SPADE[11], adopted a vertical ID-list database format that reduces the number of database scans. In contrast, pattern-growth-based algorithms, such as FreeSpan[9] and PrefixSpan[10], use database projection, making them more efficient than other Apriori-based algorithms, mainly when they deal with an extensive database. These algorithms generate a smaller database for their next pass because the sequence database is projected into a set of smaller databases, and then sequential patterns in each of them are explored. Thus, they are more efficient. More literature reviews about the state-of-the-art sequential pattern mining algorithms can be found in the study of Fournier-Viger *et al*.[25].

Recently, with the existence of a large volume of data in many applications, several sequential pattern mining algorithms have been proposed to efficiently handle large databases consisting of vast amounts of sequences using different platforms. For example, Huynh *et al*.[26] used the multi-core processor architecture to implement pDBV-SPM to improve processing speed for mining sequential patterns. Ha-GSP[27] adopted the principles of GSP and implements them on the Hadoop platform for solving the limited computing capacity and inadequate performance with massive data of the traditional GSP. MR-PrefixSpan[28] used the MapReduce platform to implement the parallel version of PrefixSpan to mine sequential patterns on a large database. Also, Spark was utilized to implement two algorithms: GSP-S (GSP algorithm based on Spark) and PrefixSpan-S (PrefixSpan algorithm based on Spark)[29]. The proposed algorithms addressed the issues of high IO overhead and imbalanced load among the computing nodes. More literature reviews about the state-of-the-art parallel sequence mining algorithms are in the survey of Gan *et al*.[30].

The input for this data mining task is sequence data, in which each point in the dataset represents an observation at a particular time. A time-series dataset is an example of sequence data, which is a collection of integer values collected over a period of time. Trajectory is also a sequence of spatial points ordered by timestamps, which capture how an object behaves through various temporal activities[31]. Another example of this type of data is biological sequences. They are nucleotide or amino acid sequences analyzed and studied for use in bioinfor-

matics and contemporary biology. Each different type of data has its research issues; however, in this paper, we consider the general common research issues.

Because finding frequent itemsets in the association rule mining tasks discards the ordering of items, some techniques such as that in the study of Patnaik *et al*.[32] take advantage of sorting items based on the timestamp. They discovered different patterns that represent the different orderings of the items. For example, the general episode is a sequence with objects A, B, and C, where A must occur first, but B and C can occur in any order. However, the serial episode is a sequence with objects A, B, and C, where A must occur first, then B, and then C. However, the time between itemsets is still discarded, and they use the time as a gap constraint between itemsets in an episode. So, an expiry constraint TX is another input besides the sequence database and min_sup threshold. TX is an additional control with the support threshold, which specifies that the appearance of symbols in an episode occurs no further than TX time units apart from each other. Some techniques were proposed to specify some timing constraints, such as the time gaps between adjacent itemsets in sequential patterns. For example, Chen *et al*.[33] modified the Apriori[1] and PrefixSpan[10] algorithms to discover the time-interval sequential patterns that satisfy the interval duration boundaries. The I-PrefixSpan algorithm in the study of Chen *et al*.[33] has another input called a set of time-intervals TI, where each time-interval has a range. Hu *et al*.[34] extended that work and proposed two algorithms: MI-Apriori and MI-Prefix. The time intervals incorporated in the patterns reveal the time between all pairs of items in a pattern; these patterns are called multi-time-interval sequential patterns. A list of intervals (t_{i3}, t_{i2}, t_{i1}) before item d in a pattern like < a, (t_{i1}), b, (t_{i2}, t_{i1}), c, (t_{i3}, t_{i2}, t_{i1}), d > means the intervals between items a, b, and c and item d are t_{i3}, t_{i2} and t_{i1}, respectively. In educational data mining, a ti-pattern model[6] is built based on the I-PrefixSpan algorithm to consider the time between students' activities. So, again, the inputs of this model are a temporal sequence database and a set of time-intervals (I_s, I_{mn}, I_h, I_d, I_w, I_{mt}), which refer to seconds, minutes, hours, days, weeks, and months. For example, one-time intervals of I_h mean that the model will find the activities of students with a gap value between one hour and one day. After the model is applied to a group of students who enrolled in mathematics and computer science program in the Learning Management System, one of the time-interval patterns (ti-pattern) was found: < Lab $_{\{1, 3\}}$ Ih Lab $_{\{2, 3\}}$ Ih Lab $_{\{2, 3, 4\}}$ >, where Lab $_{\{1, 3\}}$ means either Lab1 or Lab_3. The experts can observe that some students work sequentially on several exercise sheets from this pattern. Since the students spend this gab, it means that the students dig deep into their work. Also, AlZahrani and Mazarbhuiya[35] extracted the sequential patterns of diseases from a medical dataset within user-specified time intervals. CAI-PrefixSpan[12] is proposed to apply the confident condition from association rules besides the support condition to filter the timed sequential patterns. The advantage of this is that the decision-makers can be confident about the possibility of an event happening within a certain time interval.

The drawback of these methods is that their results will miss some frequent patterns that do not fulfil the time range constraint. To decide if a pattern is common, two conditions must be satisfied: the support of the pattern must be greater than or equal to the min_sup, and the time ranges between the itemsets in the pattern must lie within the defined time intervals. Therefore, if a pattern fulfils the first condition, which means it is common but does not fulfil the second condition, the algorithm will not report it.

Giannotti *et al*.[20] incorporated the temporal dimension in the sequential pattern by defining temporally annotated sequences (TAS), and Giannotti *et al*.[36] 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 works for one-dimensional data. Also, the times between events in a trajectory pattern are strict, which does not consider the variety of the traveling time spent between locations by using different transportation modes, for example. Yang *et al*.[21] relaxes the travel time so that it is a realistic range for traveling time. The algorithm still cannot deal with multidimensional data because it deals with only locations in trajectory data. Also, all the previous techniques do not consider all possible occurrences of a pattern in an individual sequence in a database, which means

the temporal relations are calculated based only on the first occurrence of a pattern. The issue of calculating the time intervals of the first occurrence of a pattern and ignoring other occurrences is addressed in the study of Karsoum *et al*.[23]. However, this approach is beneficial for only a few applications. For example, if a developer wants to evaluate the ease of use of a navigation system, the time of moving from A to B is tested when the users visit those locations for the first time. In contrast, in other applications, such as the transportation safety application described in Section 1 above, we must consider all possible occurrences to provide accurate time intervals.

There also exist works that consider other issues related to time-interval sequential patterns. FARPAMp (Fast Robust Pattern Mining with information about prior uncertainty)[22] can deal with timestamp uncertainties. This issue may occur if two events A and B happen during a time interval that can be overlap. This leads to the possibility of event A appearing before event B or vice versa. So, the approach is focused on using time points instead of intervals and fitting probabilistic models for the errors in the timestamps around these time points. It is an interesting research issue; however, this is outside the scope of our research. We are addressing the issues of finding all possible occurrences of timed sequential patterns and producing the most updated temporal relations between itemsets in the discovered patterns.

Besides sequential pattern mining, several other pattern mining problems have been proposed that can deal with temporal data. For example, Episode mining aims to discover frequent episodes in a single sequence, rather than a set of sequences, within a time window set by the user[37,38]. Also, periodic patterns are finding patterns that appear frequently and periodically in a single sequence based on period lengths. The period lengths of a pattern are the time elapsed between any two consecutive occurrences of the pattern[39,40]. In this paper, we concentrated on the sequential pattern and how we include the time-interval explicitly between the itemsets.

Our work fundamentally differs from the previous techniques in the following aspects. First, we focus on sequential pattern mining for any sequence data. Second, the inputs of our approach are the timed sequence database and minimum threshold; the output is a complete set of timed sequential patterns with a time interval between each item set. Third, the time interval is not a user-defined parameter but is derived from the database's timestamps.

To the best of our knowledge, there is no existing algorithm that can find the complete set of timed sequential patterns, each of which includes the itemsets that occur in time order and the transition times between them.

4. The proposed algorithm: Minutes-AllOcc

We propose an algorithm called Minits-AllOcc to discover the complete set of timed sequential patterns, which are already frequent candidates, from a timed sequence database. We have the following subsections that describe the algorithm: Section 4.1 introduces the core data structure of the algorithm; Section 4.2 gives a brief overview before the details of the algorithm are explained step by step in Section 4.3; Section 4.4 analyzes the time complexity of the algorithm, and Section 4.5 proposes enhancements to improve the efficiency of the algorithm.

4.1 Occurrence tree (O-tree)

A data structure called the occurrence tree (O-tree) is proposed to represent all possible occurrences of a pattern in a particular timed sequence in TSDB. This tree is the essence of the algorithm because it helps generate timed sequence patterns without scanning the timed sequence database many times. In the tree, the timed sequence ID (TSID) is stored as the root. The rest of the nodes stores an event ID eID and its timestamp eID.t. A node can have multiple parent nodes and multiple child nodes. The information associated with the link between a parent node and a child node represents the difference Δ between the timestamps of the two nodes: parent and its child. The structure of the tree is shown in **Figure 6**. For example, when the TS3 in **Figure 4** is scanned, three occurrence trees for items, a, b, and d are created from the timed sequences < {a, 2}, < {a, b, 19}, {d, 25} >. Since the candidate sequence < {a} > appears twice in TS3, its O-tree in **Figure 7** has two nodes connected to the root. The first one represents the first occurrence at the first event e_1 with its timestamp, and the second represents the second occurrence at the second event e_2 with its timestamp. However, the sequence

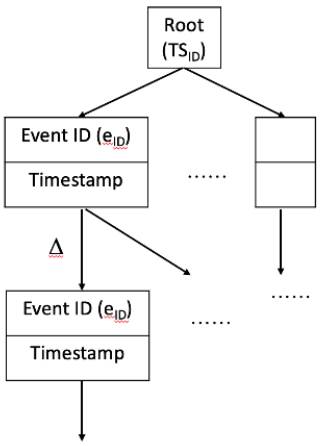

Figure 6. Occurrence tree data structure.

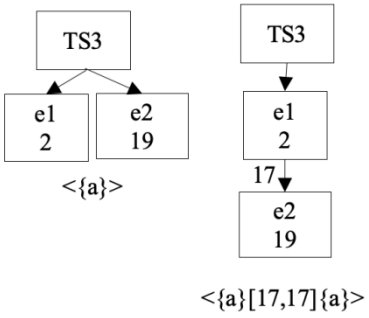

Figure 7. An O-tree for the sequences < {a} > and < {a} {a} > in TS3.

< {a} {a} > appears once in TS3 that has two nodes too, but one is connected to the root and the other is connected to the other node via a link Δ. The link holds the difference between the parent and child timestamps 19 – 2 = 17.

Since each sequence has an O-tree for each timed sequence in TSDB that contains 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 sequence A in TSDB. **Figure 8** demonstrates the forest of four sequences, < {a} >, < {b} >, < {a} [9, 20] {b} >, and < {a, b}>. Each forest is surrounded by a dotted rectangle, which has a group of O-trees that indicates all time occurrences of a sequence in TSDB.

4.2 Overview

Given a TSDB and a min_sup threshold, the main goal of Minits-AllOcc is to find the complete set of the timed sequential patterns in the TSDB such that each pattern's support is greater than or

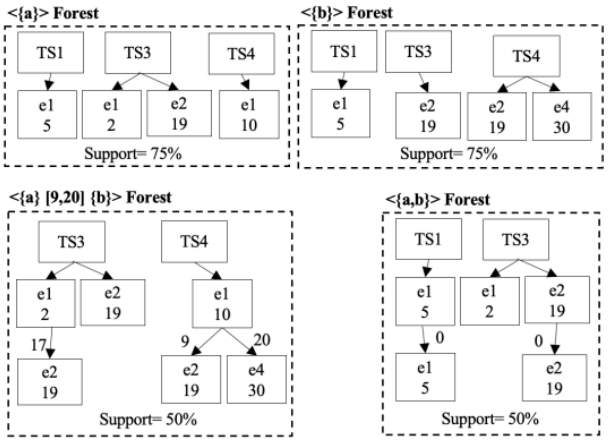

Figure 8. Merging O-trees of < {a} > and < {b} > to generate < {a, b} >-forest and <{a} [9, 20] {b} >-forest.

equal to the min_sup threshold. To achieve this goal, Minits-AllOcc utilizes the forests to store all the required information from the TSDB and uses them to mine the patterns without having to scan the TSDB many times. The flowchart in **Figure 9** provides an idea of the algorithm's general steps. More details about these steps are explained bellow.

1) Scan TSDB to build an Ij-forest for each distinct item I_j.

2) Find frequent 1-items by counting the number of O-trees in each forest, compare it against the min_sup threshold, and remove the infrequent 1-items.

3) Merge all O-trees with the same TSID (root node) from different forests to build a new forest for a candidate sequence. It should be noted that there are two relations between itemsets considered while merging the steps: *event-relation and sequence-relation*, which are defined as follows:

Definition 5. Given two itemsets X and Y, it is said that X and Y have an **event-relation** e-relation between them, denoted as < {X, Y} > if X and Y occur in the same event. For example, assume that we have the following timed sequential pattern = < {High temperature, High wind speed} [2, 3] {low temperature} >. It means that the measurement has both high temperature and high wind speed simultaneously, and after 2 to 3 hours, the measurement has a low temperature.

Definition 6. Given two itemsets X and Y, it is said that 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 Y. For example, suppose that we have the following timed sequential pattern

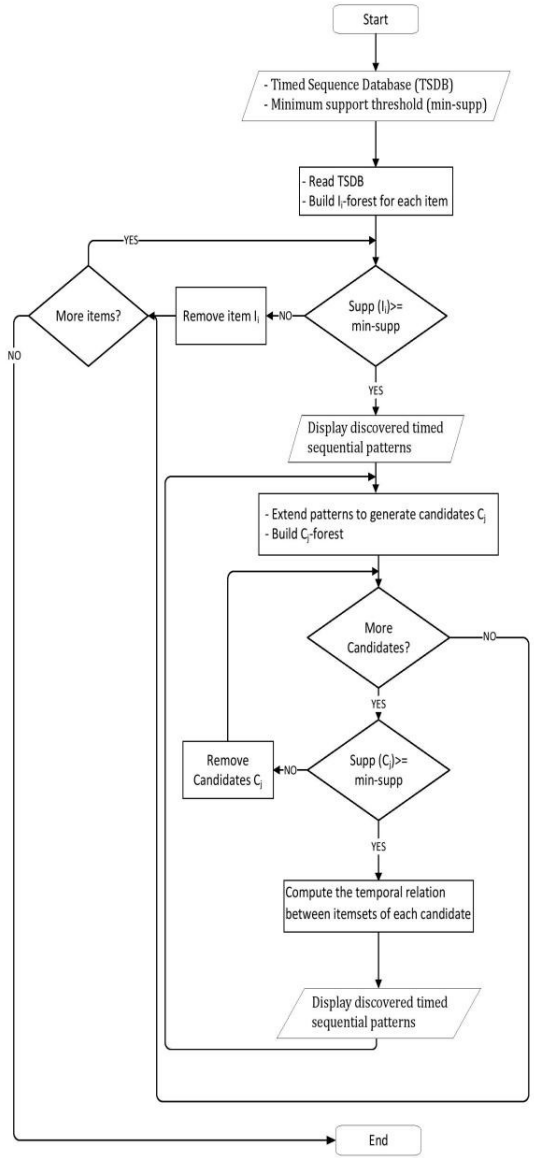

Figure 9. Flowchart of Minits-AllOcc algorithm.

= < {High temperature} [4, 6] {High wind speed} [2, 3] {low temperature} >, it means that the measurement has only a high temperature. Then after 4 to 6 hours, the measurement has the high wind speed. Later, after 2 to 3 hours, the measurement has a low temperature.

4) Count the number of O-trees in each forest, compute the support, and compare it against the min_sup threshold to find the sequential patterns among candidate sequences. By performing step 4, Minits-AllOcc avoids scanning the whole TSDB for each candidate to calculate its support.

5) Compute the temporal relation of the suffix (the new appending part of the pattern) if the candidate sequence is frequent. Then, update the temporal relation of the prefix (the previous part of the pattern) and generate a timed sequential pattern.

6) Repeat steps 3, 4, and 5 until the algorithm cannot identify any new timed sequential pattern. The pseudo-code of Minits-All Occ is presented in **Figure 10**.

4.3 The details of the Minits-AllOcc algorithm

This section describes the five steps presented in the previous Section 4.2 in detail using the running example shown in **Figure 4**. The algorithm scans the TSDB tuple by tuple and builds the associated forest for each item by adding the occurrence trees O-tree (lines 1–19). As shown in **Figure 7**, for example, after the algorithm finishes scanning the TSDB, the < {a} >-forest has three O-trees because the sequence < {a} > appears in three timed sequences: TS1, TS3, and TS4. Each O-tree captures all occurrences with their timestamps of an item and in a particular TS. Thus, in TS1, we have one node that shows the item a appears in the first event in TS1, and its timestamp is 5. To know the support of distinct items, the algorithm counts the number of O-trees in each forest and compares it against the min_sup threshold. If the support of a forest, which also represents of distinct item's support, is less than the threshold, the forest is removed (lines 20–27). The two sequences < {e} > and < {f} > are not frequent because their forests have only one tree, which means they appear only in one TS; therefore, their support is 25%. Consequently, two sets are formatted: TSP and 1-TSP. The first set of TSP contains the complete, timed sequential patterns. It will be updated periodically as a new timed sequential pattern is discovered. The second set is 1-TSP, which contains only the timed sequential patterns of length 1, which will be used as a seed set to extend the patterns in further steps. Both sets TSP and 1-TSP have these values {< {a}, {b}, {d}, {g} >} (lines 24–25). The next step is generating candidates by merging the O-trees of all 1-timed sequential patterns by calling the function find-TSPs (line 30). The mechanism of merging trees is as follows: if the relationship is an s-relation, the appended node must have an event ID e_i that is greater than the event ID in the parent node (i.e., comparing the event IDs in the two nodes) (line 57). Then, the link holds the difference between the timestamps of the parent and their child (line 59). In contrast, if the relationship is an e-relation, the appended node must have the same event ID ei as its parent (line 53). For

Algorithm 1: Minits All-Occ

Input: Timed sequence database (S), minimum support threshold (min_sup)

Output: Timed Sequential Patterns set (TSP-set) that contains all Timed Sequential Patterns TSP

```
    //Build a forest for distinct items (1-candidate sequence) and calculate the support
 1  for each Timed sequence Ts_i in S
 2      for each event e_j in Ts_i
 3          skip first item I_0 in e_j // it always represents the timestamp
 4          for each Item I_k in e_j
 5              if ((I_k) does not appear before)
 6                  create <{I_k}> forest
 7                  build I_k Occurrence-tree inside the <{I_k}> forest
 8                  NumOfOccurrenceTree+=1
 9              else
10                  if (Ts_i exist in the forest)
11                      Update the Occurrence-tree by adding the new node
12                  else
13                      build I_k Occurrence-tree inside the forest
14                      NumOfOccurrenceTree+=1
15                  end if
16              end if
17          end for
18      end for
19  end for
    //Remove infrequent 1-candidate sequences
20  for each I_k-forest
21          if (((NumOfOccurrenceTree/ NumOfTs) *100) < min_sup)
22              remove I_k-forest
23          else
24              add <{I_k}> into TSP-set
25              add <{I_k}> into 1-TSP
26          end if
27  end for
    //Extend 1-Timed Sequential Patterns TSP
28  for each pattern p_m in 1-TSP
29      for each pattern p_n in 1-TSP
30              Find-TSP (p_m, p_n, 1-TSP)
31      end for
32  end for
    //Perform Find_TSP () function recursively to discover all k-TSP, where k >1
33  function Find-TSP (prefix, suffix, suffixList)
34      prefix = Merge_Trees (prefix. forest, suffix. forest)
35          if (prefix! = null)
36              for each suffix in suffixList
37                  Find-TSP (prefix, suffix, suffixList)
38              end for
39          else
40              return
41          end if
42  end function
    //Merge trees to generate candidates, find TSP, and calculate temporal relation
43  function Merge-Trees (prefix. forest, suffix. forest)
44      number_of_merging-trees = 0
45      new_candidate_sequence = < prefix ∪ Suffix >
46      create forest for new_candidate_Sequence
47      for each OccurrenceTree pt_i in the prefix. forest
48          for each OccurrenceTree st_j in the suffix. forest
49              if (pt_i. TSID == st_j. TSID)
50                  for each leaf node N_p in pt_i
51                      for each leaf node N_s in st_j
52                          Add pt_i to new_candidate_sequence forest
53                          if (st_j. eventID == pt_i.eventID)
54                              Append N_s to pt_i
```

```
55                    Δ = 0
56                    number_of_merging-trees += 1
57                   else if (st_j. eventID > pt_i. eventID)
58                    Append N_s to pt_i
59                    Δ = st_j. timeStamp - pt_i. timeStamp
60                    number_of_merging-trees += 1
61                   else
62                    Remove N_p from pt_i
63                   end if
64                  end for
65                end for
66              end if
67             end for
68          end for
69          if (((number_of_merging_trees/ NumOfTS) *100) > = min_sup)
70             for each itemset I_n in new_candidate_sequence except the last itemset
71                Go to the level n+1 in the new_candidate_sequence.forest
72                Find min of Δ
73                Find max of Δ
74                Add [min, max] after I_n
75             end for
76              Add new_candidate_sequence int TSP
77              Prefix = new_candidate_sequence
78              return prefix
79           else
80              return null
81           end if
82    end function
```

Figure 10. Pseudo-code of the Minits-AllOcc algorithm.

instance, the forests of the two candidates < {a} [ ] {b} >, which represents an s-relation, and < {a, b} >, which represents an e-relation, are shown in **Figure 8**. The first < {a} [ ] {b} >-forest has two O-trees that are generated by combining the < {a} >-forest and the < {b} > =-forest. Even though both the 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 is removed from the < {a} [ ] {b} >-forest. In contrast, the node e_2 from the < {b} >-forest is attached to the node e_1 from the < {a} >-forest and the Δ is calculated between those nodes, which is 19 − 2 = 17. However, the node that has e_2 from the < {a} >-forest does not connect to any node. Since the algorithm is looking for all possible occurrences of sequence < {a} [ ] {b} >, the node e_1 in TS4 is connected to the two nodes, which have the event IDs e_2 and e_4, from the < {b} > O-tree, and each link between the parent node e_1 and child node e_2 and child node e_4 carries the difference between the timestamps of the two connected nodes. Because in this example, we consider a temporal relation as a range [min, max], the algorithm chooses the minimum and maximum values among all the O-trees in the < {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 the < {b} >-forest. The difference between the s-relation case and the e-relation case when we merge the trees 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% (lines 69–76). We calculated the support using the below formula:

$$Support(candidatesequence) = \frac{O_{trees} \in theforest}{timedsequences \in TSDB * 100}$$

These two timed sequential patterns are added into TSP = { < {a} >, < {b} >, < {d} >, < {g} >, < {a} [9, 20] {b} >, < {a, b} >}. The algorithm repeats the same steps, by calling function findTSPs recursively in line 37, to extend the pattern by merging O-trees, generating candidates, finding TSPs, and computing temporal relations until no more TSPs can be found.

As shown in **Figure 11**, pattern < {a} [9, 17] {b} [1, 6] {d} > result from merging between < {a} [9, 20] {b} >-forest and < {d} >-forest. The forest consists of only the O-trees that representing the candidate, then the support is calculated. Since the support is 50%, the time between the prefix < {a} [ ] {b} > and suffix < {d} > is calculated as defined before (the range [min, max]). The TSP set is updated to be {< {a} >, < {b} >, < {d} >, < {g} >, < {a} [9, 20] {b} >, < {a, b} >, <{a} [9, 17] {b} [1, 6] {d} >}. As it is noted, the temporal relation between item sets {a} and {b} in the two patterns < {a} [9, 20] {b} > and < {a} [9, 17] {b} [ ]{d} > changed.

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} >, < {b} >, < {d} >, < {g} >, < {a} [9, 20] {b} >, < {a} [6, 23] {d} >, < {b} [1, 7] {d} >, < {a, b} [6, 7] {d} >, < {a} [9, 17] {b} [1, 6] {d } >}.

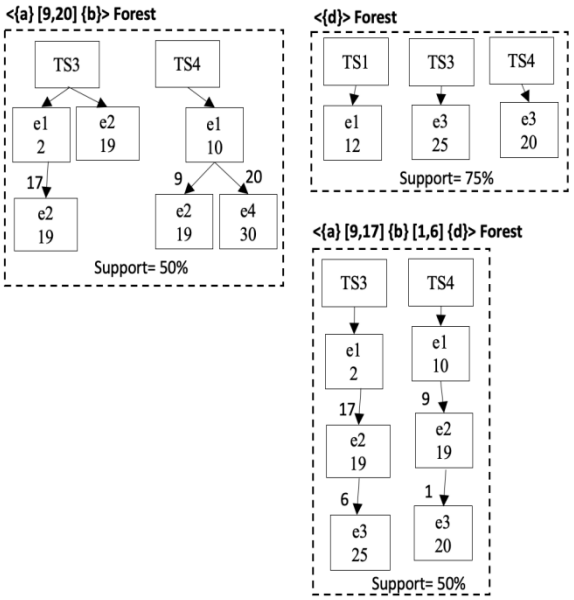

Figure 11. Merging < {a} [9, 20] {b} >-forest and < {d} > to generate < {a} [9, 17] {b} [1, 6] {d} >-forest.

4.4 Analysis of Minits-AllOcc

In this subsection, we discuss the worst-case time complexity of the Minits-AllOcc algorithm. We have:

- S(m), where |S| = the number of timed sequences TS in TSDB.
- E(r), where |E| = the maximum number of events in a timed sequence.
- I(c), where |I| = the maximum number of items in an event.
- G(s), where |G| = the number of singleton items in TSDB.
- N, where |N| = the number of all possible candidates.

We start with the first part of the algorithm that needs to check each Timed Sequence S in TSDB, each event E inside that S, and each item inside that E to build the forest (lines 1–19), which cost O(S*E*I). If it is the first time to read an item, that means its forest does not exist. So, we need to build it from scratch and start counting the number of O-trees inside that forest. Otherwise, we just need to update the forest by adding the new O-tree into an existing forest and update the number of O-tress inside that forest, which cost O(log N). Therefore, the total amount of work performed by the end of (line 19) is O(S*E*I*log N).

To keep only frequent candidate sequences and remove infrequent ones, the algorithm calculates the support for each forest (lines 20–27) and adds the frequent candidate sequences into the TSP-set. So, the total amount of work performed by the end of (line 27) is O(G+log I) because the number of forests is equal to the number of singleton (distinct) items in TSDB and removing O-trees for any infrequent sequence is log I .

After that, the algorithm extends the patterns to generate more candidates by calling the function Find_TSP( ) (line 30). The function tries to combine each item in the 1-TSP set to generate 2-length candidate sequences, for example at the first call. The prefix is the previous (k − 1)-timed sequential patterns, and the suffix is an item from the 1-TSP set. The function will append the suffix to the prefix and check the support of the new candidate sequence to decide if it can be considered as a timed sequential pattern or not. First, we need to find the time complexity of internal functions; then, we will compute the time complexity of the whole Find_TSP( ) function. The Find_TSP( ) function calls another function called Merge_Trees( ) (line 43), sends the forest of the prefix (previous timed sequential pattern), and the 1-TSP to build the forest for each new candidate sequence considering the different types of relationships, either it is an s-relation or e-relation. In line 47, the function picks an occurrence tree pt from the forest of the prefix and compares it with all occurrence trees st for each 1-length timed sequential pattern (line 48), which cost S*G.

If two occurrence trees with the same root TSID have been found, the function checks if the event ID of each leaf node from both trees is the same ID or the event ID in the suffix node is greater than the ID in the prefix node (lines 53 and 57), which needs to check all events in both tress E^2. Also, this function updates the content of the forest by adding appropriate O-trees and calculating the differences Δ between nodes if the relation type is s-relation. After that, in line 69, each candidate's support is calculated by counting the number of trees in its forest. If the candidate is frequent, then after each itemset, the temporal relation is inserted, which cost E. The total amount of work done by Merge_Trees( ) is $O(\log N*S*G*E^3)$.

Recursively, for each frequent 1-item in the suffix list, in which the time complexity is O(G), the function Find_TSP( ) is called (line 37) until no more candidates can be generated. In the worst case, the function is re-called until the length of a candidate is equal to the length of the longest timed sequence in TSDB, so it is O(E). Thus, the total amount of work done by Find_TSP( ), ended by line 42, including the work done by nested function Merge_Trees( ), is $O(\log N*S*G^2*E^3)$. Because we are considering all possible combinations between any $(k - 1)$ sequential patterns, where $k \geq 2$, and 1-sequential patterns, the algorithm returns to line 30 and tries another combination between two items in the 1-TSP set. Thus, besides the time complexity of calling Find_TSP( ), the algorithm combines all items, so O(G). The total amount of work done by the end of line 32 is $O(\log N*S*G^3*E^3)$.

The work done by this algorithm for each subsection is $O(S*E*I*\log N) + O(G+\log I) + O(\log N*S*G^3*E^3)$. We conclude that the overall worst-case time complexity of this algorithm is $O(\log N*S*G^3*E^3)$.

4.5 The proposed enhancement

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

4.5.1 Pruning the forests

This technique defines 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. **Figure 12** represents the idea by marking the deleted branch of O-trees with a cross symbol. For example, the O-tree that has a TS3 root that results from 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.

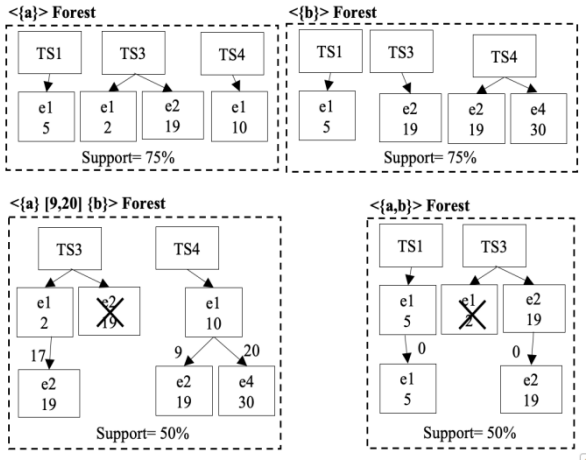

Figure 12. Pruning the original < {a} [9,20] {b} >-forest and < {a, b} >-forest in **Figure 11**.

4.5.2 Using frequency matrix

With this technique, we avoid generating unnecessary candidates, thereby reducing the number of forests. For example, the algorithm uses the 1-sequence-forests to generate 2-sequence candidates, then keeps frequent candidates 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 frequent candidates. For example, the frequency matrix of < {a} > pattern is shown in **Figure 13**. The two different relations, events, and sequences (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 **Figure 7**. For an s-relation, there are two O-trees (TS3 and TS4) in which the < {a} > and < {b} > occur at 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 an 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%

Figure 13. Frequency matrix for < {a} >.

4.5.3 Using multi-core CPUs

Another enhancement is using multi-core CPUs for implementing Minits-AllOcc, which we call MMinits-AllOcc. The independent jobs that can be done at the same time are finding all possible candidates, merging O-trees for those candidates, and deciding if they are frequent or not. A queue holds all jobs. As soon as one thread becomes idle, the next job in the queue is assigned to it and this reduces the execution time of the algorithm. For instance, in the beginning, the algorithm scans the TSDB to build the forest for each item and finds that < {a} >, and < {b} > are frequent. 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} >. Then, it starts with the pattern < {b} > and does the same thing. With the multi-core version, the algorithm inserts patterns < {a} > and < {b} > into the queue, as shown in **Figure 14**, and works on generating their candidates at the same time. Then, the candidates, < {a} [ ] {a} >, < {a} [ ] {b} >, etc., will be inserted into the queue to let any idle threads work on calculating their supports and report any of them as a time-sequential pattern. If one of these threads is done, then the pattern is extended by finding other candidates, < {a} [ ] {a} [ ] {a} >, < {a} [ ] {b} [ ] {b}, etc., and then inserting them into the queue. Those candidates wait to be assigned to an idle thread again. This process is kept going until no more jobs remain in the queue.

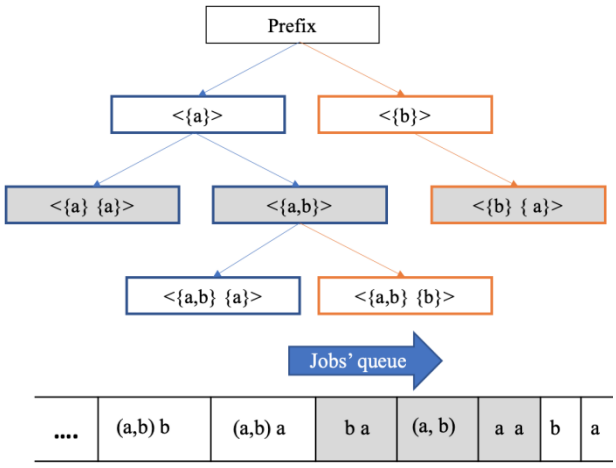

Figure 14. Multi-core implementation.

5. Performance analysis

In this section, we describe the environment of experiments and report the evaluation results of testing the algorithms that are implemented in single-core CPUs (Minits-AllOcc) and multi-core CPUs (MMinits-AllOcc). Different parameters are considered when these experiments are conducted on real and synthetic datasets. After running many experiments, we have found that MMinits-AllOcc on a multi-core performed Minits-AllOcc on a single-core.

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 two real-life, T-Drive[41,42] and Oklahoma Mesonet[2,3], and synthetic datasets. The first real dataset T-Drive is a collection of trajectories gathered by Microsoft Research Asia after tracking the movements of 10,357 taxis in Beijing, China for one day. The dataset contains the following attributes: user ID, timestamp, latitude, and longitude, as shown in **Figure 15**. For example, Taxi 1 has a sequence that contains many events to represent its movements. An event (2008-10-23 02:53:04, 39.93, 116.31) refers to timestamp, latitude, and longitude, respectively. Since the sequential pattern mining algorithm cannot deal with continuous data, we discretized the data first by using a density-based

clustering algorithm called Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [43], and the results of the discretized sequences are shown in **Figure 16**. Taxi 1 has a sequence that contains events in terms of clusters ID. For instance, event (2008-10-23 02:53:04, C1) refers to timestamp, and cluster Id, respectively. DBSCAN generates several clusters that contain the close points and replaces the latitude and longitude of a point with a cluster ID (Ci). For more details, we refer the readers to the work by Karsoum *et al*.[42]. The second real data set from Oklahoma Mesonet is a world-class network of environmental interventions by a group of scientists from the University of Oklahoma (UO) and Oklahoma State University (OSU) for weather monitoring stations. This network was established on January 1, 1994 and consists of 120 stations covering each of Oklahoma's 77 counties. The measurements are packaged into "observations" every 5 minutes, then the observations are transmitted to a central facility every 5 minutes, 24 hours per day year-round.

Taxi ID	Trajectory sequence
1	< (2008-10-23 02:53:04, 39.93, 116.31),....., (2008-10-23 11:11:12, 40.00, 116.32) >
2	< (2008-10-23 12:45:23, 39.92, 116.33),....., (2008-10-23 16:44:22, 39.93, 116.34) >
3	…
…	…

Figure 15. Sequential database for the T-Drive dataset before discretization by DBSCAN.

Taxi ID	Trajectory sequence
1	< (2008-10-23 02:53:04, C1),....., (2008-10-23 11:11:12, C2) >
2	< (2008-10-23 12:45:23, C1),....., (2008-10-23 16:44:22, C4) >
3	…
…	…

Figure 16. Sequential database for the T-Drive dataset after discretization by DBSCAN.

The dataset contains the following attributes: county ID, timestamp, air temperature, rainfall, wind, and moisture-humidity, as shown in **Figure 17**. We discretized the data first by using well-known scales in Meteorology.

Station ID	Weather sequence
1	< (2014-8-23 02:53:04, 17.2, 0.25,970, 206),....., (2014-8-23 11:11:12, 17.1,0.00,970.05,191) ..... >
2	< (2008-10-23 12:45:23, 14.2,0.00,969.91,7),....., (2008-10-23 16:44:22,12.9,0.02,690.99,4) .... >
3	…
…	…

Figure 17. Sequential database for the Oklahoma Mesonet dataset before discretization.

For air temperature, the index heat[43] is used to have the nine categories based on the temperature degree intervals in Fahrenheit: T1 (extremely hot) [>54], T2 (very hot) [53, 46], T3 (hot) [46, 39], T4 (very warm) [38, 32], T5 (warm) [31, 26], T6 (cold) [25, 0], T7 (very cold) [0, −10], T8 (bitter cold) [−11, −29], and 9 (extreme cold) [>−30]. The recurrence interval[44] is used to categorize the rainfall based on the probability that the given event will be matched or exceeded in any given year. For example, there is a 1 in 50% chance that 6.60 inches of rain will fall in X County in a 24-hour period during any given year. The classes are: R1 (1 year) [1.16–1.36], R2 (2 years) [1.37–1.69], R3 (5 years) [1.70–1.98], R4 (10 years) [1.99–2.36], R5 (25 years) [2.37–2.64], R6 (50 years) [2.65–2.90], and R7 (100 years) [2.90–3.15]. For wind, the Beaufort scale[47] defines 12 classes based on the speed of wind as: W0 (calm) [<0.3], W1 (light air [0.3–1.5], W2 (light breeze) [1.6–3.3], W3 (gentle breeze) [3.4–5.5], W4 (moderate breeze) [5.5–7.9], W5 (fresh breeze) [8.0–10.7], W6 (strong breeze) [10.8–13.8], W7 (near gale) [13.9–17.1], W8 (gale) [17.2–20.7], W9 (strong gale) [20.8–24.4], W10 (storm) [28.4], W11 (violent storm) [28.5–32.6], and W12 (hurricane) [≥32.7]. The last attribute, humidity (moisture), has 3 categories based on the "dew point" temperature[48]: H1 (uncomfortably dry) [0–20], H2 (comfortable) [20–60], and H3 (uncomfortably wet) [60–100]. The results of the discretized sequences are shown in **Figure 18**.

Station ID	Weather sequence
1	< (2014-8-23 02:53:04, T6, R2,W5, H1),....., (2014-8-23 11:11:12, T6, R3,W6, H1) ..... >
2	< (2008-10-23 12:45:23, T2, R5,W12, H3),....., (2008-10-23 16:44:22, T2, R4,W11, H2) .... >
3	…
…	…

Figure 18. Sequential database for the Oklahoma Mesonet dataset after discretization.

The synthetic dataset was generated by using a tool provided by the SPMF Library[49]. 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 applies to finding all timed sequential patterns in a timed sequence database TSDB. The second parameter is the number of timed sequences TS in TSDB (#seq), which refers to the number of tuples in the database.

The third parameter is the length of TS in TSDB, which can also 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 noted that the timestamp is a fixed attribute in all events. When it is said that the number of items per event is 3, for instance, it signifies three items plus the timestamp. We study the effects of all four parameters shown in **Table 1** on the synthetic dataset. However, for the T-Drive dataset, the only valid dynamic parameter that is shown in **Table 1** is the min-sup. Thus, all other three parameters are static. Now, we explain the range of the parameters and the default values of this analysis, as summarized in **Table 1**. When the 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 of 20% to 80% with the default value of 50%, which is the median of the interval. The range of the number of timed sequences parameters is from 1 to 100,000, and its median value of 50,000, is 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 set at 1 to 10 items per event; thus, the default value is 5, which is the median.

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

5.3 Competing algorithms

As mentioned in Section 3, no existing algorithm can discover the exact format of the timed sequential patterns and consider All-time Occurrences. Hence, we cannot compare Minits-AllOcc against any technique, and 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) and (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.

5.5.1 Accuracy

To validate that Minits-AllOcc always gives the same sequential patterns in terms of the numbers and contents, excluding the temporal relation, PrefixSpan was used[8]. PrefixSpan was chosen because it is one of the well-known algorithms for discovering sequential patterns. It has been proven to produce complete and correct sequential patterns. First, all temporal relations were removed from the patterns that were generated by Minits-AllOcc. Next, 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 them with pattern X. In case the order of at least one item set 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 did not matter because all the items appeared 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. All algorithms produced the complete and correct set of sequential patterns.

5.5.2 Execution time

The execution time was recorded 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 50% to 60% for T-Drive, Oklahoma Mesonet, and synthetic datasets, respectively, com-

pared to the execution time of Minits-AllOcc.

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
Oklahoma Mesonet	20.319 (min)	3,756	8.604 (min)	3,756
Synthetic data	27.319 (min)	3,780	10.825 (min)	3,780

5.5.3 Impact of minimum support

In this set of experiments, we compared execution time (ET) and the number of patterns (#patterns) for different values of minimum support threshold (min_sup) for datasets T-Drive, Oklahoma Mesonet, and synthetic. From **Figure 19**(**a**), **Figure 20**(**a**) and **Figure 21**(**a**), we can see that when the minimum support increased, the execution time of all algorithms decreased. This is because the algorithms generate fewer timed-sequential patterns when the min-sup is high, because of fewer candidate sequences that satisfy the min-sup condition. With a large amount of data and discovered timed sequential patterns, MMinits-AllOcc outperformed Minits-AllOcc, as shown in **Figure 19**(**a**), **Figure 20**(**a**), and **Figure 21**(**a**). Therefore, multi-core CPUs ought to be used when the size of the timed sequence database is large.

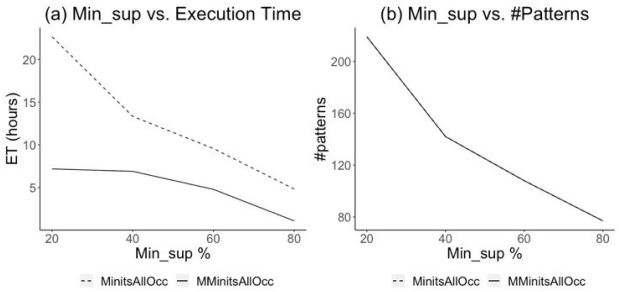

Figure 19. Parameter study for T-Drive dataset.

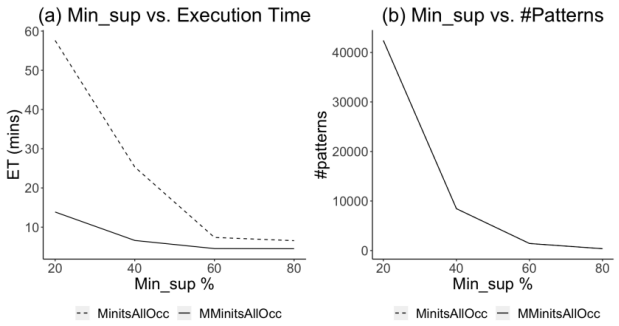

Figure 20. Parameter study for Oklahoma dataset.

The multi-core CPU version was also efficient when we had low min-sup. As shown in **Figures 20**(**a**), and **21**(**a**), the ETs of both Minits-AllOcc and MMinits-AllOcc were very close when the

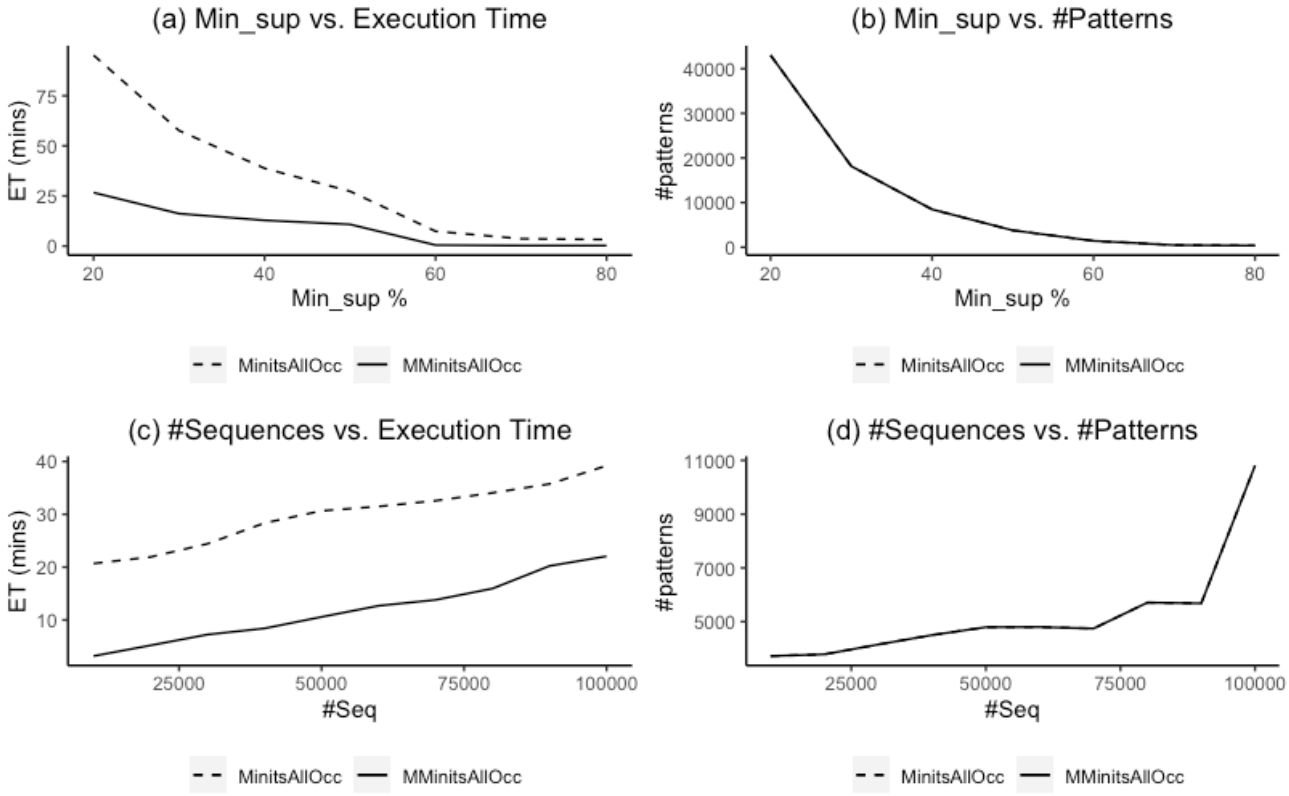

Figure 21. Parameter study (Min_sup and #Sequences) for synthetic dataset.

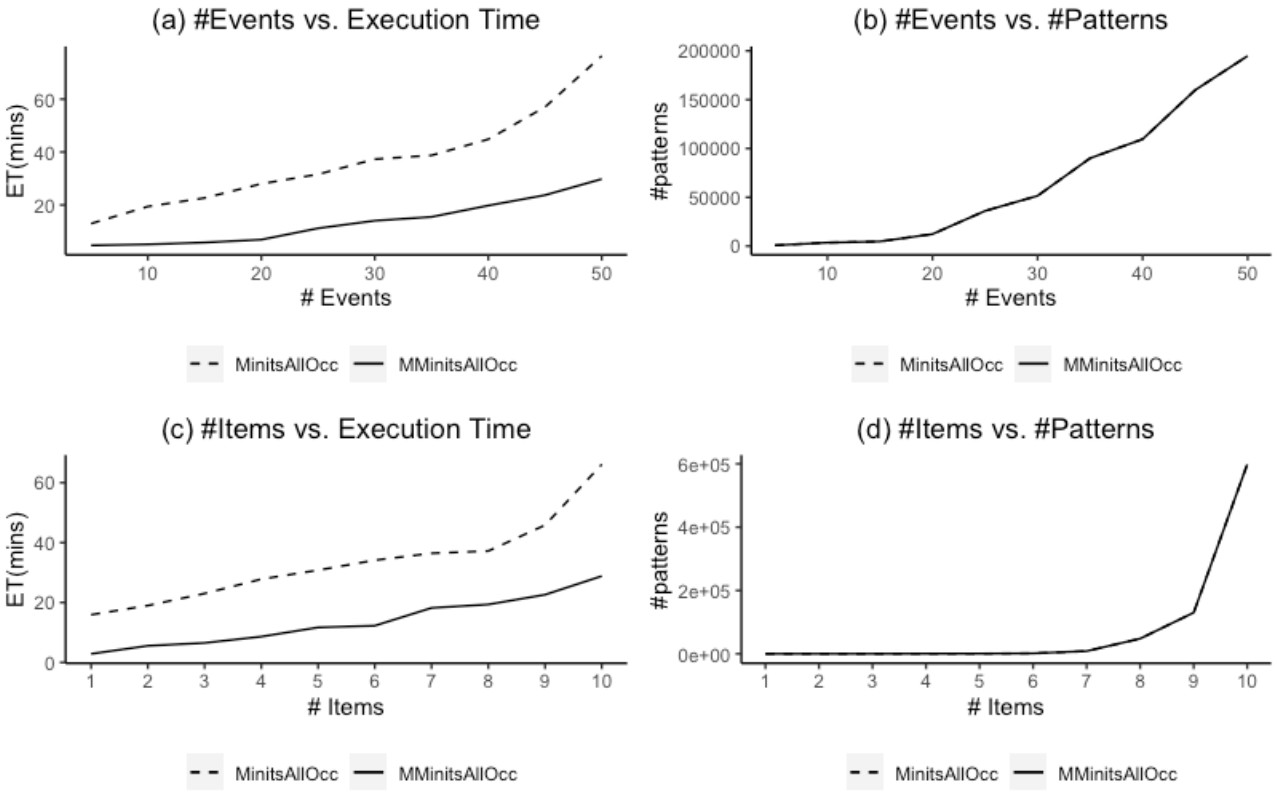

Figure 22. Parameter study (#events and #items) for synthetic dataset.

min-sup was greater than 60%. This is because 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 made 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 **Figures 19(b)**, **20(b)**, **21(b)**, **21(d)**, **22(b)**, and **22(d)**. When the min-sup increased, the number of timed sequential patterns decreased because the patterns that satisfied 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 **Figure 19(b)**, **Figure 20(b)**, and **Figure 21(b)**.

5.5.4 Impact of the number of sequences in the database

In this set of experiments, we compared the execution time (ET) and the number of discovered timed sequential patterns (#patterns) according to the number of the timed sequences (#seq). From **Figure 21(c)**, we can see that when the number of timed sequences increased, the execution times of all algorithms increased. This is because the algorithms needed more time to check the extra timed sequences that were added to the timed sequence database to decide if they contained a timed sequential pattern or not. We observed that the number of timed sequential patterns, which were generated by these algorithms, increased when the number of timed sequences increased, as shown in **Figure 21(d)**. The number of timed sequential patterns that were discovered by the algorithms also increased because the possibility of finding more patterns in the new timed sequences that satisfy the min-sup (50% as the default value) condition also increased. With an increased number of timed sequences in the database, the algorithms needed to check if some new patterns could occur and did not exist in the old timed sequences. Next, the algorithm checked their support against the threshold (min-sup). It is possible that the support of some old patterns in the database before new sequences was added did not satisfy the min-sup condition because they were not supported by enough timed sequences; but with a new timed sequence database, these patterns

became timed-sequential patterns. Thus, the number of newly discovered timed sequential patterns would increase. For example, if a database had 1,000 sequences in the synthetic dataset, the number of timed sequential patterns was 3,720, while the number of timed sequential patterns was 3,780 when the timed sequence database had 10,000 timed sequences.

5.5.5 Impact of the number of events per sequence

Figure 22 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 the discovery of more patterns because the algorithm could extend a pattern up to the length of the timed sequence. If we have a timed sequence that contains n events, we can discover a set of timed sequential patterns such that their length varies from 1 to n. Subsequently, the required time of discovering those patterns will increase as shown in **Figure 22(a)**.

5.5.6 Impact of the number of items per event

In the last experiment, we increased the number of unique items in each event. That means many new items appear in the timed sequence database TSDB which leads to detecting 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 **Figure 22(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 **Figure 22(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 timesets. A temporal relation in the timed sequential patterns is calculated after considering all possible pattern occurrences 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. Also, MMinits-AllOcc outperformed Minits-AllOcc when the dataset was enormous in size, in the length of timed sequences, or in the number of items per event. For future work, we plan to improve Minits-AllOcc to account for both long timed sequences and Dynamic Timed Sequence Database (DTSDB). The algorithm will be able to mine TSP without re-executing everything from scratch.

Conflict of interest

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Agrawal R, Srikant R. Mining sequential patterns. In: Proceedings of the eleventh international conference on data engineering; 1995 Mar 6–10; Taipei. New York: IEEE; 2002. p. 3–14. doi: 10.1109/ICDE.1995.380415.
2. Brock FV, Crawford KC, Elliott RL, *et al*. The Oklahoma Mesonet: A technical overview. Journal of Atmospheric and Oceanic Technology 1995; 12(1): 5–19. doi: 10.1175/1520-0426(1995)012<0005:tomato>2.0.co;2.
3. McPherson RA, Friedrich CA, Crawford KC, *et al*. Statewide monitoring of the mesoscale environment: A technical update on the Oklahoma Mesonet. Journal of Atmospheric and Oceanic Technology 2007; 24(3): 301–321. doi: 10.1175/JTECH1976.1.
4. Jay N, Herengt G, Albuisson E, Kohler F. Sequential pattern mining and classification of patient path. Medinfo 2004; 1667.
5. Pramono YWT, Suhardi. 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); 2014 Aug 20–21; Bandung. New York: IEEE; 2015. p. 203–208. doi: 10.1109/ICAICTA.2014.7005941.
6. Dermy O, Brun A. Can we take advantage of

time-interval pattern mining to model students activity? In: International Conference on Educational Data Mining; 2020 Jul 10–13; Online. Massachusetts: International Educational Data Mining Society; 2020. p. 69–80.
7. Rossetti MA. Analysis of weather events on US railroads [Report]. Volpe National Transportation Systems Center; 2007.
8. Simes T. A blow to train operations, can strong winds cause derailment [Report]. Australian Transport Safety Bureau; 2011.
9. Han J, Pei J, Mortazavi-Asl B, *et al*. FreeSpan: Frequent pattern-projected sequential pattern mining. In: Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2000 Aug 20–23; Boston. New York: Association for Computing Machinery; 2000. p. 355–359.
10. Han J, Pei J, Mortazavi-Asl B, *et al*. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In: Proceedings of the 17th International Conference on Data Engineering; 2001 Apr 2–6; Heidelberg. New York: IEEE; 2002. p. 215–224.
11. Zaki MJ. Spade: An efficient algorithm for mining frequent sequences. Machine Learning 2001; 42: 31–60. doi: 10.1023/A:1007652502315.
12. Jou C, Shyur HJ, Yen CY. Timed sequential pattern mining based on confidence in accumulated intervals. In: Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014); 2014 Aug 13–15; Redwood City. New York: IEEE; 2015. p. 771–778. doi: 10.1109/IRI.2014.7051967.
13. Kumar S, Mohbey KK. A review on big data based parallel and distributed approaches of pattern mining. Journal of King Saud University-Computer and Information Sciences 2022; 34(5): 1639–1662. doi: 10.1016/j.jksuci.2019.09.006.
14. Ghorbani M, Abessi M. A new methodology for mining frequent itemsets on temporal data. IEEE Transactions on Engineering Management 2017; 64(4): 566–573. doi: 10.1109/TEM.2017.2712606.
15. Zhao P, Jonietz D, Raubal M. Applying frequent-pattern mining and time geography to impute gaps in smartphone-based human-movement data. International Journal of Geographical Information Science 2021; 35(11): 2187–2215. doi: 10.1080/13658816.2020.1862126.
16. Aggarwal A, Toshniwal D. Frequent pattern mining on time and location aware air quality data. IEEE Access 2019; 7: 98921–98933. doi: 10.1109/ACCESS.2019.2930004.
17. Ritika, Gupta SK. HUFTI-SPM: High-utility and frequent time-interval sequential pattern mining from transactional databases. International Journal of Data Science and Analytics 2022; 13: 239–250. doi: 10.1007/s41060-021-00297-7.
18. Huang JW, Jaysawal BP, Chen KY, Wu YB. Mining frequent and top-k high utility time interval-based events with duration patterns. Knowledge and Information Systems 2019; 61: 1331–1359. doi: 10.1007/s10115-019-01333-6.
19. Mirbagheri SM, Hamilton HJ. Mining high utility patterns in interval-based event sequences. Data & Knowledge Engineering 2021; 135: 101924. doi: 10.1016/j.datak.2021.101924.
20. Giannotti F, Nanni M, Pedreschi D. Efficient mining of temporally annotated sequences. In: Frasconi P, Landwehr N, Manco G, Vreeken J (editors). Proceedings of the 2006 SIAM International Conference on Data Mining; 2006 Apr 20–22; Bethesda. Philadelphia: Society for Industrial and Applied Mathematics; 2006. p. 348–359. doi: 10.1137/1.9781611972764.31.
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; 2013 Nov 5; Orlando. New York: Association for Computing Machinery; 2013. p. 26–32. doi: 10.1145/2534303.2534313.
22. Titarenko SS, Titarenko VN, Aivaliotis G, Palczewski J. Fast implementation of pattern mining algorithms with time stamp uncertainties and temporal constraints. Journal of Big Data 2019; 6(1): 1–34. doi: 10.1186/s40537-019-0200-9.
23. Karsoum S, Gruenwald L, Barrus C, Leal E. Using timed sequential patterns in the transportation industry. In: 2019 IEEE International Conference on Big Data (Big Data); 2019 Dec 9–12; Los Angeles. New York: IEEE; 2020. p. 3573–3582. doi: 10.1109/BigData47090.2019.9006394.
24. Srikant R, Agrawal R. Mining sequential patterns: Generalizations and performance improvements. In: Apers P, Bouzeghoub M, Gardarin G (editors). Advances in Database Technology—EDBT'96: 5th International Conference on Extending Database Technology; 1996 Mar 25–29; Avignon. Heidelberg: Springer; 1996. p. 1–17.
25. Fournier-Viger P, Lin JCW, Kiran RU, *et al*. A survey of sequential pattern mining. Data Science and Pattern Recognition 2017; 1: 54–77.
26. Huynh B, Vo B, Snasel V. An efficient method for mining frequent sequential patterns using multi-core processors. Applied Intelligence 2017; 46: 703–16.

doi: 10.1007/s10489-016-0859-y.
27. Li H, Zhou X, Pan C. Study on GSP algorithm based on Hadoop. In: 2015 IEEE 5th International Conference on Electronics Information and Emergency Communication; 2015 May 14–16; Beijing. New York: IEEE; 2015. p. 321–324. doi: 10.1109/ICEIEC.2015.7284549.
28. Wei Y, Liu D, Duan L. Distributed PrefixSpan algorithm based on MapReduce. In: 2012 International Symposium on Information Technologies in Medicine and Education; 2012 Aug 3–5; Hokkaido. New York: IEEE; 2012. p. 901–904. doi: 10.1109/ITiME.2012.6291449.
29. Yu X, Li Q, Liu J. Scalable and parallel sequential pattern mining using spark. World Wide Web 2019; 22(1): 295–324. doi: 10.1007/s11280-018-0566-1.
30. Gan W, Lin JCW, Fournier-Viger P, *et al*. A survey of parallel sequential pattern mining. ACM Transactions on Knowledge Discovery from Data (TKDD) 2019; 13(3): 1–34. doi: 10.1145/3314107.
31. Dong G, Pei J. Sequence data mining. New York: Springer Science & Business Media; 2007.
32. Patnaik D, Butler P, Ramakrishnan N, *et al*. Experiences with mining temporal event sequences from electronic medical records: Initial successes and some challenges. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2011 Aug 21–24; San Diego. New York: Association for Computing Machinery; 2011. p. 360–368. doi: 10.1145/2020408.2020468.
33. Chen YL, Chiang MC, Ko MT. Discovering time-interval sequential patterns in sequence databases. Expert Systems with Applications 2003; 25(3): 343–354. doi: 10.1016/S0957-4174(03)00075-7.
34. Hu YH, Huang TCK, Yang HR, Chen YL. On mining multi-time-interval sequential patterns. Data & Knowledge Engineering 2009; 68(10): 1112–1127. doi: 10.1016/j.datak.2009.05.003.
35. AlZahrani MY, Mazarbhuiya FA. Discovering constraint-based sequential patterns from medical datasets. International Journal of Recent Technology and Engineering 2019; 8(4): 724–728. doi: 10.35940/ijrte.D7011.118419.
36. 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; 2007 Aug 12–15; San Jose. New York: Association for Computing Machinery; 2007. p. 330–339. doi: 10.1145/1281192.1281230.
37. Mannila H, Toivonen H, Inkeri Verkamo A. Discovery of frequent episodes in event sequences. Data Mining and Knowledge Discovery 1997; 1: 259–289. doi: 10.1023/A:1009748302351.
38. Zimmermann A. Understanding episode mining techniques: Benchmarking on diverse, realistic, artificial data. Intelligent Data Analysis 2014; 18(5): 761–791. doi: 10.3233/IDA-140668.
39. Zhang D, Lee K, Lee I. Mining medical periodic patterns from spatio-temporal trajectories. In: Siuly S, Lee I, Huang Z (editors). Health Information Science: 7th International Conference; 2018 Oct 5–7; Cairns. Berlin: Springer International Publishing; 2018. p. 123–133.
40. Zhang D, Lee K, Lee I. Mining hierarchical semantic periodic patterns from GPS-collected spatio-temporal trajectories. Expert Systems with Applications 2019; 122: 85–101. doi: 10.1016/j.eswa.2018.12.047.
41. Yuan J, Zheng Y, Zhang C, *et al*. T-drive: Driving directions based on taxi trajectories. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems; 2010 Nov 2; San Jose. New York: Association for Computing Machinery; 2010. p. 99–108. doi: 10.1145/1869790.1869807.
42. Yuan J, Zheng Y, Xie X, Sun G. T-drive: Enhancing driving directions with taxi drivers' intelligence. IEEE Transactions on Knowledge and Data Engineering 2011; 25(1): 220–232. doi: 10.1109/TKDE.2011.200.
43. Ester M, Kriegel HP, Sander J, Xu X. A density-based algorithm for discovering clusters in large spatial databases with noise. In: Simoudis E, Han J, Fayyad U (editors). Proceedings of the Second International Conference on Knowledge Discovery and Data Mining; 1996 Aug 2–4; Portland. Washington, D.C.: AAAI Press; 1996. p. 226–231.
44. Karsoum S, Gruenwald L, Leal E. Impact of trajectory segmentation on discovering trajectory sequential patterns. In: 2018 IEEE International Conference on Big Data (Big Data); 2018 Dec 10–13; Seattle. New York: IEEE; 2019. p. 3432–3441. doi: 10.1109/BigData.2018.8622209.
45. What is the heat index? [Internet]. Amarillo: Weather Forecast Office; [2021 Oct 17]. Available from: https://www.weather.gov/ama/heatindex.
46. Water Science School. The 100-year flood [Internet]. Virginia: USGS; 2018 [cited 2021 Oct 17]. Available from: https://www.usgs.gov/special-topic/water-science-school/science/100-year-flood?qt-science_center_objects=0#qt-science_center_objects.
47. The Beaufort wind scale [Internet]. London: MetMatters; [cited 2021 Oct 17]. Available from: https://www.rmets.org/resource/beaufort-scale.

48. Dew point vs humidity [Internet]. La Crosse: Weather Forecast Office; [cited 2021 Oct 17]. Available from: https://www.weather.gov/arx/why_dewpoint_vs_humidity.
49. Fournier-Viger P, Lin JCW, Gomariz A, *et al*. The SPMF open-source data mining library version 2. In: Berendt B, Bringmann B, Fromont É, *et al*. (editors). 19th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2016) Part III; 2016 Sept 19–23; Riva del Garda. Berlin: Springer; 2016. p. 36–40. doi: 10.1007/978-3-319-46131-1_8.