MinitsDays: An Efficient Algorithm for Mining Timed Sequential Pattern from Dynamic Database

Somayah Karsoum [a,1], and Le Gruenwald [a]
[a] University of Oklahoma, Norman, OK 73019, USA
E-mail:somayah.karsoum@ou.edu ; ORCID: https://orcid.org/0000-0002-8855-0175
[a] University of Oklahoma, Norman, OK 73019, USA
E-mail: ggruenwald@ou.edu; ORCID: https://orcid.org/0000-0002-5245-4747

Abstract. One of the most important tasks in data mining is the discovery of frequent sequential patterns. This task focuses on identifying subsequences within a sequence database that frequently occur in the same timestamp order. An extension of this task is timed sequential pattern mining, which discovers frequent sequences from a sequence database, along with the temporal relationships in patterns. Mining such patterns supports a wide range of applications, including recommendation systems in transportation, healthcare, and weather forecasting. While many existing approaches have been developed to mine sequential patterns and timed sequential patterns, they assume the database is static. However, in real-world scenarios, databases are dynamic, evolving in response to user interactions, updated data, and changing requirements. Consequently, finding the complete set of timed sequential patterns efficiently, without repeatedly rescanning the database from scratch, remains a significant research challenge. To fill this gap, a novel algorithm called MINIng Timed Sequential Patterns in DYnamic Sequence Databases (MinitsDays) is proposed. MinitsDays is designed to discover timed sequential patterns that capture the temporal relation in patterns in a dynamic timed sequence database. Extensive theoretical and experimental evaluations were conducted to assess the performance of MinitsDays using both real-world and synthetic datasets. The experimental results demonstrate the effectiveness and advantages of the proposed approach. Additionally, the algorithm leverages parallelism through multicore CPUs to significantly enhance performance.

Keywords. Data mining, Sequential pattern mining, Timed sequential patterns, Singe-core and multi-core processor.

[1] Corresponding Author: Somayah Karsoum, E-mail: somayah.karsoum@ou.edu

1. Introduction

Sequential pattern mining [1] is a data mining task that discovers frequent subsequences in a sequence database of time-ordered transactional data. Identifying interesting, useful, and unexpected patterns is beneficial for a wide range of real-world applications, such as educational data mining [2], network intrusion detection [3] , and customer shopping behaviors [1]. Existing sequential pattern mining algorithms, such as those in[4] ,[5], [6]and [7], use implicit timestamps to order the itemsets within a sequential pattern. However, they do not retain the transition time between these itemsets. In many applications, it is crucial to know the time it takes to move from one itemset to another within a pattern. For example, in healthcare applications, knowing when the next symptom of a heart attack will occur can help healthcare providers make more accurate diagnoses, administer timely treatments, and intervene earlier in critical cases. By including the temporal relationship of the transition time—indicating when the next symptom is likely to appear, we can answer not only questions like "In which order do heart attack symptoms frequently occur?" but also "When do these symptoms frequently occur?" This led to the idea of incorporating transition time between itemsets in a sequential pattern, indicating the required time range to move from one itemset to another. This newly discovered type of pattern is referred to as a *timed sequential pattern*.

Most sequential pattern mining algorithms, including the two proposed algorithms, Minits [8] and Minits-AllOcc [9], assume that databases are static, meaning they do not change over time. However, real-world databases are dynamic and can evolve in response to various factors such as data updates, user interactions, or changing requirements. Therefore, it is necessary to develop an algorithm capable of detecting and handling these changes, and of discovering the complete set of timed sequential patterns after the database has been modified, without re-mining the entire database from scratch. The following are different scenarios that can occur in a timed sequence database:

- CASE 1*:* New tuples are inserted into the Timed Sequence Database (TSDB). For example, Fig. 1 shows the TSDB four tuples for four patients. Each patient's temperature (T) and blood pressure (BP) are measured during their visits. After one week, for example, a new patient may be added, requiring the insertion of a new

tuples into the existing TSDB. The updated TSDB is shown in Fig. 2. As new patients continue to arrive, additional tuples will be appended to the existing TSDB.

Patient ID	Timed Sequence
P1	< {5, TC1,BPC3}, {12, TC1,BPC2}>
P2	< {2, TC1,BPC1}, {19,TC1,BPC3}, {25,TC1,BPC5}>
P3	< {21, TC1,BPC3}, {25,TC1,BPC2}>
P4	< {10, TC1,BPC2}, {20,TC1,BPC2}, {30,TC2,BPC3}>

Fig. 1. Timed Sequence Database for Patients

Patient ID	Timed Sequence
P1	<{5, TC1,BPC3}, {12, TC1,BPC2}>
P2	<{2, TC1,BPC1}, {19,TC1,BPC3}, {25,TC1,BPC5}>
P3	<{21, TC1,BPC3}, {25,TC1,BPC2}>
P4	<{10, TC1,BPC2}, {20,TC1,BPC2}, {30,TC2,BPC3}>
P5	<{34,TC2 BPC5}>

Fig. 2. Updated Timed Sequence Database after insertion a new patient

- CASE 2: New events are inserted into an existing timed sequence in the TSDB. For example, a company in the oilfield services industry may attempt to predict the failure of electric pump sensors by collecting daily measurements for temperature (TC), pressure (PC), and voltage (VC). Fig. 3 shows the TSDB for three sensors (S). After one day, new measurements for each sensor are obtained and must be inserted, highlighted in bold, into the existing tuples in the TSDB as shown in Fig. 4. Each day, a new version of the TSDB is generated, maintaining the same number of tuples but with varying sequence lengths.

Sensor ID	Timed Sequence
S1	<{5,TC1, PC5, VC1}, {6,TC1,PC5, VC2}>
S2	<{2,TC3,PC4, VC3}, {3,TC2, PC1,VC2}>
S3	<{10,TC3,VC1}>

Fig. 3. Timed Sequence Database for Sensors

Sensor ID	Timed Sequence
S1	<{5,TC1, PC5, VC1}, {6,TC1,PC5, VC2}, {**9,TC1,PC1,VC1}**>
S2	<{2,TC3,PC4, VC3}, {3,TC2, PC1,VC2}, {**6,TC3,PC3,VC1}**>
S3	<{10,TC3,VC1}, {**12,PC4,VC1}**>

Fig. 4. Updated Timed Sequence Database for Sensors

- CASE 3: An existing timed sequence is deleted from the Timed Sequence Database (TSDB). In business intelligence, stock data is often analyzed in detail to assess the viability of each investment. A snapshot of a stock's timed sequence database is shown in Fig. 5. The stock price at the opening is referred to as the open, and the price at the closing is referred to as the close. Based on their values, opening and closing prices are categorized into three levels: low (open1 or close1), medium (open2 or close2), and high (open3 or close3). For instance, the fluctuations in the stock price of a company that filed for bankruptcy five years ago, such as C4, may no longer significantly impact the current market trends. Therefore, it may be unnecessary to retain that information in the TSDB, and the corresponding tuples can be deleted. As shown in Fig.6.

Company ID	Timed Sequence
C1	<{5, Open1, Close3}, {12, Open1, Close2}, {17, Open1, Close3}>
C2	<{21, Open1, Close1}, {24, Open1, Close3}, {39, Open1, Close3}>
C3	<{2, Open2, Close3}, {19, Open1, Close2}>
C4	<{4, Open1, Close3}, {9, Open1, Close3}, {13, Open1, Close3}>

Fig.5. Timed Sequence Database for Stocks

Company ID	Timed Sequence
C1	<{5, Open1, Close3}, {12, Open1, Close2}, {17, Open1, Close3}>
C2	<{21, Open1, Close1}, {24, Open1, Close3}, {39, Open1, Close3}>
C3	<{2, Open2, Close3}, {19, Open1, Close2}>

Fig. 6. Updated Timed Sequence Database for Stocks after delete Sears company

- CASE 4: The schema of an event in the TSDB is changed. Consider the same TSDB from Case 1 in Fig. 1, which contains temperature (T) and blood pressure (BP) measurements for four patients. Originally, each event in the TSDB follows the schema: *(Timestamp, Temperature, Blood Pressure)*. However, doctors may decide to monitor an additional measurement—cholesterol level (C)—to assess its clinical relevance. As a result, the schema of each event in the TSDB is updated to: *(Timestamp, Temperature, Blood Pressure, Cholesterol Level)*. In this case, two possible scenarios may arise:

  - CASE 4.1: Changing the schema for newly incoming events. Starting from the current timestamp, the new schema will be applied, and all previous events will be ignored, since the doctors are now interested in identifying relationships among the three updated symptoms. The updated TSDB is illustrated in Fig. 7. This scenario can be treated as if we are working with a new TSDB containing only the data shown in Fig. 8. To discover the new set of timed sequential patterns among the three symptoms, an algorithm must process the updated version of the TSDB and extract the revised timed sequential patterns. Subsequently, if a new patient P5 arrives, their timed sequence will be inserted into this TSDB, falling under the process described in CASE 1.

Patient ID	Timed Sequence
P1	<{5,TC1,BPC3},{12,TC1,BPC2},**{31,TC1,BPC1,CC1},{35,TC2,BPC5,CC2}>**
P2	<{2, TC1,BPC1}, {19,TC1,BPC3}, {25,TC1,BPC5}>
P3	<{21, TC1,BPC3}, {25,TC1,BPC2}, {**32, TC2,BPC3,CC2}**>
P4	<{10, TC1,BPC2}, {20,TC1,BPC2}, {30,TC2,BPC3}>

Fig. 7. Updated Timed Sequence Database for Case 4.1

Patient ID	Timed Sequence
P1	<{**31,TC1,BPC1,CC1}, {35, TC2,BPC5,CC2}>**
P3	<{**32, TC2,BPC3,CC2}**>

Fig. 8. New Times sequence database for Case 4.1

  - CASE 4.2: Modifying the schema of all previous events. In this scenario, we assume that the data, such as cholesterol levels (C), was available but previously ignored, as doctors were not initially interested in analyzing it. Later, however, they decide to study the relationship between temperature (T), blood pressure (BP), and cholesterol level (C). To accommodate this, it becomes necessary to reprocess the existing raw data: read it again, discretize the values, and reconstruct the Timed Sequence Database (TSDB), as shown in Fig. 9. All of these tasks are considered pre-processing steps. Once the updated version of the TSDB is ready,

the algorithm can then be executed. At this stage, the database is treated as an entirely new TSDB, and the algorithm is run from the beginning to discover the timed sequential patterns that capture the relationships among the newly considered symptoms.

Patient ID	Timed Sequence
P1	<{5,TC1,BPC3,**CC1}**,{12,TC1,BPC2,**CC1}**,{**31,TC1,BPC1,CC1},{35,TC2,BPC5,CC2 }>**
P2	<{2,TC1,BPC1, **CC1}**, {19,TC1,BPC3, **CC2}**, {25,TC1, BPC5, **CC1}**>
P3	<{21,TC1,BPC3, **CC2}**, {25,TC1,BPC2, **CC1}**, {**32, TC2,BPC3,CC2}** >
P4	<{10,TC1,BP2, **CC2}**, {20,TC1,BPC2, **CC2}**, {30,TC2,BPCC3, **C2}**>

Fig.9. Updated Timed Sequence Database for Case 4.2

- CASE 5**:** An existing event is deleted or modified. In the Knowledge Discovery in Databases (KDD) process, there is a critical step known as pre-processing. During this step, any missing, erroneous, low-quality, or redundant data is either removed or corrected. As a result, once pre-processing is complete, there should be no need to delete or modify existing events. We assume that the output of the pre-processing stage is accurate and reliable, and therefore, no further changes to the existing events are required.

The contributions of this paper are as follows:

1. We propose the MinitsDays algorithm, which effectively handles CASE 1, CASE 2, and CASE 3. It can discover timed sequential patterns in an updated Timed Sequence Database (TSDB) resulting from the insertion or deletion of timed sequences, without the need to remine the entire TSDB.
2. We introduce a parallel implementation of the MinitsDays algorithm to enhance performance when working with Big Data.
3. We conduct extensive experiments comparing the performance of the single-core version of the algorithm against its multi-core counterpart, using both real and synthetic datasets.

The remainder of the paper is organized as follows. Section 2 reviews the problem definitions. Section 3 discusses related work. Section 4 details the algorithm's workings. Section 5 presents performance evaluations across different datasets. Finally, Section 6 concludes the paper and outlines future work.

2. Problem Definition

In this section, we review the definitions of the sequential pattern mining problem [1] and timed sequential pattern mining problem [9]. Recalling the traditional sequential pattern mining problem [1], we define an **itemset** I as a set of **items,** such that $I \subseteq X$, where $X = \{x_1, x_2, \ldots x_l\}$ is a set of items in the database. A **sequence** (tuple) *s* is an ordered list (based on timestamps) of itemsets. A sequence $A = <\{a_1\}, \{a_2\}, \ldots\{a_n\}>$ is **contained in** another sequence $B = <\{b_1\}, \{b_2\}, \ldots\{b_m\}>$ and B is a **super-sequence** of A if there exists a set of integers, $1 \leq j_1 < j_2 < \ldots < j_n \leq m$, such that $a_1 \subseteq b_{j_1}, a_2 \subseteq b_{j_2}, \ldots, a_n \subseteq b_{jn}$.

A **sequence database** S is a set of sequences <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. 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**. Since our problem also considers the temporal data, we incorporate timestamps explicitly in the database and introduce new definitions.

Recalling the timed sequential pattern mining problem[9] , A timed **event** is a pair $e = (t, I)$, where *I* am an itemset that occurs at the timestamp *t*. We use e. t and e.I 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** $TS = <\{e_1\}, \{e_2\}, \ldots, \{e_k\}>$, such that $e_i.x \subseteq I$ $(1 \leq i \leq k)$. 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.

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 item sets where it represents any descriptive statistic, such as an average of transition time or range, calculated based on the values of the delta Δ. TSP is denoted as: $TSP = <\{I_0\} [\tau_1] \{I_1\} [\tau_2] \{I_2\} \ldots\ldots [\tau_n] \{I_n\}>$. For brevity, in the rest of this paper, when we mention a pattern, we refer to a timed sequential pattern.

Example 1. (**Running Example**) The Timed Sequence Database shown in Fig. 10 is used as an illustrative example. For simplicity, we use letters to represent items, each corresponding to a property of an object in the database (e.g., temperature and blood pressure for patients), and integers to represent timestamps, indicating when those properties were recorded. In this example, there are four timed sequences labeled TS1 to TS4. Each timed sequence consists of a set of events ordered by their timestamps. For instance, TS1 includes two events: the first event {5, a, b}, which occurred at timestamp 5, followed by the second event {12, d, g}, which occurred at timestamp 12.

Let us assume the min-sup =50%; since the support of sequence A= <{a}{b}> is 50%, the sequence is a sequential pattern. The delta Δ is the difference between the timestamps of two events in each timed sequence in database, and it is calculated as follows:

In TS3, {a} occurs at timestamp 2, and {b} occurs at timestamp 19. Time difference: 19–2 = 17.

- In TS4, {a} occurs at timestamp 10, and {b} occurs at timestamp 19. Time difference: 19 − 10 = 9.
- TS4 also includes another {b} at timestamp 30 following the same {a} at timestamp 10. Time difference: 30 − 10 = 20
- From these observations: minimum time is 9 and maximum time is 20. Thus, the final timed sequential pattern is: <{a} [9, 20] {b}>.

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}>.

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

Fig. 10. An Example of Timed Sequence Database

3. Related Works

In this section, we review state-of-the-art techniques for extracting sequential patterns from dynamic databases. The problem of sequential pattern mining was first introduced in [1] , where several algorithms were proposed to enhance the performance of pattern discovery. We categorize the current techniques into two groups based on the type of sequence database they address: incremental and progressive. An incremental database refers to a setting where new data is continuously added, while a progressive database allows both the addition of new data and the removal of obsolete data.

One of the earliest approaches developed in the field of Incremental Sequential Pattern Mining (ISPM) is FastUP[10], which is based on the GSP [4] technique with enhancements for support counting and candidate generation. Prior to generating and validating candidates, the algorithm applies a pruning method that leverages information from previous mining results to determine sequence thresholds. It also incorporates efficient filtering techniques during support counting and minimizes the generation of unnecessary candidate sequences. These improvements result in better time and space efficiency compared to traditional sequential pattern mining algorithms. Similarly, ISM [11] builds upon the SPADE [7] approach and focuses on maintaining a sequence lattice from the previous database. This lattice contains both frequent sequences and those in the negative border, which helps in efficiently updating the mined patterns. During the initial scan of the database, ISM performs lattice pruning and identifies negative border sequences before generating new frequent sequences using the SPADE methodology. Due to the high computational cost of ISM, the IUS algorithm [12] was introduced as a more efficient alternative, also drawing from the SPADE approach. IUS reuses frequent and negative border sequences from the original database and introduces a new threshold, called the minimum support of negative border sequences (min_nbd_supp), to better manage memory and computational resources compared to ISM.

In [13] , the authors proposed GSP+ (Generalized Sequential Pattern Plus) and MFS+ (Maximal Frequent Sequence Plus), introducing the concept of sequence difference to represent changes between two sequences. This concept is used to update frequent sequences incrementally. GSP+ begins by identifying frequent sequences in the initial database using the GSP algorithm. When the database is updated with new transactions or modifications, GSP+ incrementally updates the frequent sequences without recomputing everything from scratch. By leveraging sequence differences, the algorithm efficiently focuses only on the changes rather than reprocessing the entire database. Similarly, MFS+ starts by identifying maximal frequent sequences—sequences that are frequent and cannot be extended without becoming infrequent—using the MFS algorithm. When updates occur, MFS+

incrementally updates these sequences by applying the sequence difference concept, avoiding full re-computation. For incremental discovery of sequential patterns, BSPin [14] adopts a backward mining approach and introduces the concept of stable sequences, which are patterns whose support counts remain unchanged in the updated database. Candidate sequences are generated through backward extensions from 1-patterns. Then, (k+1)-sequences are generated recursively from k-patterns, and their frequencies are evaluated. Stable sequences are excluded from this support-counting process, reducing computational overhead. Furthermore, BSPin proved that any extension of a stable sequence is also stable, enabling the algorithm to skip unnecessary computations and further accelerate the incremental mining process.

The ISPTAR algorithm (Incremental Sequential Pattern mining for Temporal Association Rules) [15] was designed to mine multivariate temporal association rules by combining historical and incremental data in dynamic datasets. It addresses the inefficiency of repeatedly scanning the historical database by retaining previously discovered patterns and merging them with new incoming data. To identify temporal relationships within multivariate time series, ISPTAR first generates a temporal sequence dataset from the temporal transaction dataset, which is produced through fuzzy discretization of the multivariate time series. Then, it considers both the valid time period of each pattern and the temporal relationships among the pattern's elements to mine fuzzy temporal sequential patterns. This mining process is based on the PrefixSpan algorithm. Finally, ISPTAR constructs fuzzy temporal association rules from the mined sequential patterns to reveal associations between different attributes across time. In [16], IncSeq addressed the challenge of mining frequent serial episodes over data streams. It initialized the parameters: stream window W, data stream, and tree data structure. A tree node is a 4-tuple consisting of: sequential pattern (α), list of minimal occurrences, which is the itemsets IDs, of α in the stream (I), descendant nodes representing patterns of size ‖α‖ + 1 (S), and descendant nodes representing patterns of size ‖α‖ + 1 with specific relationships (C). When the algorithm receives a new itemset from the data stream, two actions are triggered: (1) occurrences associated with the first itemset in the window are deleted; and (2) patterns and occurrences associated with the new itemset are added. Most of the computational effort is incurred by the addition phase, which consists of three sub-steps: finishing the lists of occurrences; merging sub-itemsets of the new itemset into the current tree; and trimming nodes with non-frequent patterns. The IncSeq has some drawbacks. The tree structure used in IncSeq to represent sequential patterns, and their occurrences may require significant memory and computational resources to maintain and update, especially as the size of the tree grows with the influx of new data. Also, the performance of IncSeq may be sensitive to parameter settings such as window size, minimal support threshold, and other configuration options, requiring careful tuning for optimal results.

Due to the increasing size of big data, traditional sequential pattern mining algorithms often suffer from memory limitations and decreased performance. MR-INCSPM [17] addresses these challenges by leveraging the MapReduce framework to efficiently process large datasets and ensure scalability. In its first phase—the map phase—the algorithm employs a backward mining strategy [14] for discovering sequential patterns from an incremental database. It also introduces a novel data structure called CRMAP, which maps items in an input sequence to their preceding items, facilitating effective candidate generation and pruning to reduce the search space. The second phase handles the processing of these candidate sequences, performs support counting, and identifies frequent sequential patterns. Experimental results demonstrate that MR-INCSPM significantly outperforms non-incremental MapReduce-based algorithms in terms of execution speed and achieves linear scalability with respect to data size.

To mine sequential patterns in progressive databases, researchers proposed an algorithm called PISA (Progressive mIning of Sequential pAtterns)[18]. This algorithm focuses on discovering sequential patterns within a recent time frame of interest by handling both the insertion of new data and the removal of obsolete data. The concept of a Period of Interest (POI) is used to define a sliding time window for analysis. Only sequences containing elements with timestamps within the POI are retained for mining, while older sequences are pruned, ensuring that only the most relevant and recent data contribute to the pattern discovery process. To manage sequences in this dynamic environment, PISA uses a data structure known as the Progressive Sequential Tree (PS-tree). The PS-tree is initialized to store sequences within the POI, and when new data arrive at timestamp t+1, the tree is traversed in post-order. During this process, outdated elements are removed, existing sequences are updated, and new elements are inserted into the PS-tree. A parallel version of this algorithm, called Parallel Progressive Sequential Pattern (PPSP), has also been implemented on a GPU platform to improve performance [19]. Additionally, the work in [20] highlighted a gap in existing sequential pattern mining algorithms: the lack of a metric to evaluate the significance of discovered patterns. They proposed assigning weights to timestamps, where the presence or absence of a pattern at a given time helps determine its importance, enhancing practical utility in real-world applications.

A novel technique called Wtd_Seq_Pat was introduced to address the challenge of quantifying the importance of extracted sequential patterns by assigning weights based on their temporal spread. Instead of relying solely on raw support counts, Wtd_Seq_Pat considers the duration over which a pattern occurs, adjusting the support value to better reflect its true significance. To implement this, a weighted M-ary tree (WM-ary tree) is used to represent sequences over time. As the algorithm processes each timestamp, it traverses the tree to insert

new elements and update existing nodes according to their occurrences, ultimately identifying frequent sequential patterns. Given the limitations of traditional algorithms in handling large-scale data, a Distributed Progressive Sequential Pattern mining algorithm (DPSP) was proposed in [21] To further address issues of scalability, noise, and non-stationary data.

To further address issues of scalability, noise, and non-stationary data, [22] proposed HDCL, a hybrid approach that integrates deep learning with sequential pattern mining in progressive databases. The technique applies Discrete Wavelet Transform (DWT) to preprocess non-stationary data, transforming it into time-series form by decomposing it into approximation and detail coefficients. These coefficients represent the low- and high-frequency components of the data, respectively. A Convolutional Neural Network (CNN) is then used to mine frequent sequence patterns from the transformed data, offering robustness against noise and improved pattern recognition in evolving data environments.

While incremental and progressive sequential pattern mining algorithms can identify the order of successive events, they do not capture the time intervals between those events. However, many existing algorithms that incorporate temporal relations within itemsets, such as those presented in [23], [24], [25], [26]and [27], are designed to mine timed sequential patterns from a static timed sequence database (TSDB). Consequently, if any updates occur in the database, such as the addition or removal of sequences, these algorithms must be re-executed from scratch to discover the updated set of timed sequential patterns. This complete reprocessing can be computationally expensive, particularly when dealing with large-scale datasets.

To the best of our knowledge, no existing algorithm can discover the complete set of timed sequential patterns from a dynamic timed sequence database. To address this gap, MinitsDays is proposed to efficiently extract the full set of timed sequential patterns from a dynamic timed sequence database.

4. The proposed Algorithm: MinitsDays

4.1. Overview of MinitsDays

MinitsDays is a technique for discovering the complete and updated set of timed sequential patterns from a dynamic database. The core idea is to leverage historical information obtained from previous executions of the algorithm and update this information based on changes that have occurred in the database. In [14], a novel methodology called backward mining was introduced, which contrasts with the traditional forward mining approach. In forward mining, when a k-pattern is discovered, one item is appended to the end of the pattern to generate a (k+1)-candidate. In backward mining, however, one item is appended to the beginning of the pattern to generate a (k+1)-candidate. Backward mining introduces the concept of stable sequences—subsequences whose support counts remain unchanged in the updated database. By identifying and eliminating these stable sequences, the algorithm optimizes the support counting process, significantly improving pattern maintenance. This approach prunes all stable sequences and their super sequences, thereby minimizing the need for repeated database projections and reducing computational overhead. The correctness of this property was approved in [14]. MinitsDays adopts this property and uses two data structures to store and manage all necessary data from the current TSDB. It traverses these structures to update the set of timed sequential patterns. The 1-sequence data structure contains all distinct 1-items in the timed sequence database. Each 1-item is associated with its support count and a list of all timed sequence IDs that contain this item, along with their corresponding timestamps, as illustrated in Fig. 11.

Timed Sequential Pattern TSP : Support count
Timed Sequence ID_1: List of timestamps of an event contain the item Timed Sequence ID_2: List of timestamps of an event contain the item ….. Timed Sequence ID_i: List of timestamps of an event contain the item

Fig. 11: 1-sequence Data Structure for an item

The 1-sequence data structure for the timed sequence database in Fig. 10 is shown in Fig. 12. For instance, item a appears in three timed sequences: TS1, TS3, and TS4. In TS1, item a occurs in the first event at timestamp 5. In TS3, it appears in two events: the first at timestamp 2 and the second at timestamp 19. This data structure is similarly created for all other distinct items, b, d, e, f, and g, as shown in Fig. 12.

a: 3	b: 3	d: 3	e: 1	f: 1	g: 2
TS1: {5} TS3: {2,19} TS4: {10}	TS1: {5} TS3: {19} TS4: {19,30}	TS1: {12} TS3: {25} TS4: {20}	TS2: {21}	TS4: {19}	TS1: {5,12} TS2: {21}

Fig.12. 1-sequence Data Structure for all Items in Timed Sequence Database Shown in Fig.10.

The second data structure is the Timed Sequential Patterns Suffix Tree (TSS-tree). This tree contains only the timed sequential patterns, meaning those patterns that satisfy the minimum support (min_sup) condition. Each node in the TSS-tree follows the same structure as the 1-sequence data structure. Specifically, every node includes: the pattern itself, its support count, and a list of all occurrences of that pattern in the database. The children of a node represent the backward extensions of that pattern. For example, the timed sequential patterns a [ ] b and (a, b) share the suffix b. In other words, all candidates derived from item b were generated using the backward mining methodology. Among these candidates, a [ ] b and (a, b) are frequent, and therefore, they were added to the tree along with their corresponding information: support count and all positions in the database where these patterns occur. Fig.13. shows the structure of the TSS-tree.

MinitsDays algorithm receives the following inputs:

- Minimum support threshold (min_sup), which is a user-defined parameter.
- 1-Sequence Data structure (1SD), which stores all required information of distinct itemsets during the entire life of the timed sequence database.
- Previous Timed Sequential patterns Suffix tree (Prev_TSS_tree), which a tree contains all timed sequential patterns that are discovered from the previous timed sequence database.
- List of Updated Timed Sequences (Updated_TS_List), which are either inserted into or deleted from the TSDB.
- Type of updating that has been done (Update_Type), which is an insertion or deletion operation.

During the first execution of MinitsDays, the database is treated as a static version. MinitsDays reads the initial list of timed sequences (TS) as the first version of the Timed Sequence Database (TSDB). It then builds the 1-Sequence data structure and the TSS-tree from scratch to discover the initial set of timed sequential patterns. For all subsequent executions, MinitsDays operates on a dynamic TSDB, utilizing previously stored data in the 1-Sequence Data Structure (1-SD) and the previous TSS-tree (Prev_TSS_tree). The following steps are then performed:

1. Determine the type of change (Set Update_Type to either insertion or deletion).
2. Check for existing data structures. If the data structures do not exist, this indicates the algorithm is being executed for the first time, so they must be created.
3. Read the list of updated timed sequences and load the Updated_TS_List. If Update_Type is insertion, call the handle_insertion() function. If it is deletion, call the handle_deletion() function.
4. Modify the content of the data structures according to the type of change. If a new distinct item is introduced through insertion, add it to the 1-sequence data structure. If an existing item becomes infrequent due to deletions, remove it from the TSS-tree along with all its derived patterns, and update accordingly.
5. Traverse the updated TSS- tree to to generate the complete and up-to-date set of timed sequential patterns.

MinitsDays's pseudo-code is presented in Fig.14.

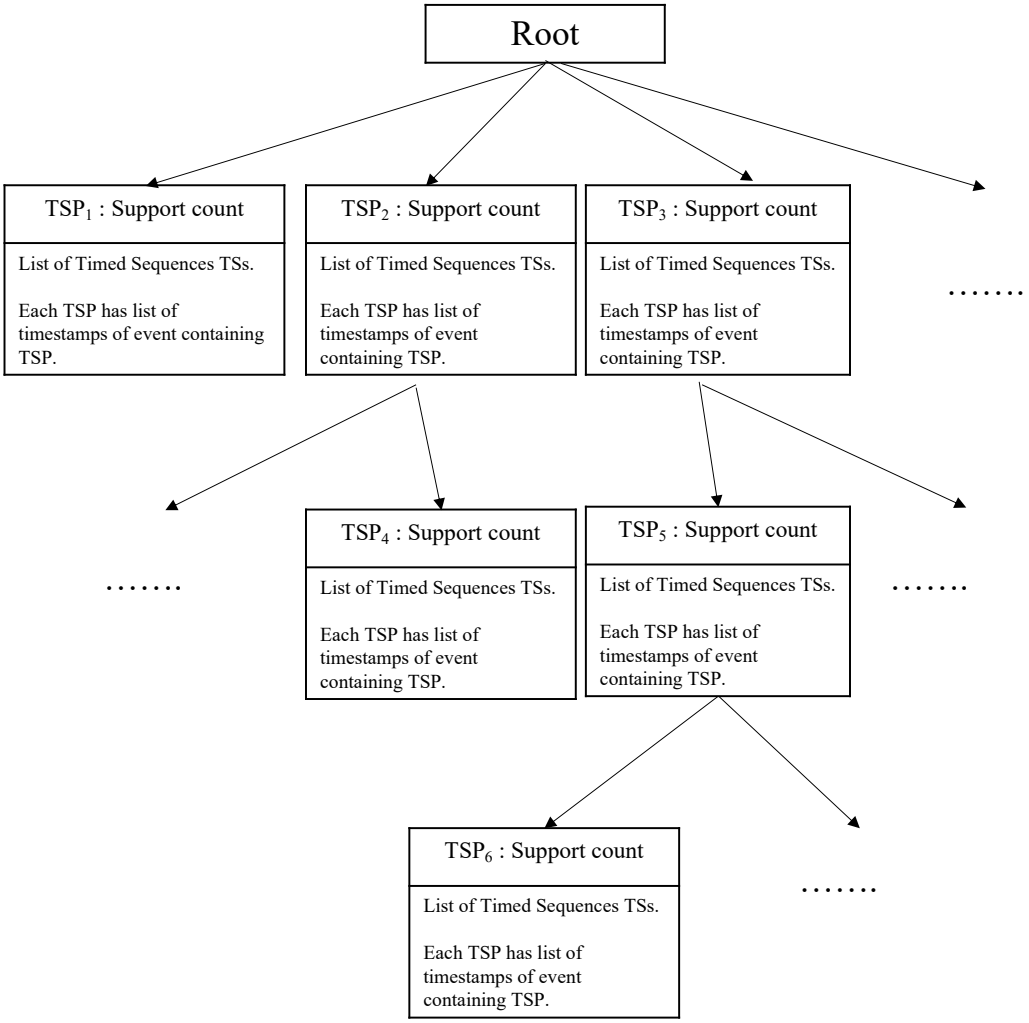

Fig.13. TSS-Tree Structure

Algorithm 1: MinitsDays

Input: min_supp *- portion of required occurrences for each item*
1SD - *1-Sequence data structure*
Prev_TSS_tree *- Previous Timed Sequential Pattern Suffix Tree*
Updated_TS_List *- List of updated timed sequences*
Update_Type *- Type of update (Insertion or Deletion)*

Output: NTSP *– New set of Timed Sequential Patterns*
New_TSS_tree *– New Timed Sequential Suffix Tree*
New_1S *- New 1-Sequence Data structure*

// Possible scenarios
1. ***if*** *1SD.length == 0:*

```
2.          if Prev_TSS_tree.length == 0:
3.                1SD= Updated_TS_List   // Start the construction treating Updated_TS_List as the
             only data available
4.          endif
5.        else:
6.          if Update_Type == "Insertion":
7.            New_1S, New_TSS_tree = Handle_Insertion(1SD, _TS_List)
8.          else if Update_Type == "Deletion":
9.             New_1S, New_TSS_tree = Handle_Deletion(1SD, Updated_TS_List)
10.         endif
11.       endif

12.       for i = 0 to Updated_TS_List.length:
13.         for each event in Updated_TS_List[i]:
14.               for each item in event.values()[1: ] //except the first one because it the  timestamp
15.               if item not in  1SD.keys: // if the item not in 1SD
16.                  1SD.add(item, new Map() )
17.                  1SD[item].support_count = 0
18.                  1SD[item].info = new Map()
19.               endif
20.                 if i not in 1SD[item].keys(): // if item appears in a sequence and it not in the list of
              item
21.                  1SD[item].support_count += 1
22.                  1SD[item].info[i] = new Array()
23.               endif
24.               1SD[item].info[i].add(event[0]) // because it's the actual timestamp
25.             endfor
26.         endfor
27.       endfor

28.       NTSP = new Array()
29.       for key in 1SD.keys():
30.         if 1SD[key].support_count > = min_supp
31.           New_TSS_tree.root.add(key, 1SD[key])
32.           NTSP.add(key)
33.         endif
34.       endfor

          // Add the descendants of the root
35.       New_TSS_tree, 1SD= Get_Descendants(min_supp, Updated_TS_List, New_TSS_tree, 1SD,
      New_TSS_tree.root.children(), NTSP)

36.       return NTSP, New_TSS_tree, New_1S


37.     Algorithm Handle_Insertion(min_supp, 1SD, Prev_TSS_tree, Updated_TS_List)

          //1. Update 1-Sequence Data Structure based on the Updated_TS_List
38.       for each TS in Updated_TS_List:
39.         for each event in TS:
40.           for each item in event:
41.             if item not in 1SD:
42.               Add item to 1SD with count 1 and the corresponding timestamp
43.             else
44.               Update the count and add the new timestamp for the item in 1SD
45.             endif
46.           endfor
47.         endfor
48.       endfor
```

```
        //2. Using the updated 1-Sequence Data Structure, rebuild the TSS tree
49.     New_TSS_tree = Update_TSS_tree(Prev_TSS_tree, 1SD, min_supp)

50.     return 1SD, New_TSS_tree

51.   Algorithm Handle_Deletion(min_supp, 1SD, Prev_TSS_tree, Updated_TS_List)

        //1. Update 1-Sequence Data Structure based on the Updated_TS_List
52.     for each TS in Updated_TS_List:
53.       for each event in TS:
54.         for each item in event:
55.           if item in 1SD:
56.             Decrease the count of item in 1SDby 1 and remove the corresponding timestamp
57.             if item.count == 0:
58.               1SD.remove(item)
59.             endif
60.           endif
61.         endfor
62.       endfor
63.     endfor

        //2. Using the updated 1-Sequence Data Structure, rebuild the TSS tree
64.     New_TSS_tree = Update_TSS_tree(Prev_TSS_tree, 1SD, min_supp)

65.     return 1SD, New_TSS_tree

66.   Algorithm Update_TSS_tree(1SD)
        // Use the previous TSS tree
67.     Updated_TSS_tree = Prev_TSS_tree

        // List of infrequent items to remove
68.     infrequentItems = new Array()

       // Update or add items from 1SD to the TSS tree based on their support count
69.    for item in 1SD.keys:
70.      if 1SD[item].count >= min_supp:
71.        if Updated_TSS_tree.contains(item):
72.          Updated_TSS_tree.update(item, 1SD[item])  // Update existing items
73.        else:
74.          Updated_TSS_tree.add(item, 1SD[item])  // Add new items
75.        endif
76.      else:
77.        if Updated_TSS_tree.contains(item):
78.          infrequentItems.append(item)
79.        endif
80.      endif
81.    endfor

        // Remove infrequent items and their associated patterns
82.     for item in infrequentItems: O(ind_n)
83.       Updated_TSS_tree.remove(item)

          // Now, search for patterns containing the infrequent item and remove them
84.       for pattern in Updated_TSS_tree.patternsContaining(item):
85.         if pattern.support_count < min_supp:
86.           Updated_TSS_tree.remove(pattern)
87.         endif
88.       endfor
89.     endfor
90.     return Updated_TSS_tree
```

```
91.  Algorithm Get_Descendants(min_supp, Updated_TS_List, New_TSS_tree , 1SD,
     add_on_keys, NTSP)

92.      changed = False
93.      for key in add_on_keys:
94.        for last_layer_key in New_TSS_tree.last_layer().keys():
             // Add the new keys in form 'a,b, ...' //
95.          if "," in key:
96.            tmp_map = new Map()
97.            tmp_map.support_count = 0
98.            tmp_map.info = new Map()

99.             for each item in 1SD:
100.             check = true
101.             for each key, timestamps in item.keys_and_values()
102.                 if  TSS_TREE.get(last_layer_key).times ∩ timestamps == ∅ ||
                    last_layer_key == key
103.                   check = false
104.                   break
105.               endif
106.
107.               if check:
108.                 if key not in tmp_map.info.keys():
109.                   tmp_map.key = new Array()
110.                   tmp_map.support_count += 1
111.                 endif
112.                 tmp_map.key.append(tiemstamps)
113.               endif
114.             endfor
115.           endfor
116.           if tmp_map.support_count >= min_supp:
117.             changed = true
118.              NTSP.add (last_layer_key+key)
119.              New_TSS_tree.get(key).add_child(last_layer_key + key, new_map)
120.          endif

           // Add the new keys in form 'a[]b[] ...'
121.        else
122.          key_sequence = new Array()
123.          key_sequence.append(last_layer_key)
124.          key_sequence += key.split([...])
125.          tmp_map = new Map()
126.          tmp_map.support_count = 0
127.          tmp_map.info = new Map()
128.          min = Integer.Max_Value
129.          max = Integer.Min_Value
130.          for each item in 1SD:
131.            for each key, timestamps in item.keys_and_values():
132.              if key in TSS_TREE.get(last_layer_key).keys:
133.                for t in TSS_TREE.get(last_layer_key).get(key).times:
134.                  if t  < min(timestamps):
135.                    if key not in tmp_map.info.keys():
136.                      tmp_map.key = new Array()
137.                      tmp_map.support_count += 1
138.                    endif
139.                    tmp_map.key.append(t + "," + min(timestamps))
140.                    min = minimum(min, min(timestamps)- t)
141.                    max = maximum(max, max(timestamps)- t)
142.                  endif
```

```
143.                         end for
144.                       endif
145.                     endfor
146.                   endfor
147.                   if min != -inf and tmp_map.support_count >= minimum_supp:
148.                     changed = true
149.                         New_TSS_tree.get(key).add_child(last_layer_key + '[' + min + ',' + max +
                        ']' + item, new_map)
150.                     NTSP.add (last_layer_key + '[' + min + ',' + max + ']' + item)
151.                   endif
152.                 endif
153.               endfor
154.             endfor
155.           if not changed:
156.             return New_TSS_tree , 1SD
157.           else
158.             returnGet_Descendants(min_supp,Updated_TS_List,New_TSS_tree,1SD,
       New_TSS_tree .root.children(), NTSP)
159.           endif
```

Fig.14. MinitsDays's pseudo-code

4.2 The Details of MinitsDays Algorithm

The five steps outlined in Section 4.1 will now be explained in detail using the TSDB shown in Fig. 10. During the first execution of the algorithm, both the 1-Sequence Data Structure (1SD) and the previous TSS-tree (Prev_TSS_tree) are empty (corresponding to Steps 1 and 2). The list of updated timed sequences (TS) corresponds to the initial version of the timed sequence database presented in Fig. 10. The algorithm handles this scenario in the same way it handles a static timed sequence database. It begins by scanning the Updated_TS_List and building the 1-Sequence Data Structure (1SD), as shown in Fig. 12. The 1SD includes all distinct items, regardless of whether they are frequent in the TSDB, along with their support counts and occurrence positions (Lines 12–27). Next, the support count of each distinct item is compared against the minimum support threshold (min_sup). Items that meet or exceed min_sup are added to both:

- The New Timed Sequential Patterns (NTSP) set, as 1-item timed sequential patterns, and
- The Timed Sequential Pattern Suffix Tree (TSS-tree) (Lines 28–34), as shown in Fig. 15.

Then, each frequent item is treated as a suffix, and all possible 2-item candidate patterns are generated by calling the function Get_Descendants() (Line 35). Assuming min_sup = 50%, the resulting TSS-tree is shown in Fig. 15, and the NTSP set becomes: NTSP = { <{a}>, <{b}>, <{d}>, <{g}> }.

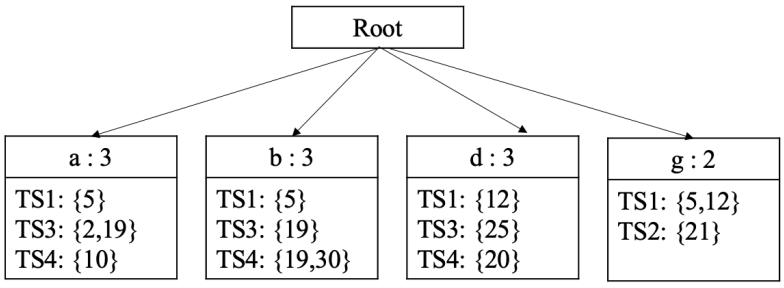

Fig.15. TSS-Tree for 1-Timed Sequential Patterns

Next, to determine which of the 2-candidates, generated by the Get_Descendants() function (Lines 91–115), qualify as Timed Sequential Patterns, the algorithm evaluates each candidate against the minimum support threshold (min_sup) (Line 116). If a candidate is found to be frequent, it is added to both the NTSP and the TSS-tree, but only after its temporal relationship is calculated (Lines 116–129). For example, after generating all 2-candidates from the current NTSP, the following patterns qualify as 2-item timed sequential patterns: <{a} [ ] {b}>, <{a, b}>, <{b} [ ] {d}>, <{a} [ ] {d}>. The temporal relation is then calculated for those patterns that

exhibit a sequential relationship. For instance, the temporal relation for the pattern <{a} [ ] {b}> is computed as follows based on the TSDB shown in Fig. 10:

- In TS3, {a} occurs at timestamp 2, and {b} occurs at timestamp 19. Time difference: 19–2 = 17.
- In TS4, {a} occurs at timestamp 10, and {b} occurs at timestamp 19. Time difference: 19 − 10 = 9.
- TS4 also includes another {b} at timestamp 30 following the same {a} at timestamp 10. Time difference: 30 − 10 = 20
- From these observations: minimum time is 9 and maximum time is 20. Thus, the final timed sequential pattern is: <{a} [9, 20] {b}>.

Accordingly, the NTSP will be = {<{a}>, <{b}>, <{d}>, <{g}>, <{a} [9, 20] {b}>, <{a, b}>, <{b} [1,8] {d}>, and <{a} [6,23] {d}> } and the TSS-tree is updated as shown in Fig.17.

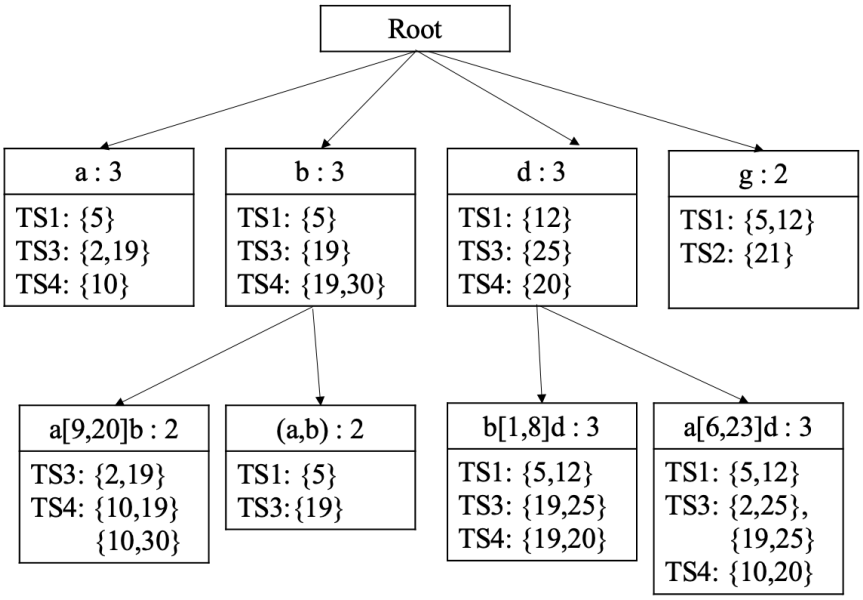

Fig.16: Updated TSS-Tree

Afterward, the algorithm proceeds to generate 3-candidate patterns. It evaluates each candidate to determine whether it qualifies as a 3-item timed sequential pattern by checking its support against the defined min_sup. For each frequent 3-pattern, the algorithm calculates the temporal relations between the itemsets, ensuring accurate time intervals between events. These valid timed sequential patterns are then inserted into the TSS-tree, while infrequent patterns are discarded. The algorithm continues this iterative procedure, generating candidates, filtering by minimum support, calculating temporal relations, and updating the TSS-tree, until no further candidates can be produced. At this termination point, the complete set of valid timed sequential patterns is finalized. The structures of 1-SD and the TSS-tree are now complete. The final TSS-tree is illustrated in Fig.17, and the resulting NTSP (New Timed Sequential Patterns) set is: NTSP = { <{a}>, <{b}>, <{d}>, <{g}>, <{a} [9, 20] {b}>, <{a, b}>, <{a} [1, 8] {d}>, <{b} [6, 23] {d}> .

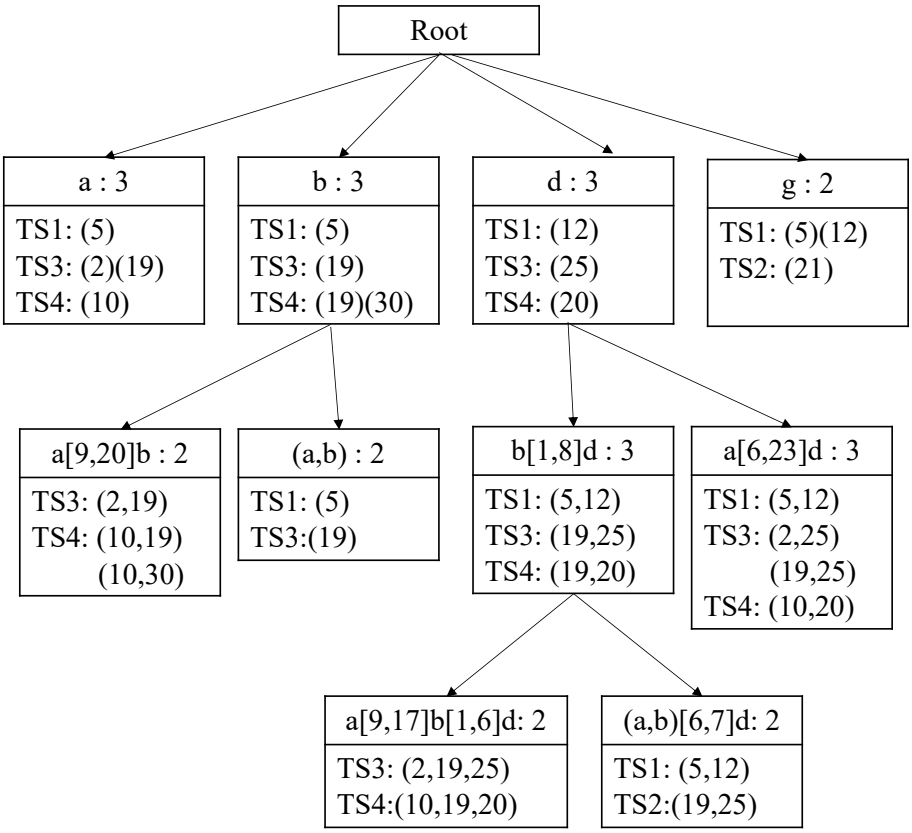

Fig.17. Final TSS-Tree

After the first execution of the algorithm, the 1-SD and TSS-tree are constructed and are no longer empty. Now, consider an update to the Timed Sequence Database (TSDB) as illustrated in Fig.10, where new timed sequences have been added, as shown in Fig.18. The algorithm will handle this update as an insertion operation, and the following inputs are provided:

- Min_sup = 50%
- 1SD as shown in Fig.12.
- Previous TSS-Tree as shown in Fig.17.
- Updated_TS_List: {TS5, TS6, TS7}, containing only the newly added timed sequences
- Update_Type: insertion

MinitsDays begins by reading the Updated_TS_List and the Update_Type and proceeds to update the 1-Sequence Data Structure (1SD) accordingly. Since the type of update operation is insertion, the algorithm calls

ID	Timed Sequence
TS1	<{5, a, b, g}, {12,d, g}>
TS2	<{21, e, g}>
TS3	<{2, a}, {19, a, b}, {25, d}>
TS4	<{10, a}, {19, b, f}, {20,d}, {30, b}>
TS5	<{31, f}, {46, f}>
TS6	<{35, f}, {40, b, f}>
TS7	<{31, b, f}, {34, f}>

Fig.18. UTSDB for Timed Sequence Database Shown in Fig.10.

the function Handle_Insertion() (Line 37) to process the new timed sequences and incorporate them into the existing data structures. For example, All the newly added timed sequences do not contain item a; therefore, its information in the 1SD remains unchanged. However, item b does appear in the Updated_TS_List, so its corresponding information in the 1SD is updated accordingly, as shown in Fig. 19.

a: 3	b: 5	d: 3	e: 1	f: 4	g: 2
TS1: {5}	TS1: {5}	TS1: {12}	TS2: {21}	TS1: {19}	TS1: {5,12}
TS3: {2,19}	TS3: {19}	TS3: {25}		TS5: {31,46}	TS2: {21}
TS4: {10}	TS4: {19,30}	TS4: {20}		TS6: {35,40}	
	TS6: {40}			TS7: {31,34}	
	TS7: {31}				

Fig.19: 1-Sequence Data Structure for Timed Sequence Database Shown in Fig.18.

Next, the TSS-tree must be updated based on different scenarios. In the first scenario, the algorithm checks the support count of each distinct item. If an item becomes infrequent (i.e., its support falls below min_sup), it is removed from the TSS-tree along with all of its child nodes. This follows the Apriori principle, which states that if a pattern is infrequent, all of its supersets must also be infrequent. For example, item g becomes infrequent due to its updated support count of 3, which is below the threshold. As a result, item g is removed from the TSS-tree, as illustrated in Fig. 20.

Another scenario occurs when the support count of an item remains unchanged. In this case, its backward extensions are considered stable, and thus, do not need to be modified. This behavior follows the backward mining strategy described in[14]. For example, the support counts for items a and d remain unchanged. As a result, all of their child nodes in the TSS-tree are preserved without updates, as shown in Fig. 20.

However, if the support count of an item changes, the corresponding node in the TSS-tree, along with all its backward extensions, must be updated. For instance, the support count of item b increases from 3 to 5. As a result, all of b's backward extensions are re-evaluated, and new candidates are generated by calling the Get_Descendants() function (Line 35). MinitsDays then compares the support count of each candidate against the defined min_sup. If a candidate is frequent, the algorithm calculates its temporal relation, using the same procedure applied earlier to the pattern <{a} [ ] {b}>, to determine the appropriate time intervals. In this example, the previous timed sequential patterns with suffix b were infrequent (support count = 2), and this remains unchanged even after the new timed sequences are added to the TSDB. Furthermore, the newly generated candidates also fail to meet the min_sup threshold. Consequently, no new backward extensions are appended to the TSS-tree, and the existing infrequent extensions of item b are removed, as shown in Fig. 20.

The final scenario occurs when a previously infrequent distinct item becomes frequent. In this case, the item must be added to the TSS-tree, and the algorithm must generate all candidate patterns involving this item, starting from 1-candidates to 2-candidates, and so on, until no more candidates can be produced. In our example, item f was initially infrequent with a support count of 1. After the addition of three new timed sequences, its support count increases to 4, which meets the min_sup threshold. MinitsDays responds by updating the TSS-tree through a call to the Update_TSS_tree() function (Line 49). As a result, item f and its corresponding timed sequential patterns are inserted into the TSS-tree, as shown in Fig. 20.

Once no further candidates can be generated, MinitsDays outputs: the updated set of new timed sequential patterns (NTSP), the new TSS-tree, and the updated 1-Sequence Data Structure (1SD). In this example, the final output is: NTSP = { <{a}>, <{b}>, <{d}>, <{f}>, <{a} [1, 8] {d}>, <{b} [6, 23] {d}>, <{b} [6, 23] {d}> }. The updated TSS-tree are illustrated in Fig. 20.

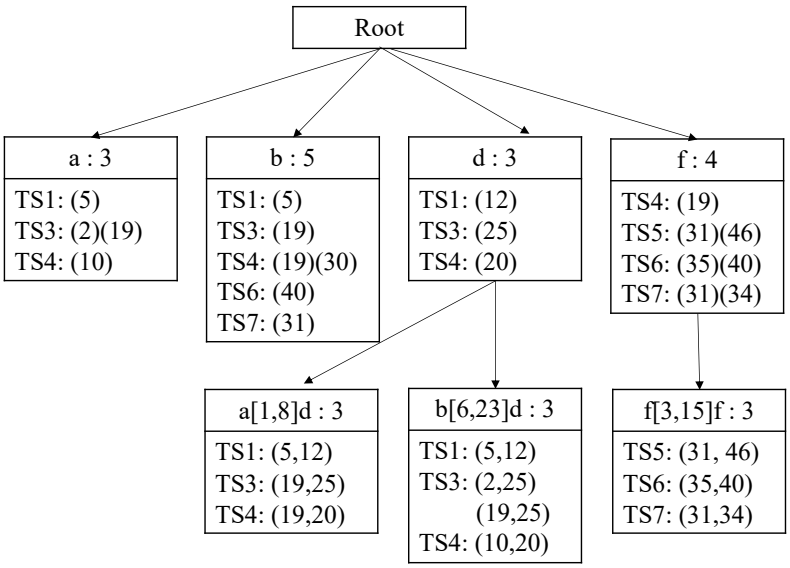

Fig.20. Final TSS-Tree

4.3 Complexity Analysis of MinintsDays

In this section, we present our analysis of the worst-case computational complexity of MinitsDays. Fig. 14 shows the pseudocode of the algorithm. The variables of the time complexity analysis of Minits-AllOcc are listed as follows:

- N: Number of timed sequences in the Timed Sequence Database (TSDB),
- L: the maximum length of a timed sequence (i.e., the maximum number of events per sequence),
- M: Number of distinct individual items across all sequences in TSDB
- U: Number of updated sequences involved in a dynamic modification.

The algorithm operates in two main modes: static mining during the initial execution and dynamic mining during subsequent updates. In the static mining phase, the 1-Sequence Data Structure (1SD) is built (Lines 12 - 27) by scanning each of the N sequences (Line 12), each containing up to L events (Line 13), with each event possibly including up to M distinct items (Line 14). This leads to a time complexity of O (N * L * M) for the initial scan and construction of 1SD. Following this, the algorithm checks all M items to identify frequent patterns of length 1, requiring O(M) time, and stores them in the Timed Sequential Pattern Suffix Tree (TSS-tree), also in O(M) time (Line 28-34). After establishing the base of frequent patterns of length 1, the algorithm invokes a recursive extension process via the Get_Descendants () function (Line 35) to discover longer patterns. In the worst-case scenario, where all items can form patterns of maximum length L, the number of candidates grows exponentially to M^L. For each such candidate, the algorithm may need to scan all N sequences of up to L events to validate support, resulting in a worst-case time complexity of O (M^L * N * L) for this pattern extension phase (Lines 91- 159). Combining these components, the total time complexity for the static execution is O (M^L * N * L), as this term dominates when $L \geq 2$.

In contrast, the dynamic mining phase focuses on efficiently handling updates without reprocessing the entire database. The first step in handling updates involves modifying the 1-Sequence Data Structure (1SD) to reflect the contents of the updated sequences (Lines 38 – 48 or Lines 52 -65). Each of the U sequences contains up to L events, and each event may include up to M items. Therefore, the time complexity for this update step is O(U*L*M). Next, the algorithm must update the Timed Sequential Pattern Suffix Tree (TSS-tree) (Lines 66- 90) to account for changes in pattern support. In the worst-case scenario, where all existing patterns may be affected by the updates, up to M^L patterns (i.e., the maximum number of possible sequential patterns of length L) could require re-evaluation. For each such pattern, the support must be verified by scanning the updated U sequences, each of which may contain up to L events. Consequently, the worst-case time complexity for this step becomes: O(M^L*U*L). Aggregating the two components, the total time complexity for the dynamic update phase is O((U*L*M) + (M^L*U*L)). Since M^L grows exponentially with pattern length L, and $L \geq 2$ is typical in practical applications, the second term dominates. Thus, the total complexity simplifies to O(M^L*U*L). The total complexity is O ((M^L * N * L) + (M^L*U*L)). Since $U \leq N$, the final time complexity is O (M^L * N * L).

This analysis confirms that MinitsDays exhibits exponential complexity in the pattern length L, which is a characteristic trait of sequential pattern mining problems. However, it scales linearly with the number of sequences N, making it practical for large but relatively shallow databases. More importantly, during dynamic updates, the algorithm avoids redundant computation by isolating only the U updated sequences and the Δ affected patterns, ensuring that performance remains efficient in real-world scenarios where updates are typically localized.

5. Performance Analysis

In this section, we describe the experimental environment and present the evaluation results of testing the proposed algorithms on both single-core CPUs (MinitsDays) and multi-core CPUs (MMinitsDays). The experiments were conducted using a combination of real and synthetic datasets, with various parameters taken into consideration to assess the performance and scalability of both implementations.

5.1. Competing Algorithm

Minits-AllOcc [9] was used as a competing algorithm in this experiment. It is designed to mine timed sequential patterns from a static timed sequence database (TSDB). Minits-AllOcc assumes that the sequence database remains unchanged over time. Therefore, if any updates occur in the database content, such as adding or removing sequences, the algorithm must be re-executed from scratch to discover the updated set of timed sequential patterns.

5.2. Experiment Setup

All experiments were conducted on a computer equipped with a 3.20 GHz Intel(R) Core (TM) i7-8700 CPU running 64-bit Windows. Both the MinitsDays and MMinitsDays algorithms were implemented in Java 1.8.

5.3. Datasets and Experimental Parameters

We used both real-world and synthetic datasets in our experiments. The primary real dataset is obtained from Oklahoma Mesonet [28] [29], a world-class network of environmental monitoring stations developed by scientists at the University of Oklahoma (OU) and Oklahoma State University (OSU). Established on January 1, 1994, the network consists of 120 stations, with at least one located in each of Oklahoma's 77 counties. These stations record various environmental parameters and package the data into "observations" every 5 minutes. The observations are then transmitted to a central facility every 5 minutes, 24 hours a day, year-round, providing a rich source of time-ordered, high-frequency data for analysis. The dataset includes the following attributes: county ID, timestamp, air temperature, rainfall, wind, and moisture/humidity, as illustrated in Fig. 21. Prior to analysis, the data was discretized using well-established scales in meteorology, ensuring consistency and meaningful categorization of continuous values.

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	…
…	…

Fig.21. Sequential database for the Oklahoma Mesonet dataset before discretization.

For air temperature, the index heat [30] has 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 [31] 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

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	…
…	…

Fig.22. Sequential database for the Oklahoma Mesonet dataset after discretization.

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 [32]: H1 (uncomfortably dry) [0–20]; H2 (comfortable) [20–60]; and H3 (uncomfortably wet) [60–100]. The results of the discretized sequences are shown in Fig.22.

The synthetic dataset was generated using a tool provided by the SPMF Library[33]. Several parameters were configured to conduct the experiments, which were divided into two categories: static and dynamic. Static parameters remained unchanged throughout all experiments, while dynamic parameters varied from one experiment to another to analyze their impact on performance and pattern discovery. In this study, we focused on four dynamic parameters. The first is the minimum support threshold (min_sup), a user-defined value that determines which patterns are considered frequent in the timed sequence database (TSDB). The second parameter is the number of timed sequences (#seq), which refers to the number of tuples or individual sequences contained within the TSDB. The third dynamic parameter is the length of each timed sequence, defined as the number of events per sequence (#events). The final parameter is the number of items per event (#items). It is important to note that the timestamp is a fixed attribute included in every event. Therefore, when an event is said to have three items, it consists of three data items plus one timestamp. The impact of these four dynamic parameters on the performance and output of the algorithm was thoroughly studied using the synthetic dataset. The values used for each parameter across the experiments are summarized in Table 1. However, for the Oklahoma Mesonet dataset, the only valid dynamic parameter that is shown in Table 1 is the min-sup. Thus, all other three parameters are static. We now explain the range of each parameter and its corresponding default value used in the analysis, as summarized in Table 1. During each experiment, we varied the value of one parameter across its defined range, while assigning the default values to the remaining parameters to maintain consistency. The minimum support threshold (min_sup) ranged from 20% to 80%, with a default value of 50%, which represents the median of the interval. The number of timed sequences ranged from 1 to 100,000, and its default value was set to 50,000, also the median of the range. For the number of events per sequence, the range spanned from 5 to 50, with a default value of 25. Finally, the number of items per event ranged from 1 to 10, with a default of 5, again selected as the median. This configuration allowed us to systematically analyze the impact of each parameter on the algorithm's performance, while ensuring balanced and controlled comparisons.

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 1. Parameter list for the synthetic dataset

5.4. Evaluation Metrics

The evaluation metrics include two measurements: (1) execution time (ET) of algorithms (MinitsDays and MMintisDays); and (2) number of patterns (#patt) generated by these algorithms.

5.5. Experimental Results

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

5.5.1. Accuracy

To validate that MinitsDays consistently produces the same sequential patterns, in terms of both number and content, excluding the temporal relations, we used the PrefixSpan algorithm[5] as a benchmark. PrefixSpan was selected because it is one of the most well-established algorithms for discovering sequential patterns and has been proven to generate complete and correct results. The validation process began by removing the temporal relations from the patterns generated by MinitsDays. These simplified patterns were then compared with those

produced by PrefixSpan to ensure that for every sequential pattern discovered by PrefixSpan, a corresponding pattern existed in the output of both MinitsDays and MMinitsDays. For example, consider the sequential pattern X = < {a} {b} {a, b} >, generated by PrefixSpan, and the timed sequential pattern Y = < {a} [2, 5] {b} [3, 7] {a, b} >, generated by MinitsDays and MMinitsDays. After removing the temporal intervals from Y, we compared it to X. If the order of itemsets in Y matched that in X, they were considered equivalent. However, if the order was different, they were not considered a match. For instance, Z = < {b} [2, 5] {a} [3, 7] {b, a} > did not match X, because the itemset {b} occurred before {a}, altering the sequence. It is important to note that within an individual itemset (e.g., {a, b}), the order of items does not matter, as all items share the same timestamp.

At the conclusion of this validation experiment, we confirmed that both MinitsDays and MMinitsDays generated the exact same set of sequential patterns as PrefixSpan (excluding temporal information), thereby ensuring the correctness and completeness of our proposed algorithms.

5.5.2 Execution time

Data sets	MinitsDays		Minits-AllOcc		MMinitsDays		MMinits-AllOcc	
	ET	# patt	ET	# patt	ET	# patt	ET	# patt
Oklahoma Mesonet	8.376 (min)	3756	21.319 (min)	3756	4.503 (min)	3756	8.604 (min)	3756
Synthetic data	10.681 (min)	3780	27.319 (min)	3780	3.23 (min)	3780	10.825 (min)	3780

Table 2. Average execution time (ET) and #patterns.

The execution time was measured from the moment a dataset was loaded until the algorithm completed the generation of timed sequential patterns. Table 2 presents the average performance of the two algorithms: MinitsDays and MMinitsDays. The results indicate that the execution time (ET) of MMinitsDays was significantly reduced compared to MinitsDays. Specifically, execution time decreased by 50% for the Oklahoma Mesonet dataset, and by up to 90% for the synthetic datasets. These results demonstrate the efficiency and scalability of the multi-core implementation in handling both real and large-scale synthetic data.

5.5.3 Impact of Minimum Support

In this set of experiments, we compared the execution time (ET) and the number of discovered patterns (#patt) across different values of the minimum support threshold (min_sup), using both the Oklahoma Mesonet and synthetic datasets. As shown in Fig. 23 and Fig. 24, we observed that increasing the min_sup value led to a decrease in execution time for all algorithms. This trend is expected, as higher support thresholds result in fewer candidate sequences satisfying the min_sup condition, thereby reducing the number of timed sequential patterns that need to be generated and processed. When dealing with large datasets and a significant number of timed sequential patterns, MMinitsDays consistently outperformed its single-core counterpart, MinitsDays. This highlights the advantage of utilizing multi-core CPUs when working with large Timed Sequence Databases (TSDB), as they offer significant improvements in performance and scalability. Interestingly, the multi-core implementation also proved efficient even at low min_sup values, where a large number of candidate sequences were generated. However, as min_sup exceeded 60%, the execution times of MinitsDays and MMinitsDays began to converge. This is due to the sharp drop in candidate sequences, resulting in fewer patterns being generated and processed. As a result, many threads in MMinitsDays remained idle, causing the algorithm to behave similarly to the single-core MinitsDays. From a performance comparison perspective, Fig. 23 shows that MinitsDays outperformed Minits-AllOcc, achieving a 52% reduction in execution time. Additionally, MMinitsDays outperformed MinitsDays, with a 28% improvement. Similarly, in Fig. 24, MinitsDays again outperformed Minits-AllOcc with a 51% improvement, while MMinitsDays showed a 21% improvement over MinitsDays.

Another key observation was that all algorithms produced the same number of timed sequential patterns across all tested support thresholds. As a result, the pattern count curves in Fig. 23 and Fig.24 were identical and overlapping. As the min_sup threshold increased, the number of discovered patterns decreased, since fewer sequences satisfied the stricter frequency requirement. This reduction is directly attributed to the smaller proportion of timed sequences in the TSDB that contained candidate patterns, as clearly shown in the figures.

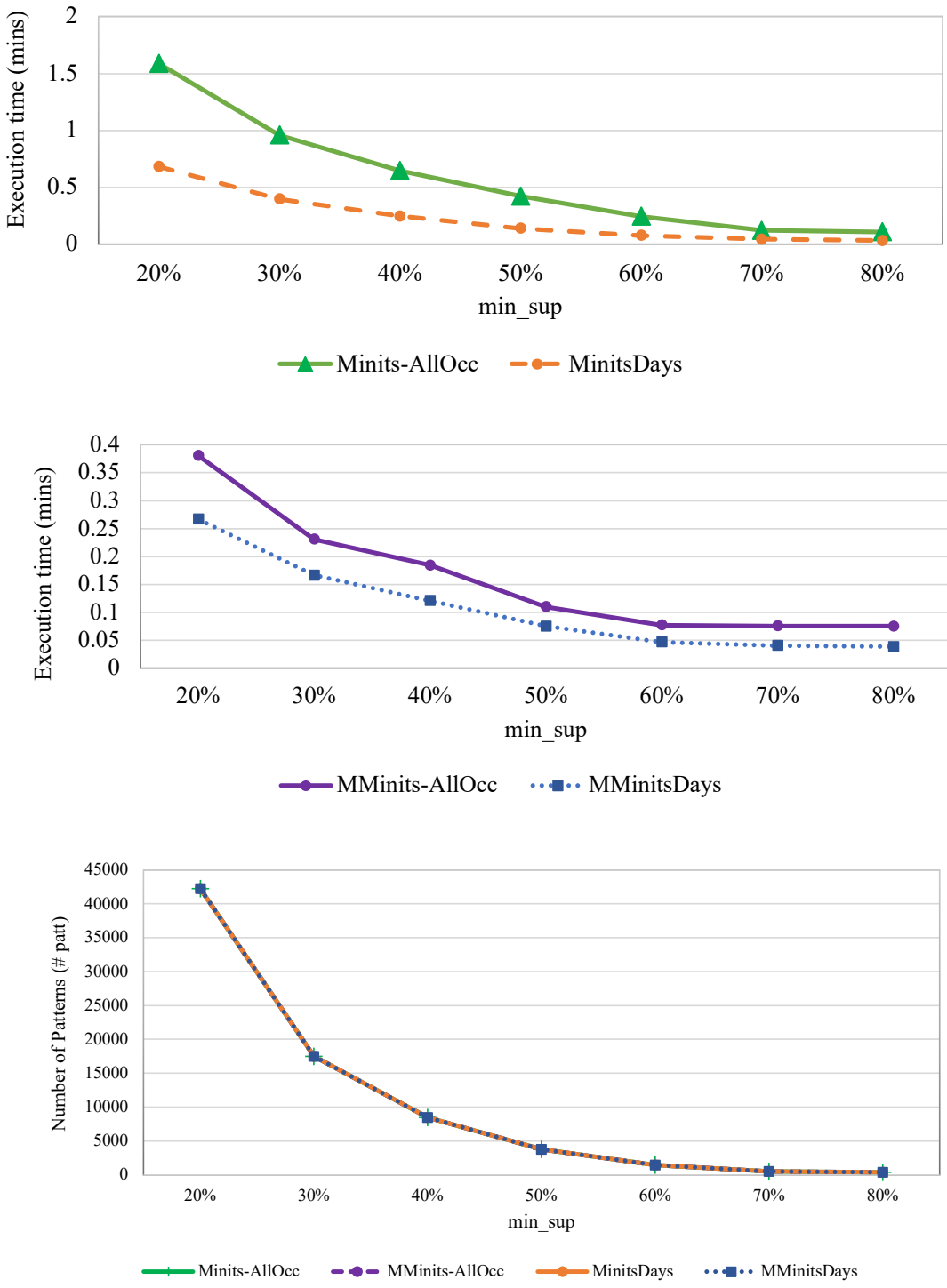

Fig.23. Parameter study (min_sup) for Oklahoma dataset.

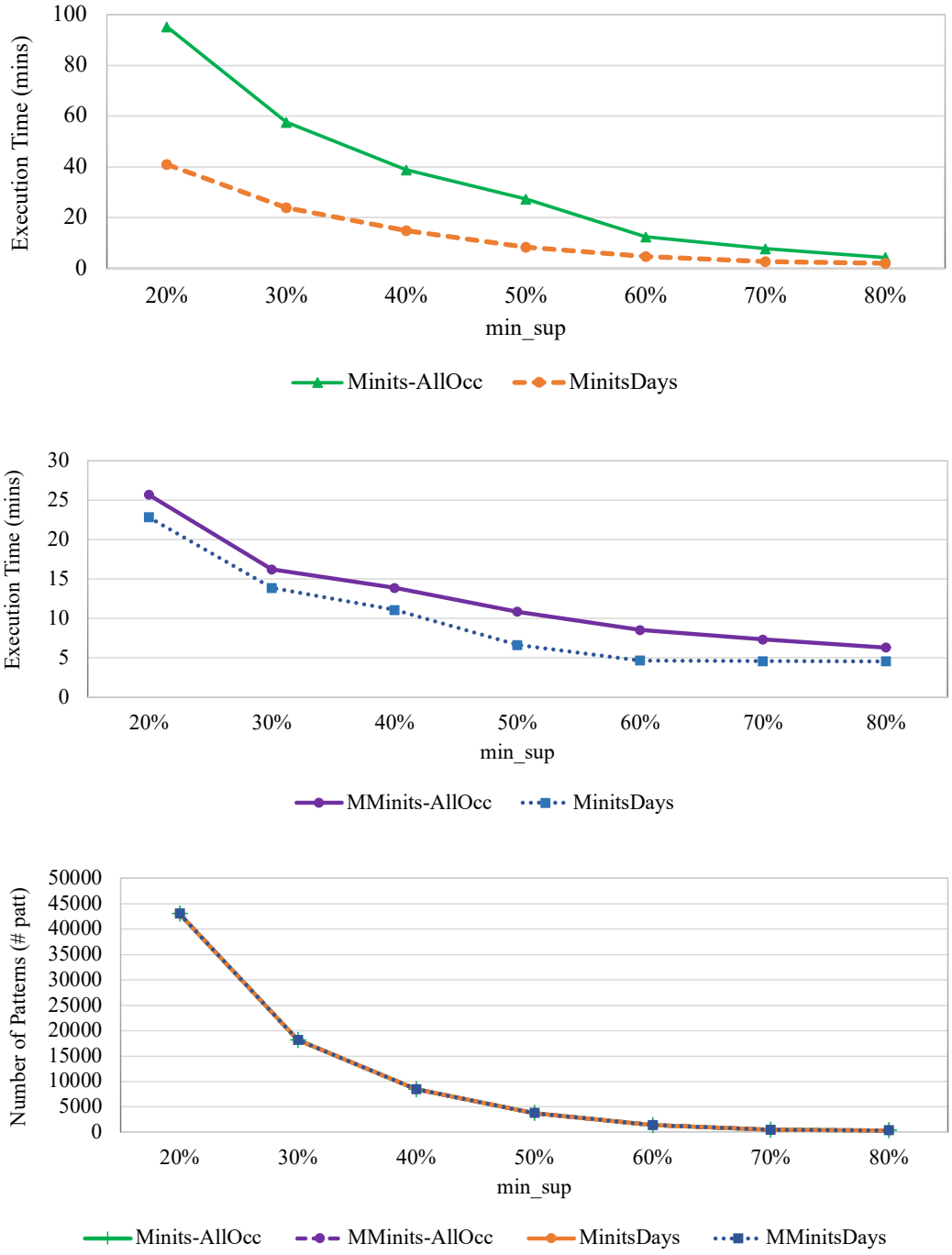

Fig.24. Parameter study(min_sup) for Synthetic dataset.

5.5.4. Impact of the Number of Timed Sequence in the Database

In this set of experiments, we compared the execution time (ET) and the number of discovered timed sequential patterns (#patt) with respect to the number of timed sequences (#seq). As shown in Fig. 25, we observed that as the number of timed sequences increased, the execution time of all algorithms also increased. This is because the algorithms required more time to process the additional timed sequences that were added to the Timed Sequence Database (TSDB) to determine whether they contained any valid timed sequential patterns. We also noted that the number of timed sequential patterns generated by the algorithms increased with the growth in the

number of timed sequences, as illustrated in Fig. 25. MinitsDays outperformed Minits-AllOcc in terms of execution time, achieving a 42% improvement. Additionally, MMinitsDays outperformed MMinits-AllOcc, demonstrating an 86% improvement in execution time. The increase in the number of discovered patterns can be attributed to the higher likelihood of identifying more patterns within the newly added timed sequences that satisfy the minimum support (min_sup) threshold, which was set to 50% by default. As the number of timed sequences in the database increases, the algorithm evaluates whether any new patterns, not previously identified in the original sequences, can now be discovered. These new patterns are then evaluated against the min_sup threshold. It is possible that certain patterns that were infrequent in the original database, due to being supported by too few timed sequences, may now meet the support threshold after the addition of new sequences. Consequently, these patterns are now classified as valid timed sequential patterns, leading to an overall increase in pattern discovery. For instance, in the synthetic dataset, a database with 1,000 timed sequences yielded 3,720 patterns, while a database with 10,000 timed sequences produced 3,780 patterns.

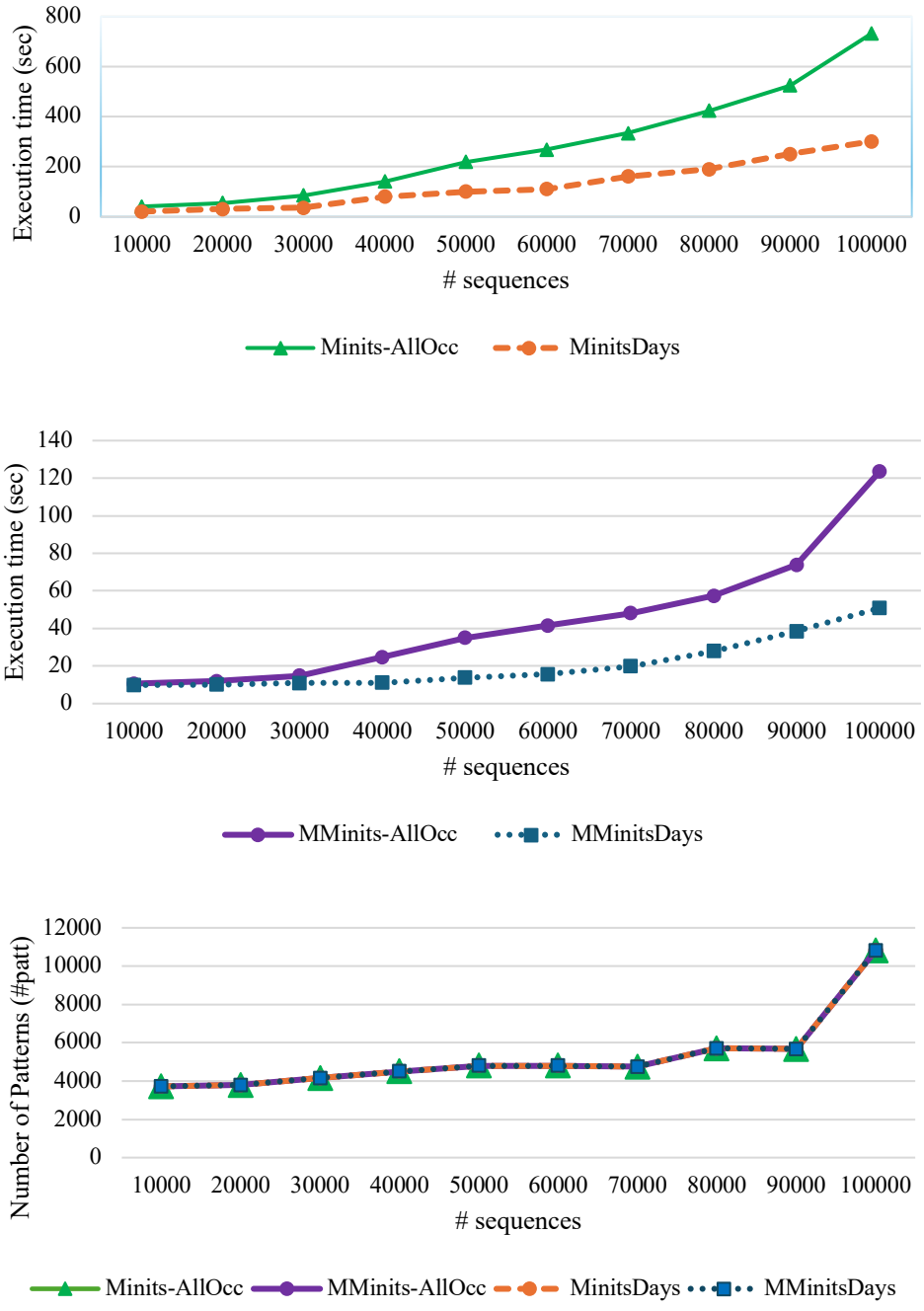

Fig.25. Parameter study (# Sequences) for Synthetic dataset.

5.5.5. Impact of Number of Events per Timed Sequence

Fig. 26 illustrates the impact of the number of events (#events) per timed sequence on both the execution time (ET) and the number of discovered timed sequential patterns (#patterns). In terms of execution performance, MinitsDays outperformed Minits-AllOcc, achieving a 19% improvement in execution time. Additionally, MMinitsDays outperformed MinitsDays, demonstrating a 41% improvement in execution time. There is a clear correlation between the length of a timed sequence and the number of patterns discovered. As the number of events per timed sequence increases, the potential for discovering more complex and longer patterns also increases. This is because the algorithm can extend each pattern up to the maximum length of the timed sequence. If a sequence contains n events, it is possible to discover timed sequential patterns ranging in length from 1 to n. As a result, the execution time increases with the length of timed sequences, due to the growing number of candidate patterns to evaluate, as demonstrated in Fig. 26.

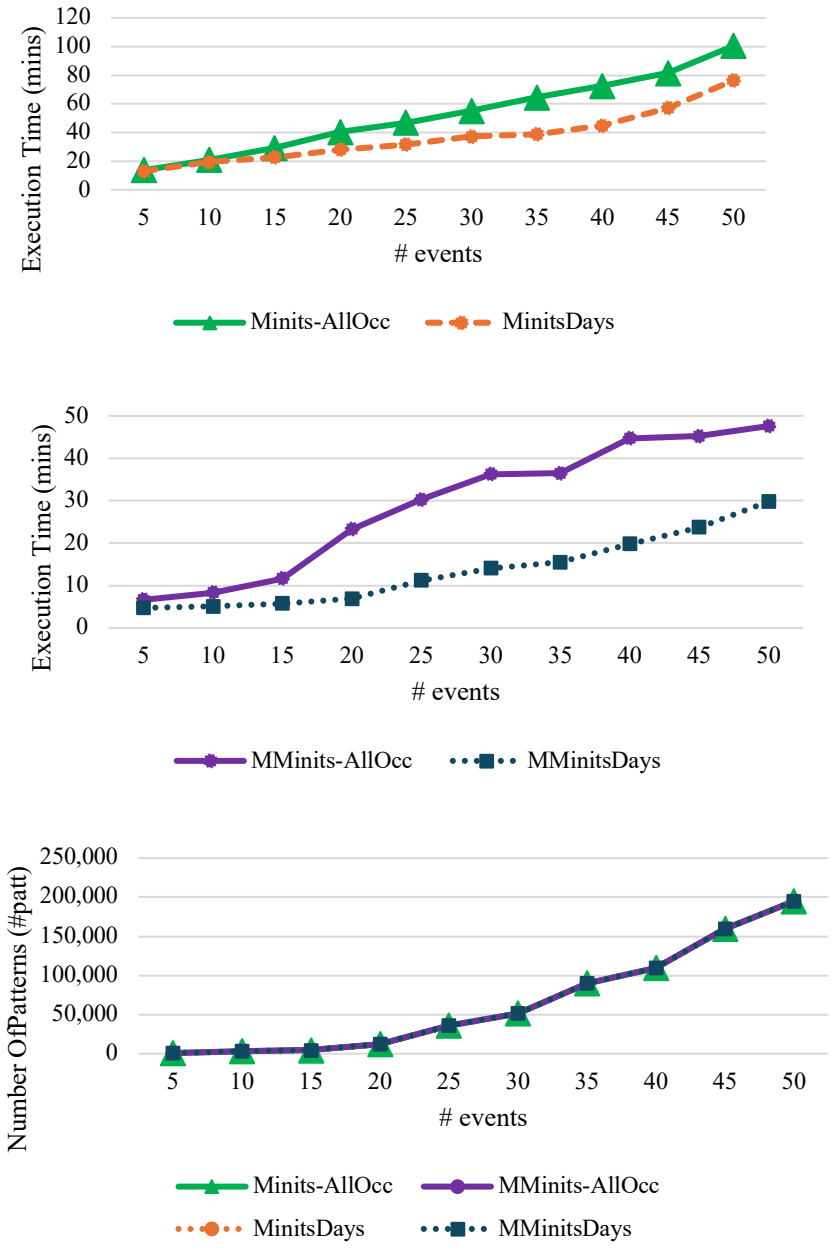

Fig.26. Parameter study (# events) for Synthetic dataset.

5.5.6. Impact of the Modification Ratio

The modification ratio refers to the rate at which changes, such as updates, insertions, or deletions, are made to an existing database of timed sequences. In the final experiment, we compared the execution time (ET) and the number of discovered timed sequential patterns (#patt) under varying modification ratios. Fig. 27 presents the results for both insertion and deletion scenarios. When the figure is read from left to right, it displays the impact of increasing the insertion ratio. Conversely, when read from right to left, it shows the effect of increasing the deletion ratio. As observed in the insertion scenario, increasing the insertion ratio leads to a rise in execution time. This is because the algorithms need additional time to examine the newly inserted timed sequences in the Timed Sequence Database (TSDB) and determine whether they contain new patterns. As the number of timed sequences increases, the likelihood of discovering more patterns that meet the minimum support threshold (min_sup = 50%) also increases. Consequently, this results in a higher number of timed sequential patterns being produced, as shown in Fig. 27. The algorithm must evaluate whether new patterns, not previously observed in the original dataset, emerge because of the inserted sequences. These potential patterns are then compared against the min_sup threshold. Some patterns that were previously infrequent (i.e., not supported by enough sequences) may become frequent after new sequences are added. As a result, the number of newly discovered timed sequential patterns increases.

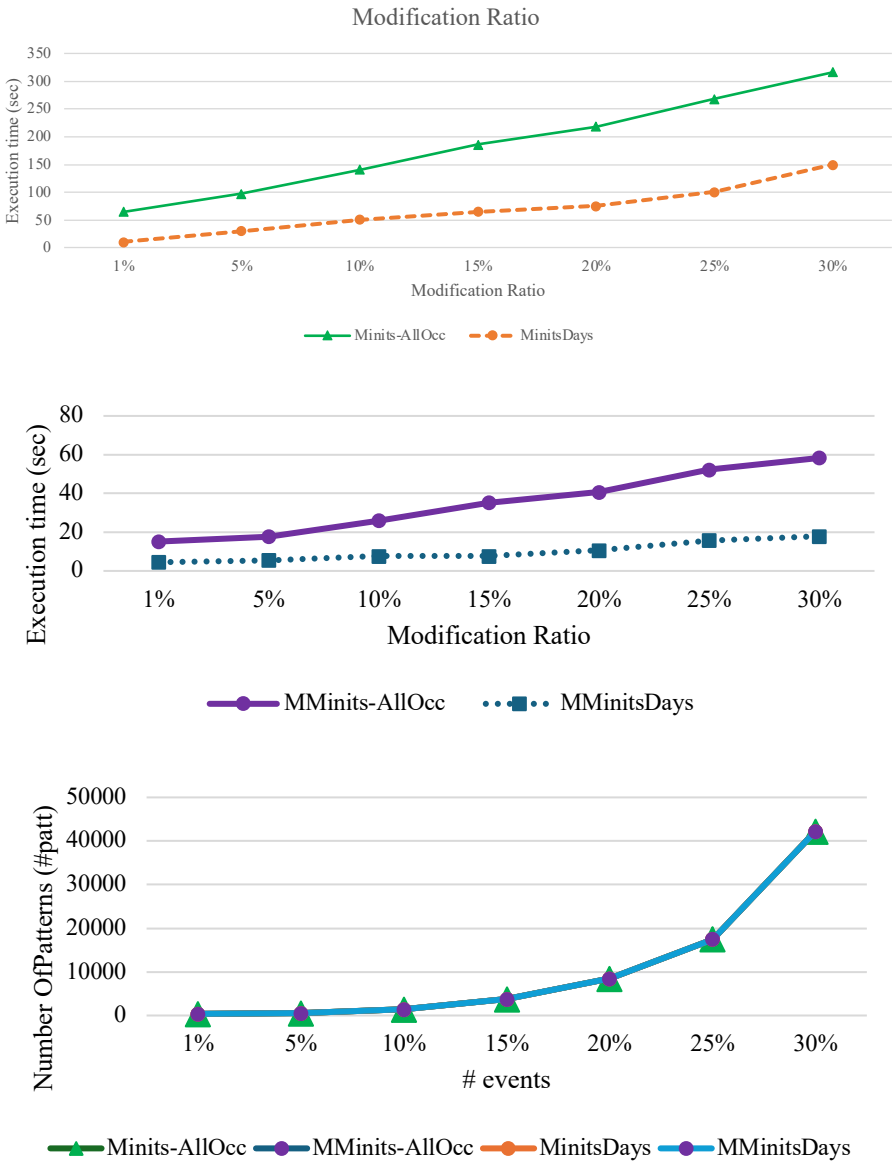

Fig.28. Parameter ET study for Synthetic dataset

In contrast, the deletion scenario exhibits the opposite effect. As sequences are removed from the TSDB, both the execution time and the number of discovered patterns decrease. This reduction occurs because the chances of identifying patterns that meet the min_sup threshold reduce, due to the overall decline in sequence count and, thus, the lower support for previously frequent patterns.

6. Conclusion and Future Work

In this paper, we presented an algorithm called MinitsDays for discovering timed sequential patterns (TSP) from dynamic timed sequence databases (TSDB). The proposed algorithm is designed to mine TSPs incrementally, eliminating the need to re-execute the entire process from scratch after each update to the database. We implemented two versions of the algorithm: the first, MinitsDays, is designed for execution on single-core CPUs, while the second, MMinitsDays, takes advantage of multi-core CPUs for parallel processing. Through a series of experiments, we evaluated the performance of both implementations in terms of accuracy and execution time. The results confirmed that the algorithms produced complete and correct sets of patterns. Additionally, MMinitsDays demonstrated significantly better execution times compared to MinitsDays, particularly in scenarios involving large datasets, long timed sequences, or a high number of items per event.

For future work, we intend to enhance the algorithm to address more complex and practical applications, such as data stream mining, where data continuously evolves over time. Furthermore, we plan to develop a more scalable version of MinitsDays by leveraging modern parallel computing platforms, including GPU and Apache Spark, to better handle the demands of Big Data environments where the number of sequences, events, and items is exceptionally large.

References

[1] Agrawal R, Srikant R. Mining Sequential Patterns. In: *Proceedings of the Eleventh International Conference on Data Engineering*. Taipei, 1995.

[2] Dermy O, Brun A. Can we Take Advantage of Time-Interval Pattern Mining to Model Students Activity? In: *International Educational Data Mining Society*. International Educational Data Mining Society, 2020.

[3] Wahyn Y, Pramono T. Anomaly-based intrusion detection and prevention system on website usage using rule-growth sequential pattern analysis: Case study: Statistics of Indonesia (BPS) website. In: *Proceedings: 2014 International Conference on Advanced Informatics: Concept, Theory and Application (ICAICTA) : Institut Teknologi Bandung, Indonesia : 20-21 August 2014*. [Institute of Electrical and Electronics Engineers], 2014.

[4] Srikant R, Agrawal R. Mining Sequential Patterns: Generalizations and Performance Improvements. In: *Springer Berlin Heidelberg*. Berlin, Heidelberg: International conference on extending database technology, 1996, pp. 1–17.

[5] Pei J, Computer Society I, Han J, et al. Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach. *Transactions on knowledge and data engineering* 2004; 16: 1424–1440.

[6] Han J, Mei-Chun Hsu, Umeshwar Dayal, et al. FreeSpan- frequent pattern-projected sequential pattern mining. *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining* 2000; 355–359.

[7] Zaki MJ. SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine learning, 2001, pp. 31–60.

[8] Karsoum S, Gruenwald L, Barrus C, et al. Using Timed Sequential Patterns in the Transportation Industry. Los Angeles, CA, USA: IEEE International Conference on Big Data, 2019, pp. 3573–3582.

[9] Karsoum S, Barrus C, Gruenwald L, et al. Minits-AllOcc: An Efficient Algorithm for Mining Timed Sequential Patterns. Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2021, pp. 668–685.

[10] Lin M-Y, Lee S-Y. *Incremental Update on Sequential Patterns in Large Data bases*.

[11] Parthasarathy S, Zakit MJ, Ogihara M, et al. Incremental and Interactive Sequence Mining. Proceedings of the eighth international conference on Information and knowledge management, 1999, pp. 251–258.

[12] Zheng Q, Xu K, Ma S, et al. The Algorithms of Updating Sequential Patterns. arXiv preprint cs/0203027, 2002.

[13] Zhang M, Kao B, Cheung D, et al. Efficient algorithms for incremental update of frequent sequences. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Springer Verlag, 2002, pp. 186–197.

[14] Lin MY, Hsueh SC, Chan CC. Incremental discovery of sequential patterns using a backward mining approach. In: *Proceedings - 12th IEEE International Conference on Computational Science and Engineering, CSE 2009*. 2009, pp. 64–70.
[15] Wang L, Gui L, Xu P. Incremental sequential patterns for multivariate temporal association rules mining. *Expert Syst Appl*; 207. Epub ahead of print 30 November 2022. DOI: 10.1016/j.eswa.2022.118020.
[16] Guyet T, Zhang W, Bifet A. Incremental Mining of Frequent Serial Episodes Considering Multiple Occurrences. *International Conference on Computational Science* 2022; 460–472.
[17] Saleti S, R.B.V. S. A MapReduce solution for incremental mining of sequential patterns from big data. *Expert Syst Appl* 2019; 133: 109–125.
[18] Huang JW, Tseng CY, Ou JC, et al. A general model for sequential pattern mining with a progressive database. *IEEE Trans Knowl Data Eng* 2008; 20: 1153–1167.
[19] Chakrabarti S, Saha HN, University of Nevada, et al. *Parallel Progressive Sequential Pattern (PPSP) Mining*.
[20] Mhatre A, Verma M, Toshniwal D. Extracting sequential patterns from progressive databases: A weighted approach. In: *2009 International Conference on Signal Processing Systems, ICSPS 2009*. 2009, pp. 788–792.
[21] Huang J-W, Lin S-C, Chen M-S. *DPSP: Distributed Progressive Sequential Pattern Mining on the Cloud*.
[22] Jamshed A, Mallick B, Bharti RK. An Analysis of Sequential Pattern Mining Approach for Progressive Database by Deep Learning Technique. In: *Proceedings - 2022 6th International Conference on Intelligent Computing and Control Systems, ICICCS 2022*. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 1409–1415.
[23] Giannotti F, Dino Pedreschi, Fabio Pinelli, et al. Trajectory Pattern Mining. Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, 2007, pp. 330–339.
[24] Le HH, Yamada T, Honda Y, et al. Methods for Analyzing Medical-Order Sequence Variants in Sequential Pattern Mining for Electronic Medical Record Systems. *ACM Trans Comput Healthc*; 4. Epub ahead of print 30 March 2023. DOI: 10.1145/3561825.
[25] Ghorbani M, Abessi M. A New Methodology for Mining Frequent Itemsets on Temporal Data. *IEEE Trans Eng Manag* 2017; 64: 566–573.
[26] Le HH, Edman H, Honda Y, et al. Fast Generation of Clinical Pathways including Time Intervals in Sequential Pattern Mining on Electronic Medical Record Systems. In: *Proceedings - 2017 International Conference on Computational Science and Computational Intelligence, CSCI 2017*. Institute of Electrical and Electronics Engineers Inc., 2018, pp. 1726–1731.
[27] Le HH, Yamada T, Honda Y, et al. Effects of mining parameters on the performance of the sequence pattern variants analyzing method applied to electronic medical record systems. In: *ACM International Conference Proceeding Series*. Association for Computing Machinery, 2019. Epub ahead of print 2 December 2019. DOI: 10.1145/3366030.3366074.
[28] Brock Fred V, Michael D. Eilts, Johnson, et al. The Oklahoma Mesonet A technical overview. *J Atmos Ocean Technol* 1995; 12: 5–19.
[29] McPherson RA, Fiebrich CA, Crawford KC, et al. Statewide monitoring of the mesoscale environment: A technical update on the Oklahoma Mesonet. *J Atmos Ocean Technol* 2007; 24: 301–321.
[30] US Department of Commerce NNWS. What is the heat index?, https://www.weather.gov/ama/heatindex#:~:text=The%20heat%20index%2C%20also%20known,for%20the%20human%20body's%20comfort. (accessed 2 April 2024).
[31] The 100-Year Flood | U.S. Geological Survey, https://www.usgs.gov/special-topics/water-science-school/science/100-year-flood?qt-sci%ADence_center_objects=0#qt-science_center_objects (accessed 2 April 2024).
[32] US Department of Commerce NNWS. Dew Point vs Humidity, https://www.weather.gov/arx/why_dewpoint_vs_humidity#:~:text=The%20dew%20point%20is%20the,water%20in%20the%20gas%20form. (accessed 2 April 2024).
[33] SPMF: A Java Open-Source Data Mining Library, https://www.philippe-fournier-viger.com/spmf/ (accessed 1 April 2024).