UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DISCOVERING TIMED SEQUENTIAL PATTERNS FROM STATIC AND DYNAMIC TIMED SEQUENCE DATABASES

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

SOMAYAH MOHAMMED KARSOUM
Norman, Oklahoma
2025

DISCOVERING TIMED SEQUENTIAL PATTERNS FROM STATIC AND DYNAMIC TIMED SEQUENCE DATABASES

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Le Gruenwald, Chair

Dr. Mohammed Atiquzzaman

Dr. Qi Cheng

Dr. Jinsong Pei

Dr. Xiangming Xiao

© Copyright by SOMAYAH MOHAMMED KARSOUM 2025
All Rights Reserved.

ACKNOWLEDGMENTS

To

My

Family,

Advisor,

Committee members,

Professors,

Friends,

and

Unsung Heroes

Thank you to each of you for being an integral part of this journey!

TABLE OF CONTENTS

ACKNOWLEDGMENTS ..... **iv**
TABLE OF CONTENTS ..... **v**
LIST OF TABLES ..... **viii**
LIST OF FIGURES ..... **ix**
TABLE OF NOTATIONS ..... **xii**
ABSTRACT ..... **xiii**
CHAPTER 1. INTRODUCTION **1**
1.1 Sequential Pattern Mining 3
1.2 Timed Sequential Pattern Mining 4
1.3 Parallel Platform: Multiprocessors 5
1.4 Application Areas of Timed Sequential Pattern Mining 6
1.5 Research Issues 11
1.5.1 Research Issues for Mining Sequential Patterns 11
1.5.2 Additional Research Issues for Mining Timed Sequential Patterns in Static Timed Sequence Databases 15
1.5.3 Additional Research Issue for Mining Timed Sequential Patterns in Dynamic Timed Sequence Databases 21
1.6 Research Objective 22
1.7 Contributions 22
1.8 Organization of This Dissertation 26
CHAPTER 2. LITERATURE REVIEW **28**
2.1 Variations of the Sequential Pattern Mining Problem 28
2.2 Literature Review of Incorporating Temporal Relation in Mining Sequential Patterns Algorithms 34
2.3 Literature Review of Mining Sequential Patterns from Dynamic Databases 50
2.3.1 Mining Sequential Patterns in Incremental Databases 50
2.3.2 Mining Sequential Patterns in Progressive Databases 60
2.4 Summary 69
CHAPTER 3. THE PREPROCESSING APPROACHES AND PROPOSED ALGORITHMS **71**
3.1 Preliminaries 71
3.1.1 Sequential Pattern Mining Problem Definitions 71

3.1.2 Timed Sequential Patterns Mining Problem Definitions 72
3.2 Data Preprocessing 75
3.2.1 Introduction to Trajectory Sequential Patterns 76
3.2.2 Trajectory Segmentation Approaches 79
3.2.2.1 Density-Based Trajectory Segmentation 79
3.2.2.2 Grid-Based Trajectory Segmentation 82
3.2.3 Discovering Frequent Trajectory Sequential Patterns 84
3.3 Minits+: An Algorithm for Discovering Timed Sequential Patterns in Static Discretized Timed Sequence Database 86
3.3.1 Motivation of Minits+ 86
3.3.2 Overview of Minits+ 88
3.3.3 Details of the Minits+ 91
3.3.4 The Proposed Enhancement 93
3.4 The Minits-AllOcc: An Algorithm for Mining Timed Sequential Patterns for All-Occurrences in Static Discretized Timed Sequence Databases 97
3.4.1 Motivation of Minits-AllOcc 97
3.4.2 Overview of Minits-AllOcc 100
3.4.3 Details of the Minits-AllOcc 104
3.4.4 Proposed Enhancement 109
3.5 MinitsDays: An Algorithm for Mining Timed Sequential Patterns in Dynamic Discretized Timed Sequence Databases 113
3.5.1 Motivation of the MinitsDays 114
3.5.2 Overview of MinitsDays 119
3.5.3 Details of the MinitsDays 130
3.6 Summary 137
CHAPTER 4. PERFORMANCE ANALYSIS 138
4.1 Theoretical Analysis of the Proposed Algorithms 138
4.1.1 Time Complexity Analysis and proof of correctness of Minits+ 138
4.1.2 Time Complexity and proof of correctness of Minits-AllOcc 144
4.1.3 Time Complexity and proof of correctness of MinitsDays 151
4.2 Experimental Analysis 157
4.2.1 Experimental Analysis of Trajectory Segmentation 157
4.2.1.1 Experimental Setup 157
4.2.1.6 Experimental Results 163

4.2.2 Experimental Analysis of Minits+ 177
4.2.2.1 Experimental Setup 178
4.2.2.2 Experimental Results 181
4.2.3 Experimental Analysis of Minits-AllOcc 188
4.2.3.1 Experimental Setup 188
4.2.3.2 Experimental Results 192
4.2.4 Experimental Analysis of MinitsDays 202
4.2.4.1 Experimental Setup 202
4.2.4.2 Experimental Results 205
4.3 Summary 217
CHAPTER 5. CONCLUSIONS AND FUTURE WORK 218
5.1 Summary of the Performance Evaluation Results 219
5.1.1 Summary of the Results of Minits+ 219
5.1.2 Summary of the Results of Minits-AllOcc 221
5.1.3 Summary of the Results of MinitsDays 224
5.2 Future Research 227
REFERENCES 229

LIST OF TABLES

Table 2.1:	Comparison of exisitng times sequential patterns …….	42
Table 2.2:	Feature comparison of discovering Timed/Dynamic Sequential Patterns techniques ….....	68
Table 4.1:	Cell size vs. SSE measurement for GeoLife dataset ……..	160
Table 4.2:	List of parameters for experiments ……..……...	161
Table 4.3:	Average of performance results …….	163
Table 4.4:	Parameters list for T-Drive ……	180
Table 4.5:	Parameters list for synthetic dataset ……...	180
Table 4.6:	Average Execution times and # patterns of the algorithms on the T-Drive and synthetic dataset ……	182
Table 4.7:	Parameter list for the synthetic dataset ……..	191
Table 4.8:	Average execution time (ET) and (#patt) ……..	193
Table 4.9:	Average execution time (ET) and (#patt) ……..	205

LIST OF FIGURES

Figure 1.1:	Shared memory vs. shared nothing architectures	6
Figure 1.2:	Patients' historical health information and discretized data ……	7
Figure 1.3:	Patients' historical health information and discretized data ……	10
Figure 1.4:	An illustration of the Apriori principle……..…….....	14
Figure 1.5:	Timed Sequence Database for patients…….…….	17
Figure 1.6:	All possible occurrences of the symptom {T1} {BP3} in P4..	19
Figure 1.7:	Dissertation Outline ……	27
Figure 2.1:	Input and Output for Sequential Pattern Mining……..	29
Figure 2.2:	Input and Output for Fuzzy Sequential Pattern Mining……..	30
Figure 2.3:	Input and Output for High- Utility Sequential Pattern Mining……	31
Figure 2.4:	Input and Output for Periodic-Frequent Pattern Mining……..	32
Figure 2.5:	Input and Output for Spatio-Temporal Mining……	33
Figure 3.1:	An example of Discretized Timed Sequence Database DTSDB …….	73
Figure 3.2:	All-time Occurrence of A in TS4 ……	74
Figure 3.3:	Drifting issue of GPS …….	78
Figure 3.4:	Density-based Trajectory Segmentation …….	82
Figure 3.5:	Grid-based Trajectory Segmentation ……..	83
Figure 3.6:	Trajectory Sequence Database ……...	84
Figure 3.7:	Vertical Representation of the Trajectory Sequence ……..	84
Figure 3.8:	Id-lists for some 2-sequences …….	85
Figure 3.9:	Sensors' historical weather information and discretized data …….	87
Figure 3.10:	Pseudo-code of the Minits+ Algorithm ……..	91
Figure 3.11:	Simplified Discretized Timed Sequence Database …….	91
Figure 3.12:	Projected database for prefix T1 - S|T1 …….	92
Figure 3.13:	Pseudo projection database for prefix T1 - S|T1 ……	95
Figure 3.14:	The mechanism of multi-core CPUs of Minits+ ……	96
Figure 3.15:	All possible occurrences of the measurement {T1} {W1} in S2 ……	98
Figure 3.16:	Occurrence Tree Data Structure …….	101
Figure 3.17:	An Example of Discretized Timed Sequence Database …….	101
Figure 3.18:	An O-tree for the Sequences <{a}> and <{a}{a}> in TS3 ……	103
Figure 3.19:	Merging O-trees of <{a}> and <{b}> to generate <{a,b}>-forest and <{a}[9,20]{b}>-forest ……	103
Figure 3.20:	Pseudo-code of the Minits-AllOcc Algorithm ……	105
Figure 3.21:	Merging <{a}[9, 20]{b}>-forest and <{d}> to generate <{a}[9, 17]{b}[1, 6] {d}>-forest …….	110
Figure 3.22:	Pruning the original < {a} [9,20] {b} >-forest and < {a, b} >-forest …….	111
Figure 3.23:	Frequency Matrix for <{a}> ……...	112

Figure 3.24: Multi-core Implantation ……. 113
Figure 3.25: Discretized Timed Sequence Database for patients …… 114
Figure 3.26: Updated Discretized Timed Sequence Database for patients …….. 115
Figure 3.27: Discretized Timed Sequence Database for Sensors …… 115
Figure 3.28: Updated Discretized Timed Sequence Database for Sensors ……. 116
Figure 3.29: Discretized Timed Sequence Database for stocks …….. 116
Figure 3.30: Updated Discretized Timed Sequence Database for stocks …… 117
Figure 3.31: Updated Discretized Timed Sequence Database for Case 4.1 …… 117
Figure 3.32: New Discretized Timed Sequence Database for Case 4.1 …….. 118
Figure 3.33: Updated Discretized Timed Sequence Database for Case 4.2 …… 119
Figure 3.34: 1-sequence data structure for an Item ……. 121
Figure 3.35: Dynamic Discretized Timed Sequence Database ……... 121
Figure 3.36: 1-sequence data structure for Discretized Timed Sequence Database shown in Figure 3.35 …….. 121
Figure 3.37: TSS-Tree Structure ……. 123
Figure 3.38: Pseudo-code of the MinitsDays Algorithm ……. 125
Figure 3.39: TSS-Tree for 1-Timed Sequential Patterns ……. 131
Figure 3.40: Updated TSS-Tree …….. 132
Figure 3.41: Final TSS-Tree ……... 133
Figure 3.42: UTSDB for Discretized Timed Sequence Database shown in Figure 3.35 134
Figure 3.43: 1-Sequence Data Structure for Discretized Timed Sequence Database Shown in Figure 3.42 ……. 134
Figure 3.44: Final TSS-Tree ……... 136
Figure 4.1: Best value of (ε) for deer dataset …… 159
Figure 4.2: Impact of ε on Execution time (ET) …… 166
Figure 4.3: Impact of ε on number of patterns …….. 167
Figure 4.4: Impact of MinPts on Execution Time (ET) …… 169
Figure 4.5: Impact of MinPts on Number of Patterns …… 170
Figure 4.6: Impact of cell size on Execution Time (ET) …… 171
Figure 4.7: Impact of cell size on Number of Patterns …….. 172
Figure 4.8: Impact of the number of trajectories on Execution Time (ET) …….. 173
Figure 4.9: Impact of the number of trajectories Number of Patterns …….. 174
Figure 4.10: Impact of minimum support (min-sup) on Execution time (ET) …… 175
Figure 4.11: Impact of minimum support (min_sup) on Number of Patterns ……. 177
Figure 4.12: Sequential database for the T-Drive dataset before discretization by DBSCAN …… 180
Figure 4.13: Sequential database for the T-Drive dataset after discretization by DBSCAN ……. 181

Figure 4.14: Impact of min_sup on execution time (ET) and number of patterns (#patt) using T-Drive dataset ……. 184
Figure 4.15: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Synthetic dataset ……... 185
Figure 4.16: Impact of # sequences on execution time (ET) using Synthetic dataset …. 187
Figure 4.17: Impact of # sequences on number of patterns (#patt) using Synthetic dataset ……. 187
Figure 4.18: Sequential database for the Oklahoma Mesonet dataset before discretization …….. 188
Figure 4.19: Sequential database for the Oklahoma Mesonet dataset after discretization 189
Figure 4.20: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Oklahoma Mesonet dataset ……... 194
Figure 4.21: Impact of min_sup on execution time (ET) and number of patterns (#patt) using T-Drive dataset ……. 195
Figure 4.22: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Synthetic dataset ……... 196
Figure 4.23: Impact of number of sequences on execution time (ET) and number of patterns (#patt) using Synthetic dataset …….. 198
Figure 4.24: Impact of the number of events on execution time (ET) and number of patterns (#patt) using Synthetic dataset …….. 199
Figure 4.25: Impact of the number of the items on execution time (ET) and number of patterns (#patt) using Synthetic dataset ……. 201
Figure 4.26: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Oklahoma Mesonet dataset ……... 208
Figure 4.27: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Synthetic dataset ……... 209
Figure 4.28: Impact of number of sequences on execution time (ET) and number of patterns (#patt) using Synthetic dataset …….. 211
Figure 4.29: Impact of the number of events on execution time (ET) and number of patterns (#patt) using Synthetic dataset ……. 213
Figure 4.30: Impact of the modification ratio on execution time (ET) and number of patterns (#patt) using Synthetic dataset …….. 216

TABLE OF NOTATIONS

Notation	Meaning
X	Set of items
x_l	An item
I	An itemset (a set of items)
A, B, s	Sequences
S	A sequence database
sup (A)	A support of sequence A
min_sup	A user-defined threshold called minimum support
e = (I, t)	An event, where *I* is an itemset that occurs at the timestamp *t*
DTSDB	A Discretized Timed Sequence Database
TS	A Timed Sequence in DTSDB
Δ	Delta, difference between the timestamps of two different events in a timed sequence
TSP	A Timed Sequential Pattern
TSPs	A set of complete Timed Sequential Patterns
NTSP	A New set of complete Timed Sequential Patterns

ABSTRACT

Mining sequential patterns is one of the data mining tasks that aims to find the subsequences that frequently occur in a specific timestamp order within a sequence database. For example, a patient's health history database is a type of sequence database that records medical events (such as diagnoses and treatments) in chronological order. Discovering sequential patterns in such a database can help predict the symptoms of an impending heart attack, assist healthcare providers with diagnosis, enable timely treatment, and facilitate early intervention in critical cases.

However, clinical decision-making requires not only pattern identification but also precise temporal forecasting of when symptoms may occur. This necessitates the discovery of sequential patterns that incorporate temporal information, referred to as timed sequential patterns (TSP). For example, patients typically develop high cholesterol first; within three to four days, this progresses to high blood pressure, followed by elevated body temperature after another one to two days. Ultimately, a heart attack may occur within the following one to two months. Identifying interesting, useful, and temporally accurate patterns is valuable for many real-world applications, such as illness symptom prediction, weather forecasting, and transportation and arrival time analysis.

While several approaches have been proposed for mining sequential patterns, most overlook the temporal relationships between the events in the subsequence. As a result, they fail to discover patterns that include this critical temporal information. To address this, some studies have incorporated temporal information into sequential patterns; however, challenges remain. For instance, some approaches require users to provide time-related inputs to identify temporal relations, rather than enabling algorithms to discover them automatically. Others consider only the first occurrence of a pattern when calculating temporal relations, ignoring other possible

occurrences in the database. Furthermore, most existing sequential pattern mining algorithms assume static databases, despite the inherently dynamic nature of real-world data, which undergoes frequent insertions, deletions, and modifications. The dynamic nature complicates the efficient discovery of a complete set of sequential patterns without re-scanning the entire database. The problem is compounded by the accelerating growth of databases due to the widespread use of tracking devices, increased social media activity, and the general rise in stored data. Therefore, there is an increasing need for efficient and scalable algorithms that leverage parallelism to manage this rapid expansion effectively.

This dissertation addresses these challenges through three novel algorithms: Minits+, Minits-AllOcc, and MinitsDays. Minits+ and Minits-AllOcc are designed to identify frequent timed sequential patterns in static sequence databases. While Minits+ does not consider all possible occurrences of a pattern, Minits-AllOcc does. MinitsDays, on the other hand, efficiently discovers frequent timed sequential patterns in dynamic sequence databases without requiring a complete database rescan. Due to the large scale of sequence data, significant computational power is required. To ensure scalability, parallelized versions, MMinits+, MMinits-AllOcc, and MMinitsDays, were developed for multi-core architectures.

To evaluate the effectiveness and efficiency of the proposed algorithms, extensive theoretical and experimental evaluations conducted using both real-world and synthetic datasets under various parameters. The results demonstrate that the algorithms can efficiently discover accurate frequent timed sequential patterns. Minits+ outperformed the existing timed sequential pattern mining algorithm SID-PrefixSpan by 9% in execution time, while MinitsDays showed a 42% improvement over Minits-AllOcc. The parallel multi-core versions, MMinits+, MMinits-

AllOcc, and MMinitsDays, achieved an average execution time improvement of 50% compared to their single-core counterparts.

CHAPTER 1
INTRODUCTION

Sequential pattern mining [1] is a data mining task that discovers frequent sequential patterns (subsequences) in a sequence database of time-ordered transactional data. This database contains a set of transactions (sequences), each of which consists of events occurring in chronological order. An example of such a sequential pattern in a sequence database of patients' health is that patients frequently first have high cholesterol, then high blood pressure, followed by high temperature, before eventually having a heart attack. Discovering interesting, useful, and unexpected sequential patterns is beneficial for many real-world applications, such as illness symptom pattern prediction [2]; network intrusion detection [3]; educational data mining [4]; and customer shopping behaviors [1].

However, in many applications, it is important to know the temporal relation between events in a sequential pattern. For example, in healthcare applications, knowing when the next heart attack symptom will occur helps healthcare providers make diagnoses, administer treatments at the right time, and intervene earlier in critical cases. Therefore, it is essential to incorporate the temporal relation between events in a sequential pattern, indicating the required time range to move from one event to another. This newly identified pattern type is termed a *timed sequential pattern*. An example of a timed sequential pattern that can be discovered from a given sequence database of patients' health is as follows: Patients frequently first develop high cholesterol, which remains undetected or unmanaged for two to four years, before progressing to high blood pressure. After one half to one and a half years of persistent hypertension, they begin experiencing a high temperature (fever) and chest pain, typically occurring 7 to 30 days before they eventually suffer a heart attack.

Existing sequential pattern mining algorithms, such as [5], [6], [7], and [8], rely on implicit timestamps to order events within a sequence. However, they fail to preserve the explicit times between successive events, which is crucial for many real-world applications such as illness symptom pattern prediction, weather forecasting, and transportation industry and arrival time analysis. Moreover, competitive algorithms such as [9], [10], [11], and [12] attempt to integrate temporal constraints by requiring user-defined time windows or only considering the first occurrence of a pattern, ignoring all other possible occurrences, which is crucial for many real-world applications such as stock analysis and stock market prediction, web mining system and session analysis, and wildlife tracking.

Furthermore, most existing sequential pattern mining algorithms assume that databases are static, meaning they do not account for changes over time. While this assumption simplifies computation, it does not hold in real-world scenarios where databases continuously grow, update, and evolve. Consequently, finding the complete set of timed sequential patterns efficiently without repeatedly rescanning the database from scratch remains a significant research challenge. Given these gaps, this dissertation first addresses the limitations of existing static sequential pattern mining algorithms by developing a new approach that efficiently discovers timed sequential patterns. Then, this work is extended to dynamic databases by introducing algorithms capable of handling updates incrementally without reprocessing the entire dataset.

This chapter first introduces the background on sequential pattern mining, timed sequential pattern mining, and multi-core CPUs, which serve as the parallel platform for the developed algorithms. Next, several real-world applications are discussed, which highlight the importance of timed sequential patterns. Then, the research challenges associated with mining timed sequential patterns in both static and dynamic timed sequence databases are examined. Finally, the research

objectives are outlined, the contributions of the research are summarized, and the organization of this dissertation is presented.

1.1 Sequential Pattern Mining

Time-ordered events are common in daily life, science, healthcare, marketing, etc., and can be represented as sequences. Agrawal and Srikant [1] introduced the idea of sequential pattern mining that entails identifying intriguing subsequences within a collection of sequences. A subsequence's interestingness can be assessed using a variety of metrics, including how frequently it occurs. For instance, from a heart attack patient's health history, we can find a frequent sequential symptom pattern; for example, 70% of patients who had heart attacks typically have high cholesterol level, followed by high blood pressure, and finally high temperature. The order in which such symptoms occur is significant. Here, 70% reflects the proportion of patients exhibiting these symptoms. Therefore, the problem of sequential pattern mining for this example is defined as follows [1]:

> Given two inputs: a user-defined minimum support threshold, for example 70%, and a database that contains a set of sequences, for example, patients' historic health database, where each sequence consists of a list of elements ordered by the timestamps when the elements occurred and each element has a set of items, sequential pattern mining is to find all sequential patterns whose support is not less than the minimum support, where the support of each pattern is the number of sequences that contain the pattern.
>
> Accordingly, the generalized sequential patterns mining problem is defined as follows [1]:

Given a sequence database, where each sequence is an ordered list of elements, and a user-defined minimum support threshold, the objective is to discover all sequential patterns whose frequency across the database meets or exceeds this threshold.

Many algorithms, such as [5], [6], [7], and [8], have been developed to discover repeating patterns in a sequence database to understand the relations between these ordering items or events. These relations can then be analyzed and interpreted for business strategy, planning, management, and decision-making.

1.2 Timed Sequential Pattern Mining

Timestamps order events in a sequential pattern, but the temporal relation between events is not explicitly represented. In many applications, it is crucial to know the time interval, such as the duration between two consecutive events in a discovered frequent sequential pattern. This is referred to as a timed sequential pattern. The time interval can be expressed using any descriptive statistic according to the user's preference, such as range, average, or median. For example, even if doctors know that symptom B will occur after symptom A for a specific disease, they also want to know the time range in which symptom B is expected to appear, in order to minimize the disease's risks and provide appropriate treatment.

An example of such a discovered pattern from a patients' health history database is < {low temperature, high blood pressure} [2 days, 7 days] {high temperature} >. This pattern consists of two events: event 1, which includes two items (low temperature and high blood pressure), and event 2, which includes one item (high temperature). Event 2 occurs within 2 to 7 days after event 1. In our notation, all items enclosed within braces { } occur at the same timestamp and constitute an event, while the use of square brackets [ ] indicates the temporal relation between two

consecutive events. This timed sequential pattern reveals that when patients experience low temperature and high blood pressure, they are likely to have a high temperature within 2 to 7 days. However, if we apply sequential pattern mining, the resulting pattern would be < {low temperature, high blood pressure}, {high temperature} >, which does not capture the transition time [2 days, 7 days].

1.3 Parallel Platform: Multiprocessors

Multi-core CPUs enable concurrent task execution using multiple processors [13]. These processors are interconnected and share system resources, such as memory and input/output devices, to enhance efficiency, reliability, and scalability especially for tasks that can be divided into parallel subtasks. Parallel processing systems adopt various architectures. There are two main types of multiprocessor systems: Shared-Memory Multiprocessors and Message-Passing Multiprocessors (or shared-nothing architecture). In the first model, all processors Ps share a common memory space M, allowing each processor to access any memory location. This simplifies communication between processors and facilitates data sharing, making it well-suited for tasks that require frequent interaction between processes. However, this model can lead to contention for memory access, as multiple processors may attempt to access the shared memory simultaneously.

In the second model, each processor P has its own local memory M, and processors communicate with each other by exchanging messages. This approach is commonly used in distributed systems, where processors may be physically separated. By providing each processor with its own memory, this model helps reduce memory contention, as there is no shared memory space for processors to compete over. Since the processors in this architecture are independent, the

failure of one processor will not affect the others. However, data cannot be accessed if the associated processor fails. Additionally, inter-processor communication is costly because a processor must request data from another processor's local memory and wait for a response. Figure 1.1 shows the architectures of both shared memory and shared nothing systems.

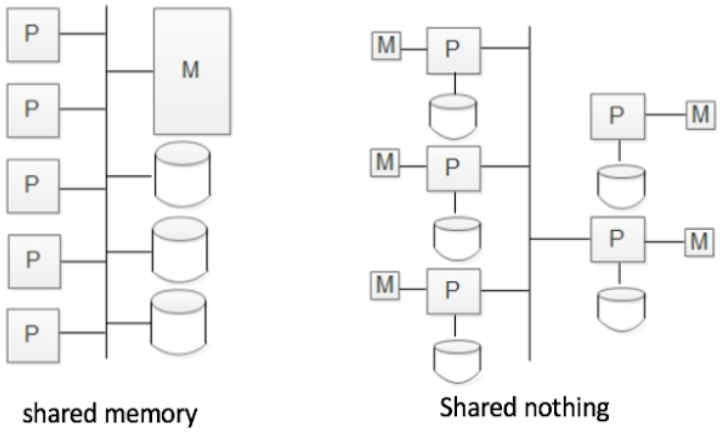


Figure 1.1: Shared memory vs. shared nothing architectures
[www.kullabs.com/classes/subjects/units/lessons/notes/note-detail/6131]

1.4 Application Areas of Timed Sequential Pattern Mining

Timed sequential pattern mining has broad applications, including disease symptom analysis, weather forecasting, the transportation industry, and arrival time analysis. This section describes the applications.

1.4.1 Disease Symptom Analysis

Sequential patterns that include temporal information about symptom transitions can help answer questions such as in what order and when do the symptoms of a particular disease, like a heart attack, frequently occur? Understanding when the next symptom of a heart attack is likely to

occur enables care providers to improve diagnosis, administer treatment at the right time, and intervene earlier in critical cases.

Patient ID	Timestamp	Timestamp Class (days)	Tempera-ture (T)	Temperature Class (TC)	Blood Pressure (BP)	Blood Pressure Class (BPC)
P1	2/10/15 10:50:03	673	37.0	1	131	3
P2	4/9/13 3:15:00	1	36.0	1	112	1
P1	3/12/15 11:00:00	703	37.5	1	128	2
P2	4/9/13 9:00:00	1	37.3	1	134	3
P2	10/10/15 5:23:00	915	38.0	1	110	1
P3	5/13/13 2:20:00	34	37.1	2	135	3
P4	4/11/17 10:45:55	1464	36.9	1	130	2
P3	7/23/16 4:29:00	1202	41.3	1	123	2
P4	6/11/17 11:00:00	1525	38.1	1	139	3
P4	10/1/17 03:30:10	1637	38.3	2	139	3

Figure 1.2: Patients' historical health information and discretized data

Figure 1.2 shows the historic health information (temperature T and blood pressure BP) of four patients who experienced a heart attack. Each patient is denoted as (P) with a unique ID, as indicated in the first column. The time of each measurement is recorded in the second column. Because sequential pattern mining algorithms cannot process continuous data directly, we discretize the data into classes that share similar features or fall within the same range. For the timestamp, the first day of data collection is determined, and the last day of data collection is also determined to specify the time window for data collection. Then, the timestamp is discretized based on a unit such as a day or month, depending on the nature of the application. In the example in Figure 1.2, the first day of data collection is (4/9/13), and the last day of data collection is (10/1/17). After calculating the day number for each timestamp, where the first recorded date is considered Day 1, an additional column was added next to the Timestamp column called

Timestamp Class. Also, the patients' blood pressure (BP) can be categorized into five levels [14]: (1) normal (BP < 120); (2) elevated (120 ≤ BP ≤ 129); (3) high Stage 1 (131 ≤ BP ≤ 139); (4) high Stage 2 (140 ≤ BP ≤ 180); and crisis (BP > 181). To incorporate this categorization, an additional column is added next to each column containing the equivalent class ID. For example, in the first row, the last column shows a class ID of 3 because the blood pressure value of 131 falls within the high Stage 1 range.

A timed sequential pattern that we aim to discover pertains to the symptoms that frequently occur among patients, along with the typical temporal relation between symptoms (measured in days, for example). The format of a pattern that would be identified in this study is as follows:

< {TC1, BPC3} [2 days, 7 days] {TC1}>

For the first event, the temperature from the first class and the blood pressure from the third class appear together, which, in two to seven days, is followed by the temperature from the first class. The temporal relationship [2, 7] is derived through mathematical calculations, which will be explained in detail in Chapter 3.

1.4.2 Weather Forecasting

Sequential pattern mining can be applied in weather forecasting to analyze historical meteorological data, identify recurring weather patterns, and improve the predictive accuracy of future conditions. For monitoring weather forecasts during a tornado season, it is important to track the temporal relation between cities when a tornado strikes. For example, knowing that a tornado will hit Moore 10 to 15 minutes after striking Oklahoma City, and will then reach Norman 3 to 5 minutes later, can assist in preparing a safety plan to minimize damage and losses. Similarly, in Florida during the hurricane season, timed sequential patterns derived from historical weather data

can be used to estimate the temporal relation between tornado strikes in Miami, Fort Lauderdale, and West Palm Beach in sequential order.

Figure 1.3 shows the historical weather data (temperature T and wind speed W) collected from four sensors, with each sensor denoted as (S) and identified by an ID listed 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 cannot process continuous data, a discretization algorithm is applied to segment the data into classes that share similar features or belong to the same group. To address this, a column is added next to each measurement column to include the corresponding class ID. For the timestamp, the first day of data collection is determined, and the last day of data collection is also determined to specify the time window for data collection. Then, the timestamp is discretized based on a unit such as minutes or hours, depending on the nature of the application. In the example in Figure 1.3, the first hour of data collection is (4/9/13 9:00:00), and the last hour of data collection is (4/11/17 10:45:00). After calculating the hour number for each timestamp, where the first recorded date is considered as hour 1, an additional column was added next to the Timestamp column called Timestamp Class. Also, wind speed (W) is categorized into five levels [15] with each level indicating the extent of damage it may cause: (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 ($W \geq 157$). In the first row of the last column, the wind speed is classified as level 3 because the value 112 falls within the range for class 3 (extensive). A timed sequential pattern that we want to discover is a sequence of frequently occurring measurements among sensors (or events) and typical temporal relation 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:

< {TC1, WC3} [30 hours, 36 hours] {WC5}>

Sensor ID	Timestamp	Timestamp Class (hours)	Tempera-ture (T)	Tempera-ture Class (TC)	Wind Speed (W)	Wind Speed Class (WC)
S1	2/10/15 10:50:00	673	54	1	112	3
S2	4/9/13 3:15:00	2	45	1	75	1
S1	3/12/15 11:00:00	703	50	1	100	2
S2	4/9/13 9:00:00	1	41	1	115	3
S2	10/10/15 5:23:00	915	51	1	90	1
S3	5/13/13 2:20:00	35	47	2	125	3
S4	4/11/17 10:45:00	1464	55	1	118	3
S3	7/23/16 4:29:00	1202	50	1	99	2
S4	1/11/17 11:00:00	1374	48	1	127	3

Figure 1.3: Sensors' historical weather information and discretized data

For the first event, the temperature from the first class and the wind speed from the third class occur together. Then, after 30 to 36 hours, the wind speed from the fifth class follows. The temporal relationship [30 hours, 36 hours] is derived through mathematical calculations, which will be explained in detail in Chapter 3.

1.4.3 Transportation Industry and Arrival Time Analysis

In the transportation industry, high temperatures and strong wind speeds can cause train crashes by warping tracks or overturning lightly loaded trains[16], [17]. Sequential patterns that include temporal information about the temporal relation between events can help answer questions such as when the next measurements of train aerodynamics are likely to occur. Additionally, using timed sequential patterns, we can estimate that the travel time range, for

example, for a taxi traveling from Broadway to Times Square in New York City is between 20 and 40 minutes.

1.5 Research Issues

This section discusses the research issues that must be addressed when designing an algorithm for mining timed sequential patterns from static or dynamic timed sequence databases. These issues are categorized and discussed in the following sections: 1) Section 1.5.1 discusses the issues related to sequential pattern mining; 2) Section 1.5.2 discusses the additional issues related to timed sequential pattern mining from a static timed sequence database; and 3) Section 1.5.3 discusses the additional issues specific to timed sequential pattern mining from a dynamic timed sequence database.

1.5.1 Research Issues for Mining Sequential Patterns

This section outlines the main challenges of sequential pattern mining, which include maintaining the order of itemsets, minimizing the number of database scans, reducing the size of the candidate set, and handling timestamp data. An algorithm designed to mine timed sequential patterns must address these common challenges, along with the additional issues discussed in Section 1.5.2 for static timed sequence databases and Section 1.5.3 for dynamic timed sequence databases.

1.5.1.1 Maintaining the Order of Itemsets Based on Timestamps

A sequence $S_A = < A_1 A_2 A_3 ... A_r>$ is an ordered list of itemsets based on timestamps where each A_k (for $1 \leq k \leq r$)is an itemset. An itemset A_k is a set of items $I = \{i_1, i_2 , \dots ,i_m\}$. A sequence $S_B = < B_1 B_2 B_3 ... B_t>$ is a subsequence of S_A if and only if there exists integers, $1 \leq j_1 < j_2 < \dots < j_t \leq r$, such that $B_1 \subseteq A_{j1}$, $B_2 \subseteq A_{j2}$, ….., $B_t \subseteq A_{jt}$. [18].

Therefore, it is crucial to preserve the order of itemsets when generating sequences that might possibly be frequent (or candidates [19]) for sequential pattern mining. The objective of a sequential pattern mining algorithm is to identify the complete set of subsequences whose occurrence frequencies meet or exceed the user-defined minimum support threshold, which is a user-defined parameter that specifies the minimum frequency at which a pattern must appear in the database to be considered frequent. Additionally, all itemsets within these subsequences must be ordered based on their timestamps. Consequently, a sequential pattern mining algorithm must generate a complete and accurate set of frequent sequential patterns, ensuring that each subsequence maintains the correct itemset order according to their respective timestamps.

Requirement 1. An algorithm that mines timed sequential patterns from a static or dynamic timed sequence database must consider the timestamp order of itemsets.

1.5.1.2 Minimizing the Number of Times a Database is Scanned

This issue concerns the number of full database scans required to check whether or not a candidate is a frequent sequence [20]. A sequence in the database must be frequent to be a frequent sequential pattern. Existing algorithms generate all the sequence candidates of 1-length (i.e. the sequence candidates each of which contains only one item [6]), compute the support for each sequence by scanning the whole database, and check to see if it is frequent or not. This process continues for the next k-length sequence candidates (i.e., the sequence candidates each of which contains k items [6]) until no more new candidates are generated. This means the database must be scanned many times to find all sequence candidates. Therefore, it becomes challenging to scan a huge database many times. Accordingly, many proposed algorithms, such as [6], and [21], try to find a method or use a data structure to minimize the number of scans of the database.

Requirement 2. An algorithm that mines timed sequential patterns from a static or dynamic timed sequence database should minimize the number of scans of the database.

1.5.1.3 Reducing the Size of the Sequence Candidate Set

This issue points to the number of candidates generated in every phase to produce frequent sequences of k-length. To find the frequent sequential patterns, first, all possible permutations of single items are generated. Next, the set of frequent sequences of 1-length is defined and used as a seed set for the next step. Then, all possible candidates of 2-length sequences are generated. Each candidate in this set is tested to decide if it is frequent or not. After that, the frequent sequential pattern of 2-length is used as a seed set to generate a new candidate set of 3-length, and so on. For example, if we have 1000 distinct frequent items, the Apriori-based methods will generate $1000 \times 1000 + (1000 \times 999) /2 = 1{,}499{,}500$ candidates of 1-length. After infrequent sequences are eliminated, the candidates of 2-length are generated, so the size of each new candidate will increase, and the test time will also increase. Thus, the algorithm should minimize both candidate set size and validation time. Some of the early algorithms, such as GSP, use a generate-and-test procedure, which uses exhaustive join operators to generate candidates and then test them to keep only the frequent candidates. One of the well-known principles is called Apriori, which states that "If an itemset is frequent, then all of its subsets must also be frequent." The following example is used to explain how this property works. If we have the itemset lattice shown in Figure 1.4. and the <{c, d, e}> is found to be a frequent sequence, then <{c}>, <{d}>, <{e}>, <{c, d}>, <{c, e}>, and <{d, e}> are frequent because any tuple in the database that contains {c, d, e} must contain all previous subsets (the gray oval). In contrast, the < {a, b}> is found to be infrequent, so any supersets that contain < {a, b}> are not frequent and should be pruned.

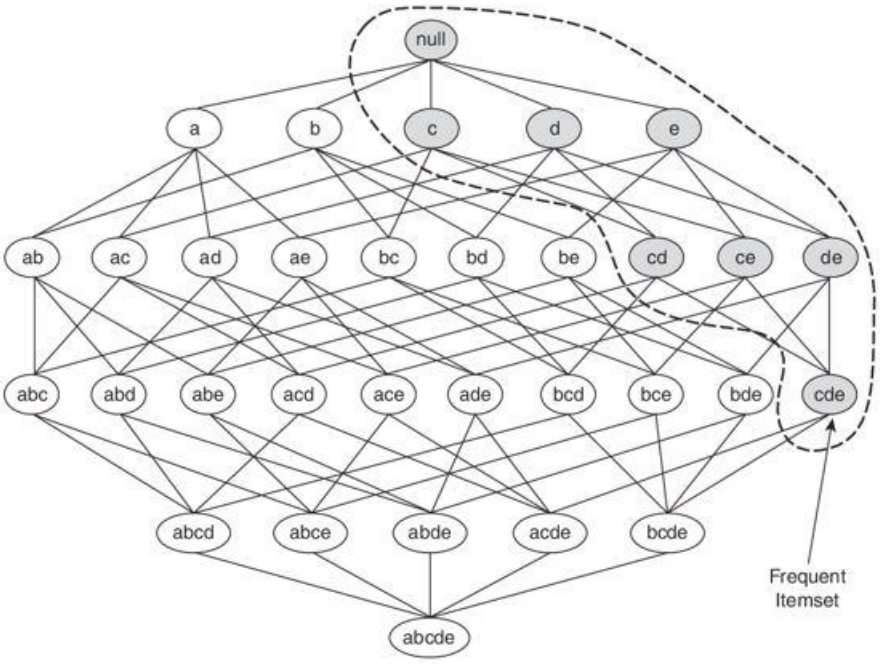


Figure 1.4 : An illustration of the Apriori principle

Requirement 3. An algorithm that mines timed sequential patterns from a static or dynamic timed sequence database should include a strategy to prune the generated candidates.

1.5.1.4 Handling Large-Scale Timed Sequence Databases

Large-scale sequence databases are those that have many sequences and each sequence has many itemsets. Executing an algorithm for discovering timed sequential patterns requires scanning each tuple in the database. This task will become expensive if the database has many tuples, and the tuples are very long. To handle this problem, parallel computing platforms, such as multi-core CPUs, should be used.

Requirement 4. An algorithm that mines timed sequential patterns from a static or dynamic timed sequence database should be designed to exploit the benefits of parallel computing architectures to deal with large-scale timed sequence databases.

1.5.2 Additional Research Issues for Mining Timed Sequential Patterns in Static Timed Sequence Databases

This section introduces the additional challenges of mining timed sequential patterns in a static timed sequence database, which include handling timestamp data, tracking all possible occurrences of a sequence, and updating temporal relations.

1.5.2.1 Handling Timestamp Data

All existing sequential pattern mining algorithms, such as [5], [6], [7], [8] use timestamps to order the itemsets in a sequence in a database but discard the time dimension while mining the database. The temporal relation between the itemsets is not shown in the discovered sequential patterns. In many applications, it is important to have sequential patterns that include the temporal relations, for example [min, max], between the itemsets in discovered frequent sequential patterns, which are called a timed sequential pattern. Therefore, a timed sequential pattern mining algorithm needs to address the time dimension in the discovered patterns.

Requirement 5. An algorithm that mines timed sequential patterns from a static or dynamic timed sequence database needs to incorporate the time dimension and calculate the temporal relations.

1.5.2.2 Tracking All Possible Occurrences of a Sequence

While both sequential pattern mining and timed sequential pattern mining involve determining whether a pattern occurs in certain tuples of a database, timed sequential pattern mining also requires identifying how many times the pattern occurs within each tuple. Many real applications need to consider all possible occurrences of a pattern to provide accurate temporal information such as stock analysis and stock market prediction, web mining system and session analysis, and wildlife tracking. In business intelligence, it is possible to analyze stocks in detail to

determine whether each one is a viable investment. The existence of frequent patterns suggests that specific market behaviors often occur together, making it possible to predict corresponding trends. An algorithm for mining timed sequential patterns can be developed to discover a timed sequential pattern, such as the following:

<{Open1} [3 days, 21 days] {Close3}>

The interpretation of the pattern is the following: approximately the stocks of most companies opened at a low price (as the opening price was discretized to the low price class 1) and frequently closed at a high price (as the closing price was discretized to the high price class 3) over a period of 3 to 21 days.

Additionally, in web mining systems and session analysis, analyzing user behavior while browsing a website is essential for improving the user experience and increasing engagement. By studying browsing patterns, we can identify what users are looking for and which content attracts them the most. Analyzing which areas of a webpage receive the most attention helps optimize the webpage structure, ensuring that critical information and calls to action are strategically placed where they are most likely to be noticed. An algorithm for mining timed sequential patterns can be developed to discover a timed sequential pattern, such as the following:

<{URL1} [10 mins,15 mins] {URL2} [5 mins,9 mins] {URL1}>

The sequential pattern indicates a recurrent user behavior wherein a majority of users transitioned to URL 2 after spending an average duration of 10 to 15 minutes on URL 1.

Another application is wildlife tracking, which involves analyzing animal movement sequences to study migration patterns. This helps identify locations where animals frequently stop,

which may be critical for resting and feeding. An algorithm for mining timed sequential patterns can be developed to discover a timed sequential pattern, such as the following:

<{Lake} [2 days, 4 days] {Lake}>

The interpretation of the pattern is as follows: ducks stopped multiple times at lakes during their migration journey, and this stopping lasted around 2 to 4 days.

Patient ID	Tuples
P1	< {2/10/15 10:50:03, TC1, BPC3}, {3/12/15 11:00:00, TC1, BPC2} >
P2	< {4/9/13 03:15:00, TC1, BPC1}, {4/9/13 09:00:00, TC1, BPC3} {10/10/15 05:23:00, TC1, BPC1} >
P3	< {5/13/13 02:20:00, TC2, BPC3}, 7/23/16 4:29:00, TC1, BPC2} >
P4	< {4/11/17 10:45:55, TC1, BPC2} {6/11/17 11:00:00, TC1, BPC3} {10/1/17 03:30:10, TC2, BPC3}>

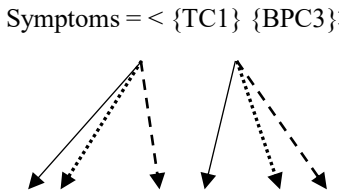
Figure 1.5: Timed Sequence Database for patients

As illustrated in Figure 1.5, we present historical health data consisting of the temperature (T) and blood pressure (BP) of four patients (P) who had a heart attack. The time of each measurement, whether for temperature or blood pressure, is recorded in the second column. Hypothetically, we have a tuple of a patient containing all measurements over six months, with the following symptoms occurring multiple times: low temperature followed by high blood pressure after some time. Since the goal of the timed sequential pattern mining problem is to precisely determine the temporal relations between low temperature and high blood pressure, it is insufficient to identify and report only the first occurrence of this symptom sequence along with

its temporal relationship. All possible occurrences of this sequence and their corresponding timestamps must be identified and reported to ensure accurate time information, as stakeholders rely on this temporal data to make critical decisions. For instance, in Figure 1.5, we observe that P4 exhibits the following symptoms in timestamp order: the temperature from Class 1 and the blood pressure from Class 2 <{TC1, BPC2}>, followed by the temperature from Class 1 and the blood pressure from Class 3 <{TC1, BPC3}>, and then the temperature from Class 2 and the blood pressure from Class 3 <{TC2, BPC3}>. The tuple representing this patient is: P4 = < {TC1, BPC2}, {TC1, BPC3}, {TC2, BPC3}>.

To find the temporal relation between the two symptoms {TC1} and {BPC3} (for the pattern denoted as < {TC1} {BPC3}>), we need to do the following as shown in Figure 1.6:

1. Find the timestamp difference t1 between the first occurrence of TC1 and the first occurrence of BPC3 (solid arrows).
2. Find the timestamp difference t2 between the first occurrence of TC1 and the second occurrence of BPC3 (dotted arrows).
3. Find the timestamp difference t3 between the second occurrence of TC1 and the first occurrence of BPC3 (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] for example



P4 = < {TC1, BPC2}, {TC1, BPC3}, {TC2, BPC3}>

Figure 1.6: All possible occurrences of the pattern <{TC1} {BPC3}> in P4

To find all possible occurrences of a pattern, the naïve method is to scan each tuple until the last tuple in the database is reached. 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 tuples. First, it is necessary to consider all possible occurrences of the pattern, as many real applications, such as stock analysis and stock market prediction, web mining system and session analysis, and wildlife tracking, need to identify this kind of pattern and all the different timestamps of each occurrence and find the temporal relation. After the temporal relation is found for one patient, it is necessary to check the temporal relation for the same symptoms among all patients. The final temporal relationship represents the time difference between the itemsets among all patients in the database.

Requirement 6. An algorithm that mines timed sequential patterns from a static or dynamic timed sequence database should be able to deal with the case when all possible occurrences of a sequential pattern need to be considered and calculate the temporal relations between the itemsets in a pattern accordingly.

1.5.2.3 Updating Temporal Relations

The issue discussed in the previous section 1.5.2.2 introduces a challenge in developing a timed sequential pattern mining algorithm that updates the temporal relationships between itemsets

as soon as a pattern is identified. When a timed sequential pattern is found, it indicates that the ratio of the number of tuples containing this pattern to the total number of tuples in the database is greater than or equal to the user-defined minimum support threshold. When we attempt to extend a pattern by including additional itemsets, such as the extra symptoms mentioned in the previously described healthcare application, it does not necessarily imply that the extended pattern will appear within the same tuples as the original pattern. Some tuples may still contain the newly extended pattern exactly as the original, so no update to the temporal relationships is ever needed. However, some tuples may not contain the newly extended pattern, even if the original pattern is present, so updating the temporal relationships is needed.

If a pattern pt1 is frequent, it means that the proportion of tuples (denoted as n) containing pt1 to the total number of tuples in the database is greater than or equal to the minimum support threshold. Since pt1 is frequent, it can be extended to generate a new pattern pt2, for example. A pattern pt2 is an extension of pattern pt1, and it will be considered frequent if it appears in m tuples, where m is greater than or equal to the minimum support threshold.

Of course, m and n can hold the same or different values, meaning that the number of tuples containing pattern pt1 and pattern pt2 may be equal or different. Consequently, the temporal relationship becomes invalid if the values differ and must be updated based on the new timestamps of the tuples containing the extended pattern pt2. For example, let us consider the timed sequential pattern: < {TC1, BPC3} [t1, t2] {TC1}> From Figure 1.5, we observe that patient P1 (Tuples 1 and 3) and patient P2 (Tuples 4 and 5) exhibit this pattern. Consequently, t1 and t2 are calculated based on the timestamps associated with this pattern in the respective tuples. When the pattern is extended, it becomes: < {TC1, BPC3} [t1, t2] {TC1} [t3, t4] {TC1, BPC1}>. At this stage, we notice that the tuples of P2 do not include this pattern, whereas only the tuples of P1 exhibit this

pattern. Therefore, t1 and t2 must be updated based on the timestamps associated with the symptoms in Tuples 1 and 3, rather than Tuples 4 and 5.

The brute-force algorithm requires rescanning the entire database to update the pattern's temporal relations. Thus, for every pattern, the database must be scanned multiple times to ensure that the correct temporal relations are maintained.

Requirement 7. An algorithm that mines timed sequential patterns from a static or dynamic timed sequence database needs to ensure that the temporal relations between the itemsets in the new pattern is correct when a new sequential pattern is found.

1.5.3 Additional Research Issue for Mining Timed Sequential Patterns in Dynamic Timed Sequence Databases

This section discusses the additional challenge of mining timed sequential patterns in a dynamic timed sequence database, which deals with different types of database modifications.

1.5.3.1 Dealing with Different Types of Database Modifications

Most of the existing sequential pattern mining algorithms assume that a database is static, such as [5, 6], [7], and [8]. However, this does not hold in practice because databases in real-world applications change over time. Adding new data to an existing sequence database, deleting old data from an existing sequence database, or modifying the existing data would affect the identification of recent frequent timed sequential patterns. For example, after deleting some of the old data from a database, some of the obsolete timed sequential patterns are no longer frequent timed sequential patterns but will still be saved. Thus, after the content of the database is changed, the algorithm needs to identify this change, whether it is insertion, deletion, or modification, and update the discovered timed sequential patterns accordingly.

Requirement 8. An algorithm that mines timed sequential patterns from a dynamic timed sequence database should be able to deal with different types of database changes and update the timed sequential patterns without re-mining the entire database.

1.6 Research Objective

The objective of this research is to develop novel algorithms for efficiently and correctly discovering frequent timed sequential patterns from static and dynamic timed sequence databases that address the following research issues:

- Maintaining the order of itemsets, minimizing the number of database scans, and reducing the size of candidate set (Research Issues 1.5.1.1, 1.5.1.2, 1.5.1.3)
- Addressing scalability through parallel processing on multi-core CPUs (Issue 1.5.1.4)
- Calculating and updating the temporal information between itemsets within a pattern (Research Issues 1.5.2.1, and 1.5.2.3)
- Tracking all occurrences of a pattern over time (Research Issue 1.5.2.2)
- Avoiding rescanning the entire database when mining timed sequential patterns in a dynamic timed sequence database caused by three types of database updates: insertion, deletion, and modification (Research Issue 1.5.3.1)
- Storing the historical information of the old timed sequence database in data structures that facilitate efficient access

1.7 Contributions

Mining frequent sequential patterns in a sequence database supports many real-world applications, such as analysis of customer shopping behaviors [1]; illness symptom pattern

prediction [2]; network intrusion detection [3]; educational data mining [4]; and weather prediction [22], [23]. In many applications, it is important to discover sequential patterns that include the temporal relationships between itemsets, which are called timed sequential patterns. The sequential pattern algorithms that exist in the literature, such as [5], [6], [7], and [8] mostly disregard the time dimension in the discovered patterns. There exist a few sequential pattern mining algorithms that deal with time, notably [4, 9, 24, 25], but they either do not provide the desired patterns proposed in this work or fail to address all the research challenges in timed sequential pattern mining. Some of these timed sequential pattern mining algorithms either require the user to enter time intervals as input, assume that the database is static, or do not consider the scalability issue concerning the database size, and thus do not reflect databases in many real-world applications.

In this dissertation, three algorithms are proposed for mining timed sequential patterns: Minits+, Minits-AllOcc, and MinitsDays.

Minits+ (MINIng Timed Sequential patterns) is a single-core CPU algorithm designed for the following:

- incorporating temporal relationship between itemsets into a sequential pattern, which indicates the required time, based on the user's requirement, to move from one itemset to another
- discovering the frequent timed sequential patterns from a static timed sequence database

We analyze the time complexity and prove the correctness of the algorithm. Moreover, we conduct experiments using real-world and synthetic datasets, comparing Minits+ with the existing algorithm SID-PrefixSpan [11] in terms of execution time and the number of discovered patterns. Minits+ outperformed SID-PrefixSpan, achieving a 9% average improvement in execution time. Also, Minits+ is 100% accurate, producing the same patterns (excluding temporal relations) in

terms of quantity and content as PrefixSpan [6]. Multi-core CPUs version of Minits+ called MMinits+ is also implemented to improve the performance of Minits+ for large-scale timed sequence databases. MMinits+ outperformed Minits+ with a 40% average speedup.

Minits-AllOcc (MINIng Timed Sequential Pattern for All-time Occurrences) is a single-core CPU algorithm designed for the following:

- incorporating time intervals between itemsets in a sequential pattern, which indicates the required transition time range to move from one itemset to another
- finding the transition time between itemsets in a timed sequential pattern based on all occurrences of a pattern in a timed sequence database
- discovering the patterns from a static timed sequence database

We analyze the time complexity and prove the correctness of the algorithm. Moreover, we conduct experiments using real-world and synthetic datasets in terms of execution time and the number of discovered patterns. Minits-AllOcc is 100% accurate, producing the same pattern count and content, excluding temporal relations, as PrefixSpan. The multi-core CPU version of Minits-AllOcc, called MMinits-AllOcc, is also implemented to tackle the issue of expensive operations when the size of the timed sequence database grows. MMinits-AllOcc outperformed Minits-AllOcc in terms of execution time, achieving, on average, a 50% improvement.

MinitsDays (MINIng Timed Sequential Pattern in DYnamic Sequence Database) is a single-core CPU algorithm proposed for the following:

- incorporating time intervals between itemsets in a sequential pattern, which indicates the required transition time range to move from one itemset to another
- finding the transition time between itemsets in a timed sequential pattern based on all occurrences of a pattern in the timed sequence database

- discovering the patterns from a dynamic timed sequence database considering three cases of database updates: insertion, deletion, and modification

We analyze the time complexity and prove the correctness of the algorithm. Moreover, we conduct experiments using real-world and synthetic datasets, comparing MinitsDays with Minits-AllOcc in terms of execution time and the number of discovered patterns. MinitsDays outperformed Minits-AllOcc, achieving an average 42% improvement in execution time. To demonstrate the scalability of this algorithm, especially when the content and size of the database keep changing, the multi-core CPU version called MMinitsDays is also implemented. MMinitsDays outperformed MinitsDays, demonstrating on average a 32% improvement in execution time.

To the best of our knowledge, no existing algorithm for discovering timed sequential patterns from either static or dynamic timed sequence databases considers all possible occurrences of a pattern across the entire database. Furthermore, no algorithm in the current literature can discover timed sequential patterns from dynamic timed sequence databases. Current algorithms discover either sequential patterns from dynamic timed sequence databases or discover timed sequential patterns from static timed sequence databases.

This work provides a comprehensive theoretical and experimental analysis of the proposed algorithms. In the experimental analysis, the impact of various user-defined threshold values (minimum support threshold), timed sequence database size, and timed sequence length on the execution time of the proposed algorithm is evaluated using both real and synthetic datasets.

1.8 Organization of This Dissertation

The dissertation is outlined as in Figure 1.7, where the arrows show the dependencies between the chapters. It is organized as follows: Chapter 2 reviews the existing work related to sequential pattern mining and dealing with temporal relations. Chapter 3 presents our proposed algorithms and their implementations. Chapter 4 presents the analytical and experimental results of our algorithms. Finally, Chapter 5 provides conclusions and future research directions.

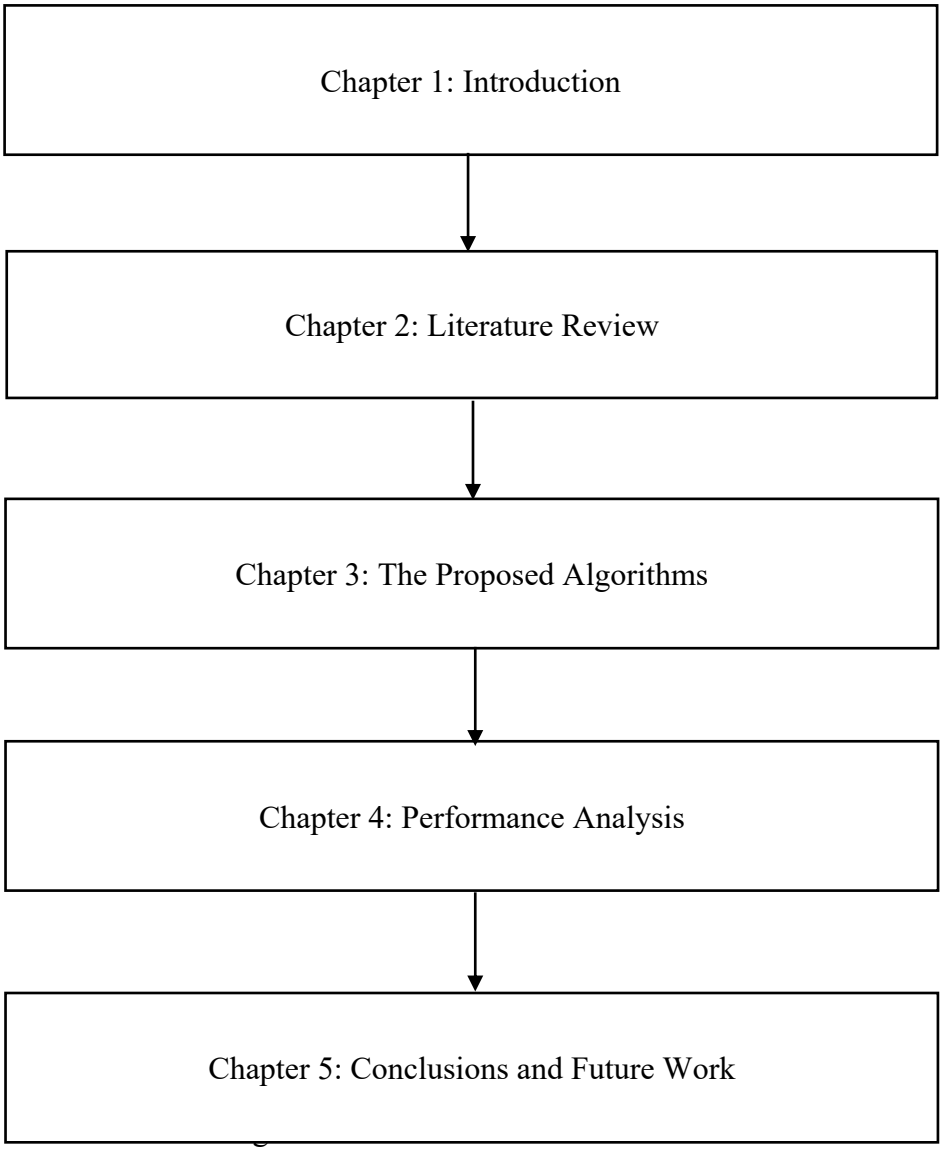


Figure 1.7: Dissertation Outline

CHAPTER 2
LITERATURE REVIEW

Sequential pattern mining is a data mining task used to identify subsequences within a sequence dataset that co-occur in a specific temporal order [1]. While sequential pattern mining algorithms can uncover the sequential relationships between successive events, they do not provide information about the time intervals between them. To address this limitation, the literature has been reviewed and categorized into three key sections. Section 2.1 briefly presents the main extensions of sequential patterns discussed in the literature. Section 2.2 provides a survey of techniques for incorporating temporal relationships into sequential patterns. Section 2.3 presents a survey of techniques for discovering sequential patterns in dynamic databases.

2.1 Variations of the Sequential Pattern Mining Problem

The sequential pattern mining problem has been extended and adapted in various ways to address specific challenges and application scenarios, resulting in several variations that go beyond the traditional approach. These variations aim to incorporate additional dimensions, improve efficiency, and enhance the quality of the discovered patterns. This section reviews some of the most popular extensions of sequential pattern mining.

2.1.1 Sequential Patterns [1]

Sometimes, it is important to determine whether customers purchase products in a specific sequence, such as buying beer first and bread later. This can be achieved by using sequential pattern mining. In this approach, all events in the dataset are recorded with timestamps, ensuring that, for example, a customer is known to have purchased bread before milk.

Various techniques have been developed to generate subsequences, calculate their support count, and compare it against a user-defined minimum support threshold, denoted as min_sup. A subsequence is identified as a sequential pattern if its support count is greater than or equal to min_sup. For instance, with a min_sup of 50%, some discovered patterns from the sequence database are listed in the output.

Input: Sequence Database

Customer ID	Data Sequence
1	< {Bread}, {Milk} >
2	< {Bread, Diapers}, {Beer, Eggs} >
3	< {Bread, Milk, Diapers}, {Beer} >
4	< {Bread, Milk, Diapers}, {Coke} >

Output: Sequential Patterns

Sequential patterns
< {Beer} >
< {Milk} >
…..
< {Bread}, {Beer} >
< {Diapers}, {Beer} >
…..
< {Bread, Milk, Diapers} >
……

Figure 2.1: Input and Output for Sequential Pattern Mining.

2.1.2 Fuzzy Sequential Patterns [26]

Sequential pattern algorithms are not designed to process numerical information, even though they are well-suited to the temporal component of data. To address this limitation, these algorithms have been extended into fuzzy algorithms, which handle databases where objects in sequences have numerical values. Numerical values are mapped to nominal values using fuzzy membership functions. Each item is represented as a pair: (item name, item amount). For example, fuzzy algorithms can define three fuzzy regions: *low, middle*, and *high*. Using this approach, fuzzy

sequential patterns can represent scenarios such as the sequential purchase of bread in average quantities followed by a large quantity of diapers.

Given the sequence database below and a min_sup of 50% as inputs, some of the discovered fuzzy sequential patterns are listed in the output.

Input: Sequence Database

Customer ID	Data Sequence (Items and quantities)
1	< {Bread,2}, {Milk,3}>
2	< {Bread,5}, {Diapers,2}, {Beer,4}, {Eggs,9} >
3	< {Bread,7}, {Milk,8}, {Diapers,7}, {Beer,9} >
4	< {Bread,1}, {Milk,3}, {Diapers,12}, {Coke,5} >

Output: Fuzzy Sequential Patterns

Fuzzy Sequential Patterns
< {Beer. High}>
< {Milk. Low}>
< {Bread. Middle}, {Diapers. High}>
< {Bread. Middle, Milk. Middle}, {Coke. Low}>

Figure 2.2: Input and Output for Fuzzy Sequential Pattern Mining.

2.1.3 High-Utility Sequential Patterns [27]

High-utility sequential pattern mining is an extension of sequential pattern mining. It aims to discover subsequences with high utility (importance or weight) in a sequence database with quantitative attributes. For example, in a retail market, each item may have a different price or profit value, and a user may purchase multiple units of the same item. In this context, each item in the database is associated with an external utility value, such as the unit profit of the purchased item, to reflect the relative importance of the patterns. Additionally, every instance of an item is linked to an internal utility, representing the quantity of the item a consumer bought during a transaction. Instead of the min_sup threshold used in sequential pattern mining, a new user-defined parameter, called the minimum utility threshold (min_utility), is introduced. The min_utility is

calculated as the sum of the maximum utility (profit) generated by the pattern across all sequences in which it appears.

For instance, given the sequence database below, along with the profit of each item and a min_utility of 120 as inputs, some of the discovered high-utility sequential patterns are listed in the output.

Input: **(1) Sequence Database**

Customer ID	Data Sequence with internal utility
1	<{Bread,2},{Milk,3}>
2	<{Bread,5}, {Diapers,2},{Beer,4},{Eggs,9 }>
3	<{Bread,7},{Milk,8},{Diapers,7}, {Beer,9}>
4	<{Bread1},{Milk,3},{Diapers,12},{Coke,5}>

(2) **Item Profit**

Item	Profit ($) per unit
Bread	5
Beer	7
Diapers	10
Milk	6

Output: High-utility sequential patterns

High-utility sequential patterns
Bread: 121 …..
Bread, Diapers:310 …..
Bread, Milk, Beer: 292….

Figure 2.3: Input and Output for High- Utility Sequential Pattern Mining.

2.1.4 Periodic-Frequent Patterns [28]

This work aims to identify patterns in a database that occur periodically. For example, in a retail market, a seller might be primarily interested in items that are regularly offered compared to other products. The interval between two occurrences of a pattern is referred to as the pattern's period, which can be measured in terms of the number of transactions between two occurrences of an event in the database. The user must determine the period length, such as the average or

maximum length of the period. For instance, if min_sup = 3 and MaxPer = 2, a pattern is considered periodic if its period length does not exceed 2 transactions and it appears in at least 3 transactions.

Input: Transaction Database

Trans-action ID	Items
1	Bread, Milk
2	Bread, Diapers, Beer, Eggs
3	Milk, Diapers, Beer, Coke
4	Bread, Milk, Diapers, Beer
5	Bread, Milk, Diapers, Coke

Output: Periodic-frequent patterns

Periodic-frequent patterns	Support Count	Max Period
{Diapers}	4	2
…..	…	…
{Beer, Diapers}	3	1
…..	…	…

Figure 2.4: Input and Output for Periodic-Frequent Pattern Mining.

2.1.5 Spatio-Temporal Patterns [29]

This is a special type of sequential pattern mining that focuses on the movement of objects. In this case, the database contains sequences of spatial locations recorded at specific timestamps. Many studies have been proposed to identify frequently repeated paths, which can help in understanding, analyzing, and predicting the movement of people, animals, vehicles, and more. The format of spatio-temporal data is typically represented as (timestamp, latitude, longitude). For example, given the spatio-temporal database below and a min_sup of 50% as inputs, some discovered spatio-temporal sequential patterns are listed in the output.

Input: Spatio-temporal Database

Taxi ID	Trajectory Sequences
1	<(2008-10-23 02:53:04, 39.93, 116.31),……, (2008-10-29 11:11:12, 14.2,1.03)>
2	<(2010-10-23 12:45:23, 39.92, 116.33),……., (2010-11-12 16:44:22, 39.93, 116.31)>
3	<(2014-8-23 02:53:04, 17.2, 0.25),……, (2014-8-23 11:11:12, 17.1,0.10,) …… >
4	<(2014-3-13 00:11:12, 39.93, 116.31), (2014-5-27 05:12:50, 14.2,1.03), …. >

Output: Spatio-Temporal Patterns

Spatio-temporal sequential patterns
< (39.93, 116.31)>
…..
< (39.93, 116.31), (14.2,1.03)>
……..

Figure 2.5: Input and Output for Spatio-Temporal Mining.

In this dissertation, we focus exclusively on one type of sequential pattern: mining sequential patterns. After reviewing the literature on discovering sequential patterns from sequence databases, we compiled all existing techniques that incorporate temporal information, as summarized in Table 2.1. These techniques address temporal relationships from varying perspectives, leading to the development of different types of sequential patterns. Table 2.1 provides a detailed comparison of the input, output, and format of these patterns.

2.2 Literature Review of Incorporating Temporal Relation in Mining Sequential Patterns Algorithms

2.2.1 I-Apriori and I-PrefixSpan [10]

This research discusses the significance of time-interval sequential patterns in data mining, emphasizing their ability to provide valuable insights beyond sequential patterns. Thus, two novel algorithms are proposed, the I-Apriori and I-PrefixSpan for mining time-interval sequential patterns from sequence data. The I-Apriori is a modification of the Apriori algorithm [1] and I-PrefixSpan is a modification of the well-known PrefixSpan algorithm [6]. The study shows that the I-PrefixSpan algorithm is more efficient than I-Apriori algorithm, especially when the minimum support threshold is small. The I-PrefixSpan algorithm outperforms the I-Apriori algorithm in terms of efficiency. The inputs for these algorithms are: minimum support min_sup, sequence DB, and time intervals TI={ I1, I2, ...}, and the output is the set of time-interval patterns, which each pattern denoted as P = (I1, &1, I2, &2, ..., Im-1, &m-1, Im), where Ii represents an item and &i represents the time interval between item Ii and item Ii+1.

2.2.2 T-Pattern [9]

A novel type of spatio-temporal called Trajectory Patterns (T-patterns) was introduced in this study. These patterns represent the movement behavior of people or things traveling through various geographical locations with comparable travel times. They are a spatio-temporal variant of the Temporally Annotated Sequences [30]. Formally, a T-pattern is represented as (S, A), where: S = <(x0, y0), ..., (xk, yk)> is a sequence of spatial points in R^2, and A = <α1, ..., αk> ∈ R^k+ is the temporal annotation of the sequence. T-patterns are denoted as (S, A) = (x0, y0) α1 --> (x1, y1) α2 --> ... αk --> (xk, yk). T-patterns enable a detailed analysis of movement patterns,

considering not only the order of locations visited but also the time taken to transition between them which was informative in diverse applications and fields.

2.2.3 MI- Apriori and MI-Prefix [24]

The shortcomings of previous studies in general as well as those of I-Apriori and I-PrefixSpan [10] are addressed in this work. The general methods used in previous publications are to set time constraints on the patterns or find the time-intervals between two successive items. Unfortunately, these methods are unable to obtain the time-interval data in a sequential pattern between non- successive items. A new method was put up to address the shortcomings of the methods that were already in use. Some possible expansions included the use of rough set theory or fuzzy theory, the creation of a hierarchy of time intervals, and the imposition of meta-rule constraints on the patterns that were formed. To expose the time intervals between each pair of items in the pattern, I-Apriori and I-PrefixSpan algorithms were modified to create the MI-Apriori and MI-PrefixSpan algorithms. The inputs for these algorithms are: Minimum support min_sup, Sequence DB, and Time Intervals TI={ ti1, ti2,..tir-1}, and the output is the set of multi-time-interval patterns which each pattern denoted as A= <a1, ti1,a2, (ti2, ti1), a3, (ti3, ti2, ti1),….,as-1 (tis-1….. ti3, ti2, ti1) as >, where ai represents an item and (ti) represents the time interval between item a1 and item ai.

2.2.4 STARM-TS [25]

In [9], the travel time from location A to location B is fixed. The pattern has this format: Railway Station → [After 15 min] Castle Square → [After 2h] Museum. However, in real life, the travel time is not strict due to various circumstances. Therefore, this research proposed a novel real-time framework called STARM-TS for extracting patterns from trajectory data streams and relaxing the constraint on travel time from a fixed to a range of times. A discovered pattern looks

like: Railway Station → [After 10 min to 60 min] Castle Square → [After 2h to 4h] Museum. So, rather than a taxi arriving at Castle Square from the Railway Station in 15 minutes, a taxi departing from the station to Castle Square arrives in 10 to 60 minutes, then arrives at the Museum in 2 to 4 hours due to traffic. The inputs for STARM-TS are: Minimum support min_sup, raw trajectory data TS, and user-defined interesting locations IL. The algorithm scans the entire trajectory dataset only once, builds a compact data structure called TSM-tree for storing patterns, traverses the tree to discover frequent interesting trajectories, and then extracts the maximal frequent interesting trajectory patterns.

2.2.5 CAI-PrefixSpan [31]

This work presents the CAI-PrefixSpan algorithm, devoted to timed sequential pattern mining with a guarantee of event occurrences within designated time intervals. Algorithms such as PrefixSpan may not capture the containment relationship among time periods, resulting in unclear support counting and potential pattern omission. CAI-PrefixSpan addresses this by first ensuring a minimum confidence level for event occurrence and then calculating the minimal time interval to meet the support requirement. CAI-PrefixSpan reads the following: Minimum support min_sup, Minimum confidence min_conf, set of accumulated intervals TI, and timed sequential database. After identifying a particular event sequence, the algorithm ensures that only patterns with a minimum confidence requirement are considered for further analysis. For patterns that pass the confidence requirement, CAI-PrefixSpan computes the minimal time interval needed to satisfy the support requirement. Then, projected databases are generated based on the timed sequential patterns identified in the previous steps. These projected databases contain sequences that match the timed patterns. Finally, CAI-PrefixSpan calls itself recursively to the databases that are

projected to investigate, and extract timed sequential patterns in more detail. By using a recursive method, timing information is obtained for fewer, more reliable sequential patterns.

2.2.6 Alzahrani's Algorithm [32]

The sequential pattern mining of medical data, which includes disease, symptom, and patient records ordered chronologically based on hospital visits, is the main emphasis of this study. The goal of the researchers' approach is to find illness sequences that occur within user-specified time intervals by extracting sequential patterns from medical datasets that have temporal constraints. The study aims to provide insights into cause-and-effect interactions in diseases by addressing the issue of time gaps between causes and effects. The algorithm requires the following inputs: minimum support (min_sup), patients' medical data (specifically, the Bupa and Heart Disease datasets), and user-defined time intervals. The algorithm identifies frequent diseases in the dataset by counting their counts the support. After that, it generates frequent 1-sequences, which are sequences of size 1 representing individual diseases, then generates candidate k-sequences by combining frequent (k–1)-sequences to form potential k-sequences. The dataset is scanned to calculate the frequency of each candidate to determine how often the sequence occurs within the specified time constraints. The process of candidate generation and frequency calculation iterates to find sequential patterns of varying lengths by increasing values of k. The algorithm enforces time constraints throughout the process to ensure that the time intervals based on frequent sequential patterns discovered reflect realistic temporal relationships between diseases.

2.2.7 ti-Pattern Model [4]

The paper explores the use of time-interval pattern mining to model students' activities in educational settings. It highlights the importance of considering time in understanding students' behaviors and designing accurate models. In educational data mining, a ti-pattern model is built

based on the I-PrefixSpan algorithm to consider the time between students' activities. The inputs of this model are: Minimum support min_sup, a temporal sequence database, and a set of time intervals (Is, Imn, Ih, Id, Iw, Imt), which refer to human natural time seconds, minutes, hours, days, weeks, and months. The output is set of time-interval sequential patterns. For example, one of the patterns that uses the gap interval between 1 hour and 1 day after the model is applied on a group of students who enrolled in a mathematics and computer science program in Learning Management System (LMS) is <Lab {1, 3} Ih Lab {2, 3} Ih Lab {2, 3, 4} >. It means students work in depth to understand and solve the exercise sheet in Lab 1 or Lab 3 because they spend some hours before they move to the exercise sheet in Lab 2 or Lab 3, and so on.

2.2.8 T-PrefixSpan [33]

T-PrefixSpan is an algorithm for mining time-interval sequential patterns from electronic medical records (EMRs). It enhances the PrefixSpan algorithm [6] by focusing on the timing and efficacy of medical treatments. T-PrefixSpan manages the time intervals between treatments by considering statistical information (minimum, maximum, average, median, and most frequent value). This is crucial in medical contexts where timing can significantly affect treatment outcomes. The algorithm starts with an initial T-sequential database and a minimum support threshold. It recursively explores the database to find frequent T-sequences that meet the minimum support criteria. For each T-sequence, it generates a set of candidate sequences and evaluates their support in the database until no more candidates can be generated. Overall, the experiments demonstrated that T-PrefixSpan is a valuable tool for extracting meaningful clinical pathways from EMRs, particularly when focusing on the efficacy of treatments and managing time intervals effectively. However, the need for further evaluation by medical professionals was emphasized to ensure the clinical relevance of the findings. While T-PrefixSpan handles time intervals well, it

may struggle with highly dynamic or variable time intervals between treatments, which can occur in complex medical cases.

2.2.9 T-CSpan [34]

The paper presents a novel approach to generating clinical pathways from electronic medical records (EMRs) using a sequential pattern mining algorithm called T-CSpan. The primary challenges addressed include the slow generation of clinical pathways due to the presence of duplicate patterns and the time-consuming nature of human-based pathway creation. The T-CSpan algorithm enhances the existing CSpan algorithm [35] by incorporating time intervals between events and focusing on closed sequential patterns, which are sequences that cannot be extended by adding more items without losing their frequency. This method significantly reduces execution time, making the generation process more efficient. Also, T-CSpan employs an occurrence-checking method that efficiently identifies and includes only the longest frequent sequences during the mining process. This method helps in the early detection of closed sequential patterns, thereby reducing the computational load and execution time.

The authors conducted experiments using real medical treatment data from the University of Miyazaki Hospital, specifically focusing on two clinical pathways: cryptorchidism fusion surgery (CFS) and transurethral resection of a bladder tumor (TUR-Bt). The results demonstrated that T-CSpan improved execution times across various support values while producing clinical pathways that closely align with typical flows established by medical professionals. The study concludes that the T-CSpan algorithm significantly enhances the efficiency of clinical pathway generation, which can improve decision-making in medical settings.

2.2.10 SID-PrefixSpan [11]

This study evaluated a method for analyzing sequence pattern variants (SPVs) in the context of electronic medical records (EMRs). SID-PrefixSpan is an extension of the PrefixSpan algorithm [36] specifically designed for mining sequential patterns while retaining sequence ID (SID) information. This approach is particularly useful in contexts like EMRs, where it is important to track the original sequences from which the patterns are derived. The algorithm takes a sequence database (SDB) and a minimum support threshold (MinSup) as input. The SDB consists of sequences that may include time information and other relevant attributes. Like the original PrefixSpan, SID-PrefixSpan uses a prefix projection approach. It starts with a prefix (initially empty) and projects the database into smaller subsequences based on this prefix. As the algorithm processes each prefix, it retains the sequence IDs of the original sequences that contribute to the frequent patterns. This is crucial for later analysis, as it allows researchers to trace back the patterns to their source sequences. The algorithm identifies frequent patterns that meet or exceed the specified MinSup. It does this by recursively exploring the projected sequences and counting occurrences. In summary, SID-PrefixSpan enhances the PrefixSpan algorithm by incorporating sequence ID retention, making it a powerful tool for analyzing sequential patterns in contexts where tracking the origin of data is critical.

2.2.11 SP-TCSpan [12]

An algorithm named SP-TCSpan was proposed in this paper for analyzing electronic medical records (EMR) to extract frequent disease-treatment sequences. The study focuses on identifying medical-order sequence variants (SPVs) and the background factors influencing these variants, which include patient demographics, hospitalization details, and specimen test results. SP-TCSpan is designed to be computationally efficient, allowing for the mining of frequent

patterns while maintaining the sequence and position identifiers of items within those patterns. This efficiency is crucial when dealing with large datasets typical in healthcare settings. The algorithm takes as input a T-sequential database (D) that contains multiple T-sequences, where each sequence consists of items with associated time values. It also requires a specific T-sequence (seq) and a minimum support threshold (Min_sup) to determine the frequency of patterns. First, it creates a projected database (D'|seq) based on the input sequence. This projected database excludes time interval information, allowing the algorithm to focus on the sequence structure while still retaining the necessary identifiers. SP-TCSpan generates single frequent items (sfi) from the projected database. Each frequent item is associated with a sequence-position set (SP_sfi), which includes pairs of sequence and position identifiers. This means that the algorithm keeps track of where each item occurs within the sequences. For each single frequent item, the algorithm constructs closed frequent sequences. A closed frequent sequence is defined as a sequence that has no super sequence with a larger support value. This step helps in reducing the number of patterns generated, focusing only on the most relevant ones. SP-TCSpan includes a mechanism to calculate statistical values related to the time intervals between frequent items in the extracted patterns. This is crucial for understanding the timing of medical treatments and their implications. The algorithm ensures that only closed frequent sequences with the specified prefix (seq) are included in the final output set of T-frequent sequential patterns (SetT_SP). This filtering step enhances the relevance of the extracted patterns for clinical analysis. The authors conclude that their methods can significantly aid in understanding treatment variations and improving healthcare delivery. They suggest further exploration of the implications of their findings in clinical settings and the potential for broader applications in healthcare analytics.

Algorithm	Input	Output	Pattern Format and Explanation	Different between these patterns and Timed Sequential Patterns
I-Apriori **I-PrefixSpan** [10]	Minimum support min_sup Sequence DB Time Intervals TI= {I_1, I_2,..}	Time-interval sequential patterns within the given time interval. (Whose supports are more than or equal to min_sup.)	*(b I_1 e I_0 c)* Having bought a laser printer (*b)*, a customer returns to buy a scanner (*e)* in 3 months (I_1) and then a CD burner (*c)* in 6 months (I_0).	Time intervals is a user-defined parameter. Time intervals between two successive items.
T-pattern [9]	Trajectory dataset Minimum support min_sup A grid placed in the spatial dimension of trajectories Radius for spatial neighborhoods Temporal threshold	Set of trajectory patterns	Railway Station -> [After 15 min] Castle Square -> [After 2h] Museum A taxi that leaves from station to the Castle Square arrives in 15 minutes, then, arrives at the Museum in 2 hours.	Algorithm can deal only with trajectory data. The location is considered as one item in the event. Thus, the event can't have multi-items. Time intervals between two successive items. A new user-defined parameter called IL
MI-Apriori **MI-Prefix**				

[24]	Minimum support min_sup Sequence DB Time Intervals TI ti_0: (within 1 day), ti_1: (between 1 day and 8 days), ti_2: (between 9 days and 16 days), and ti_3: (over 17 days).	Multi-time-interval sequential patterns whose supports are more than or equal to min_sup.	*<a, ti_1, b, (ti_2, ti_1), c, (ti_3, ti_2, ti_1),d>* The list of intervals (ti_3, ti_2, ti_1) before item *d* means the intervals between items *a, b*, and *c* and item *d* are *ti_3, ti_2* and *ti_1*, respectively. *(Time-intervals between two non-successive items)*	Time intervals is a user-defined parameter. Time intervals between two successive items.
STARM-TS [25]	Minimum support min_sup Raw Trajectory Data TS User-defined interesting locations IL	Maximal trajectory stream patterns	Railway Station -> [After 10 min to 60 min] Castle Square -> [After 2h to 4h] Museum A taxi that leaves from station to the Castle Square arrives from 10 min to 60 minutes, then, arrives at the Museum from 2 to 4 hours.	Algorithm can deal only with trajectory data. The location is considered as one item in the event. Thus, the event can't have multi-items. Time intervals between two successive items. A new user-defined parameter called IL
CAI-PrefixSpan [31]	Minimum support min_sup	All the frequent timed sequential patterns satisfying both the	*<A, B I_0, E>*	Time intervals is a user-defined parameter.

	Minimum confidence min_conf Set of accumulated intervals TI Projected timed-sequential DB for sequential patterns.	min_sup and min_conf conditions.	After a customer's purchasing of products *A* and *B,* a retailer would like to know the percentage that he/she would return to buy product *E* within a week ($I_{0)}$	It has a new user-defined parameter called Minimum confidence min_conf.
Alzaharni's algorithm [32]	Mini mum support min_sup Patients' medical data User-defined time intervals	Time intervals based frequent sequential patterns	(Dc -> De) $_{[t, t/]}$ Disease $_{c}$ and Disease $_{e}$ occurred frequently in the time frame [t, t/] and De follows Dc", Then Dc can be considered as cause and thus De can be its effect within the time interval [t, t/].	Time intervals is a user-defined parameter. Time intervals between two successive items.
t_i-pattern model [4]	Minimum support min_sup Sequence DB Time Intervals TI= $\{I_1, I_2,..\}$	Time-interval sequential patterns within the given time interval. (Whose supports are more than or equal to min_sup.)	<Lab $_{\{1,3\}}$ I_h Lab $_{\{2,3\}}$ I_h Lab $_{\{2,3,4\}}$ > Students work in depth to understand and solve the exercise sheet in lab 1 or lab 3 because they spend some hours before they move to	Same as I-PrefixSpan because this model is built based on this algorithm. Time intervals is a user-defined parameter. Time intervals between two successive items.

			the exercise sheet in lab 2 or lab 3 and so on.	
T-Prefix [33]	Minimum support min_sup T-sequential database	Set of T-frequent sequential patterns	$P =< i_1, X_1, i_2, X_2, \ldots\ldots, i_{n-1}, X_{n-1}, i_n> \forall j\ i_j$ is an item, $\forall k\ X_k$ is a set of five values: min_k: minimum time interval mod_k: most frequent time interval ave_k: average time interval med_k: median time interval max_k: maximum time interval Considering the clinical pathway: <Specimen Test, (2,3,3,3,5), Radiation Test, (3,5,5,5,7), Injection>, the Specimen Test result took in minimum 2 days, usually 3 days, 3 days in average, 3 days in intermediate, and in maximum 5 days until the Radiation Test can be done. Then after minimum 3 days, usually 5 days, 5 days in average, 5 days in intermediate, and in	Do not consider the case of all possible occurrences of a pattern in the database. Do not update the time intervals between events in a pattern. Do not deal with different types of database modifications Do not handling large-scale sequence databases

			maximum 7 days, the doctor can give the injection.	
T-CSpan [34]	Minimum support min_sup T-sequential database	Set of close T-frequent sequential patterns	P =< i_1, X_1, i_2, X_2 , ……., i_{n-1}, X_{n-1} , i_n> ∀ j i_j is an item, ∀ k X_k is a set of five values: min_k: minimum time interval mod_k: most frequent time interval ave_k: average time interval med_k: median time interval max_k: maximum time interval Considering the clinical pathway: <Specimen Test, (2,3,3,3,5), Radiation Test, (3,5,5,5,7), Injection>, the Specimen Test result took in minimum 2 days, usually 3 days, 3 days in average, 3 days in intermediate, and in maximum 5 days until the Radiation Test can be done. Then after minimum 3 days, usually 5 days, 5 days in average, 5 days in intermediate, and in	Find only closed not complete set of T-sequential patterns Do not consider the case of all possible occurrences of a pattern in the database. Do not update the time intervals between events in a pattern. Do not deal with different types of database modifications Do not handling large-scale sequence databases

			maximum 7 days, the doctor can give the injection.	
SID-PrefixSpan [11]	Minimum support min_sup Sequence database	Set of frequent sequential patterns and a union of the original sequences that contain each pattern.	{P, Set $_{sidp}$ } P =< i_1, X_1 , i_2, X_2 , ……., i_{n-1}, X_{n-1} , i_n> ∀ j i_j is an item, ∀ k X_k is a set of five values. Set $_{sidp}$ = original sequences IDs that contain P. Considering the clinical pathway: {<Specimen Test, (2,3,3,3,5), Radiation Test, (3,5,5,5,7), Injection>, <1,2>}, the Specimen Test result took in minimum 2 days, usually 3 days, 3 days in average, 3 days in intermediate, and in maximum 5 days until the Radiation Test can be done. Then after minimum 3 days, usually 5 days, 5 days in average, 5 days in intermediate, and in maximum 7 days, the doctor	Do not consider the case of all possible occurrences of a pattern in the database. Do not deal with different types of database modifications Do not handling large-scale sequence databases

			can give the injection. The last set of numbers included in the output specification is the SIDs of the original sequences containing the pattern.	
SP-TCSpan [12]	Minimum support min_sup T-sequential database	Set of close T-frequent sequential patterns	{P, Set $_{sidp}$ } P =< i_1, X_1 , i_2, X_2 , ……., i_{n-1}, X_{n-1} , i_n> ∀ j i_j is an item, ∀ k X_k is a set of five values. Set $_{sidp}$ = original sequences IDs that contain P. Considering the clinical pathway: {<Specimen Test, (2,3,3,3,5), Radiation Test, (3,5,5,5,7), Injection>, <1,2>}, the Specimen Test result took in minimum 2 days, usually 3 days, 3 days in average, 3 days in intermediate, and in maximum 5 days until the Radiation Test can be done. Then after minimum 3 days,	Find only closed not complete set of T-sequential patterns Do not consider the case of all possible occurrences of a pattern in the database. Do not update the time intervals between events in a pattern. Do not deal with different types of database modifications Do not handling large-scale sequence databases

			usually 5 days, 5 days in average, 5 days in intermediate, and in maximum 7 days, the doctor can give the injection. The last set of numbers included in the output specification is the SIDs of the original sequences containing the pattern.	

Table 2.1: Comparison of existing timed sequential patterns techniques

2.3 Literature Review of Mining Sequential Patterns from Dynamic Databases

This section reviews state-of-the-art techniques for extracting sequential patterns from dynamic databases. It is organized around the issues identified in Section 1.5 of Chapter I. We classified the surveyed techniques into two categories based on the updating operation applied to the sequence database: incremental and progressive databases.

2.3.1 Mining Sequential Patterns in Incremental Databases

Incremental databases mean new data are added as new sequences or tuples in an existing sequences database. This type of mining is known as Incremental Sequential Patterns Mining (ISPM) in the literature. In the following, we present a survey of some discovering sequential patterns in incremental database algorithms.

2.3.1.1 FastUP [37]

One of the first approaches developed in the field of ISPM is the FastUP approach. Its foundation is the GSP technique, with improvements to support counting and candidate generation. Before candidate generation and validation, the algorithm employs the pruning method, which obtains information about the sequence thresholds by using the prior mining results. Therefore, it can avoid producing a specific sequence based on their support. Additionally, it uses methods for quickly filtering during the counting of supports and the reduction of successive candidate sequences. By doing this, the algorithm's overall time and space efficiency surpasses that of the SPM algorithm.

2.3.1.2 ISM [38]

This algorithm follows a similar approach to SPADE [8] and focuses on maintaining the sequence lattice of a previous database. The lattice comprises sequences that are frequent or within the negative border, which is helpful for the algorithm in keeping the database updated. In the first

database scanning, ISM performs lattice pruning and generates negative border sequences. Then, the algorithm generates the new frequent sequences using the SPADE approach. Since the negative border can potentially contain many sequences, this technique may require a significant amount of memory and time to execute. However, if the minimum support is low, it is highly improbable for sequences in the negative border to become frequent in the updated database.

2.3.1.3 IUS [39]

Because of the high computing cost of ISM [38], IUS was introduced that shares a similar approach to SPADE [8]. The IUS reused frequently occurring and negatively bordering sequences within the original database. The method defines a new threshold, known as the minimal support of negative border sequences (min_nbd_Supp), to regulate the memory and time resources used by ISM. Consequently, the frequent sequences that have a support value more than min_nbd_Supp and lower than the min_sup are classified as negative border sequences. The generation of candidates from the original database by IUS is accomplished by expanding both the prefix and suffix of frequent sequences.

2.3.1.4 ISE [40]

The algorithm aims to reduce computing complexity by iteratively utilizing previously identified frequent sequences and employing optimization approaches. The ISE method demonstrates superior performance compared to the naïve approach, the GSP algorithm [5]. However, the algorithm in question exhibits two primary drawbacks. First, the candidate set has the potential to become exceedingly large, resulting in a significant slowdown during the test phase. Second, the algorithm demands several scans of the whole database, which can be quite expensive, particularly when dealing with lengthy sequences.

2.3.1.5 GSP+ and MFS+ [41]

The authors propose algorithms that can update frequent sequences efficiently by taking advantage of the properties of the sequences. They introduce the concept of sequence difference to represent the changes between two sequences and use it to update the frequent sequences incrementally. Two algorithms are proposed: the GSP+ (Generalized Sequential Pattern Plus) which is a modified version of GSP [5] and the MFS+ (Maximal Frequent Sequence Plus) algorithm, which is an extension of MFS [42]. GSP+ starts identifying the 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 recalculating everything from scratch. GSP+ uses the concept of sequence difference to represent the changes between the old and new sequences. This allows the algorithm to efficiently update the frequent sequences by considering only the changes instead of reprocessing the entire database. MFS+ starts identifying the maximal frequent sequences in the initial database using the MFS algorithm. Maximal frequent sequences are sequences that are frequent and cannot be extended further without becoming infrequent. When the database is updated with new transactions or modifications, MFS+ incrementally updates the maximal frequent sequences without the need to recompute everything from scratch. Like GSP+, MFS+ uses the concept of sequence difference to represent the changes between the old and new sequences. This allows the algorithm to update the maximal frequent sequences efficiently by considering only the changes in the database. The drawbacks of the GSP+ and MFS+ algorithms can include overhead and scalability. While the algorithms aim to reduce computational overhead by incrementally updating frequent sequences, there may still be some overhead associated with managing and processing the sequence differences. The scalability of GSP+ and MFS+ algorithms for very large databases or sequences

may be limited, as the incremental update process could become more computationally intensive with larger datasets.

2.3.1.6 IncSP [43]

Incremental update on sequential patterns in large databases by implicit merging and efficient counting IncSP uses a combination of early candidate reduction, and efficient separate counting techniques. The process of integrating newly updated sequences with previously valuable sequences is performed implicitly. The process of candidate pruning involves examining the counts in the increment database to generate a reduced set of candidates that are of higher quality. However, the process of counting potential candidates over the original database yields new and updated patterns. Also, IncSP identifies new patterns from the updated database in a continual manner. While the IncSP (Incremental Sequential Pattern Update) algorithm offers several advantages in terms of efficiency and scalability for updating sequential patterns in large databases, there are also some potential disadvantages to consider. The algorithm may still require significant memory resources, especially when dealing with large databases or high increment ratios. This could lead to memory constraints and performance issues, particularly in resource-constrained environments. Also, when the size of the incremented database surpasses the original database significantly, the efficiency of IncSP may decrease. In such cases, where the database undergoes substantial changes rather than incremental updates, remining the entire database could be more appropriate. In addition, the algorithm heavily relies on previously discovered patterns and knowledge for incremental updating. If the existing patterns become outdated or irrelevant due to significant changes in the database, the effectiveness of IncSP in updating patterns may diminish.

2.3.1.7 FS-Miner [44]

FS-Miner is an efficient and incremental mining tool designed to extract frequent sequence patterns from web logs. It uses the FS-tree structure to store essential information about frequent sequences, allowing for quick analysis and discovery of patterns. The FS-tree, or frequent sequence tree, is a data structure consisting of three main components. First, the tree includes a special root node and a set of sequence prefix subtrees as children. Each node in the FS-tree represents an item from the input database, and edges in the tree represent link relationships between nodes. Each edge has fields for edge-name, edge-count, and edge-link. Second, the header table (HT) stores information about frequent and potentially frequent links in the database. Each entry in the header table contains fields for the link name, the count of the link in the database, and a list head pointer pointing to the first edge in the tree with the same edge-name. Third, the non-frequent links table (NFLT) stores information about non-frequent links and is used to support the incremental feature of the system. It includes fields for the link name, count of the link in the database, and IDs of tuples in the database that include that link.

FS-Miner begins with an initial scan of the database to obtain support counts for subsequences of length two. This step helps in identifying potentially frequent sequences in the dataset. After the initial scan, FS-Miner constructs a compressed data structure called the FS-tree. The FS-tree stores potentially frequent sequences in a compact form, facilitating efficient mining of frequent patterns. FS-Miner performs a second scan of the database to extract potentially frequent sequences of any length. These sequences are represented in the FS-tree, which enables the mining of frequent sequence patterns from the tree. The algorithm supports incremental and interactive mining functionalities. It can adapt to changes in user behavior over time by updating the FS-tree and discovered patterns incrementally without the need for full re-computation.

2.3.1.8 BSPinc [45]

The BSPinc algorithm uses a backward mining approach for the incremental discovery of sequential patterns. This technique aims to efficiently mine patterns by identifying stable sequences and eliminating them from the support counting process. By generating candidate sequences in a backward manner and utilizing stable sequence pruning, BSPinc can significantly speed up the mining process. The following is an overview of how BSPinc works. BSPinc enhances the SPAM [21] mining framework with backward mining. It uses a bitmapped representation and starts by obtaining the set of 1-patterns after scanning the updated database (UD) once. Then it identifies stable sequences whose support counts remain unchanged in the updated database. These stable sequences are then eliminated from the support counting process, reducing computational overhead. Candidate sequences are generated using backward extensions from the 1-patterns. BSPinc recursively generates candidate (k+1)-sequences from k-patterns and counts them to determine their frequency. BSPinc introduces a unique property that any extension of a stable sequence must also be stable. This property allows for skipping the support counting step of stable sequences, speeding up the incremental mining process.

By combining backward mining with stable sequence pruning, BSPinc efficiently handles incremental pattern discovery in sequence databases, offering significant improvements in speed and performance compared to forward mining approaches.

2.3.1.9 MI-Apriori and MI-Prefix [24]

The shortcomings of previous studies in general as well as those of I-Apriori and I-PrefixSpan in particular are addressed in this work. The general methods used in previous publications are to set time constraints on the patterns or find the time intervals between two successive items. Unfortunately, these methods are unable to obtain the time interval data in a

sequential pattern between non-successive items. A new method was proposed to address the shortcomings of the methods that were already in use. Some possible expansions included the use of rough set theory or fuzzy theory, the creation of a hierarchy of time intervals, and the imposition of meta-rule constraints on the patterns that were formed. To expose the time intervals between each pair of items in the pattern, the I-Apriori and I-PrefixSpan algorithms were modified accordingly to create the MI-Apriori and MI-PrefixSpan algorithms. The inputs for these algorithms are: Minimum support min_sup, Sequence DB, and Time Intervals TI={ ti_1, ti_2, … ti_{r-1}}, and the output is the set of multi-time-interval patterns which each pattern denoted as A= $<a_1, ti_1, a_2, (ti_2, ti_1), a_3, (ti_3, ti_2, ti_1), \ldots, a_{s-1}\ (ti_{s-1} \ldots ti_3, ti_2, ti_1)\ a_s>$, where a_i represents an item and (ti_i) represents the time interval between item a_1 and item a

2.3.1.10 NIA [46]

Due to the challenges of processing data from wireless sensor networks, a novel algorithm called NIA (Novel Incremental Algorithm) is designed to address these issues. First, the PrefixSpan+ is introduced, which is an enhancement of the original PrefixSpan algorithm, to improve mining efficiency. PrefixSpan suffers from the high computation cost of building the projected database and reconstructing it recursively. Thus, a data structure called Co-occurrence MAP (CMAP) [47] is used to extend the prefixes to frequent sequential patterns. The CMAP structure consists of two main components. CMAPi maps each item k to the set of items j that succeed k by i-extension in a minimum number of sequences in the sequence database, and CMAPs maps each item k to the set of items j that succeed k by s-extension in a minimum number of sequences in the sequence database. Then, another novel structure called a reticular sequence tree is proposed to speed-up the support count calculation for the potential sets of frequent sequence. Each node represents a potential frequent sequence in the reticular sequence tree, and child nodes

are subsequences of their parent nodes. If a sequence is a subsequence of another, it must be in the reticular sequence tree with the parent sequence as the root node. Layers of sequences are arranged in the tree according to length in the reticular sequence tree creation process. The first layer is made up of the longest sequences, and each node acts as a root node. Subsequent layers are then checked to determine if sequences are subsequences of nodes in higher layers. If so, they are added as child nodes; otherwise, they become root nodes.

2.3.1.11 MR-INCSPM [48]

With the increase in big data, sequential pattern mining algorithms suffer from memory limitations and decreases in performance. This work leverages the MapReduce framework to efficiently handle the processing of large datasets and achieve scalability. MR-INCSPM is built based on the wo-phase approach and introduce the two Co-occurrence Reverse MAP CRMAP data structures. The first one is $CRMAP_s$, which is mapping each item to a set of items that precede it in the sequence-extension in no less than min_sup threshold. The second one is $CRMAP_i$, which is mapping each item to a set of items that precede it in the event-extension in no less than a specified minimum support threshold. This mapping helps in identifying promising candidates that are likely to appear in the input database, thereby reducing the search space and improving the overall efficiency of the mining process.

The first phase focuses on efficiently generating candidate sequences by incorporating a backward mining strategy [45] for discovering sequential patterns from incremental DB and utilizing CRMAP data structures to map items in an input sequence to their preceding items, aiding in candidate generation and pruning to reduce the search space. The second phase efficiently processes the generated candidate sequences, performs support counting, and identifies frequent

sequential patterns. Experimental results showcase the algorithm's performance compared to non-incremental MapReduce algorithms, highlighting its speedup and linear scalability.

2.3.1.12 ISPTAR [48]

To handle temporal datasets efficiently, the PrefixSpan approach is combined with temporal association rule mining in the Incremental Sequential Patterns for the Multivariate Temporal Association Rules (ISPTAR) mining algorithm. The technique integrates historical and incremental temporal data to tackle the problem of mining sequential patterns in dynamic datasets. ISPTAR reduces the need for repetitive historical database scans by keeping previously mined information and combining it with fresh incremental data. This increases mining efficiency.

The ISPTAR does the following to mine the temporal relationship between the sequential patterns for multivariate time series. First, a temporal sequence dataset is created from the temporal transaction dataset that was produced via fuzzy discretization of multivariate time series. Second, the valid time period of the pattern and the temporal relationship between elements in the pattern are taken into consideration when mining the fuzzy temporal sequential patterns, which are based on the PrefixSpan method. Third, fuzzy temporal association rules can be constructed to mine the association between different attributes based on the mined fuzzy temporal sequential patterns. In addition, during the sequential pattern mining process, when incremental data are added, the proposed algorithm can update the sequential patterns by merging the information mined in the historical database and incremental database without rescanning the historical database. ISPTAR uses more memory space than other algorithms for finding fuzzy temporal association rules, despite testing results showing that it performs better than other algorithms in terms of mining efficiency and the number of rules generated.

2.3.1.13 IncSeq [49]

This work discusses the Incremental Sequence (IncSeq) framework for mining frequent serial episodes over data streams. It addresses the challenge of efficiently extracting frequent sequential patterns without starting from scratch each time new data arrives and consider if the patterns occur in many events in a sequence. The paper introduces a new incremental sequential pattern mining problem and provides an algorithm for efficient mining with search space pruning. IncSeq initializes 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 events 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 event from the data stream, two actions are triggered: (1) occurrences associated with the first event in the window are deleted; and (2) patterns and occurrences associated with the new event 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-events of the new event into the current tree; and trimming nodes with non-frequent patterns. Therefore, to minimize the size of the tree before the computationally demanding merging and completion sub-steps, their approach executes the deletion phase before the insertion of a new event. 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.

2.3.2 Mining Sequential Patterns in Progressive Databases

Progressive databases mean new data are added into the database and obsolete data are removed. This type of mining is known as Progressive Sequential Patterns Mining (PSPM) in the literature. In the following, we present a survey of some discovering sequential patterns in progressive database algorithms.

2.3.2.1 Pisa [50]

To solve the limitations of mining sequential patterns from static databases, this work discusses the concept of progressive sequential pattern mining. The proposed Progressive mIning of Sequential pAtterns algorithm Pisa aims to discover sequential patterns in recent time periods of interest by considering simultaneously both new data additions and the removal of obsolete data from the database. A Period of Interest (POI) concept represents a sliding window of time for analyzing sequences. The sequences containing elements with timestamps falling within the POI are considered for analysis, while sequences with timestamps older than the POI are pruned from the database. By pruning obsolete data and patterns, the algorithm ensures that only up-to-date information contributes to the mining process. Pisa manages sequences in a dynamic database by keeping track of a data structure known as the Progressive Sequential tree (PS-tree). Pisa starts by initialized PS-tree to store sequences fall within the POI and traverses the PS-tree of timestamp t in post-order when new elements arrive at timestamp t+1. All obsolete elements are deleted, existing sequences are updated, new elements are inserted into the PS-tree of timestamp t based on the incoming data, and the PS-tree is iterated. For root nodes, Pisa examines all sequences with new elements and creates combinations of candidate elements from the incoming data to find the complete set of up-to-date sequential patterns and delete obsolete data and patterns accordingly. After that, Pisa moves to the next timestamp and carries out the traversal and updating procedure

again until no more new data is added to the database. Furthermore, a fast variant of Pisa is suggested, which maximizes the processing of sequential patterns to drastically cut down on memory usage and execution time.

2.3.2.2 Wtd_Seq_Pat [51]

This paper claimed that most updated frequent sequential patterns can be obtained by using any existing progressive sequential pattern mining approach. There is no sequential pattern mining algorithm that offers a metric to evaluate the significance of the sequential patterns that are mined. The assumption is the likelihood that the pattern will occur decreases with an increase in the pattern's span. Also, the weight is assigned to the timestamps to promote practical utilization; that is, the presence or absence of a pattern on a timestamp may aid in weighing the pattern. A novel technique to assign weights to sequential patterns based on their spread over time is proposed called Wtd_Seq_Pat; it addresses the issue of quantifying the importance of extracted sequential patterns. By considering the time period over which a pattern spans, the Wtd_Seq_Pat adjusts the support count of a pattern to reflect its significance accurately. The weighted M-ary tree (WM-ary tree) is a modified version of the M-ary tree. The M-ary tree is a tree data structure to represent sequences of elements over time. After modification to a WM-ary tree, it stores information about node labels, sequence lists, timestamp lists, original timestamps, and time gaps between elements in the pattern. The weighted M-ary tree (WM-ary tree) is, then, a modified version of the M-ary tree. The M-ary tree is a tree data structure to represent sequences of elements over time. The node structure of WM-ary tree is modified to have (1) a label, which represents the element of the sequence; (2) a sequence list, which contains a list of sequence IDs; (3) a timestamp list, which stores timestamps corresponding to each sequence; (4) an original timestamp, which denotes the actual timestamp when the element occurs in a particular sequence; and (5) time gap, which

represents the number of inter-item time gaps in the pattern. The algorithm progressively updates information about each candidate pattern and stores the results in the WM-ary tree. At each timestamp, the algorithm traverses the tree, inserting new elements and updating existing nodes to reflect the current time based on their occurrence in sequences. Nodes are added or updated with information about timestamps and time gaps to track the sequence of elements over time. The main concept used in the WM-ary tree is to keep track of time gaps and important timestamps between occurrences of elements in sequential patterns. By assigning weights based on these factors, the algorithm can prioritize and distinguish between different patterns based on their temporal characteristics. Using this approach, Wtd_Seq_Pat extracts the most recent sequential patterns that are least affected by obsolete data.

2.3.2.3 DPSP [52]

Scaling up single-core or serial algorithms proves to be difficult. As a result, this work proposes the Distributed Progressive Sequential Pattern mining algorithm (DPSP). It focuses on mining sequential patterns within each Point of Interest (POI), which refers to a specific timestamp or time interval within the sequential data being analyzed. The Hadoop platform is used for implementing this algorithm and two Map/Reduce jobs are designed: the candidate computing job, and the support assembling job. The first job, the CCMapper, generates pairs of <sequence number, candidate event with corresponding timestamp> if reading input from the candidate set summaries, and <sequence number, arriving event> pairs if reading input data from a sequence. The CCReducer computes candidate sequential patterns based on the input received from CCMapper. The second job, the SAMapper, is responsible for aggregating occurrence frequencies of different candidate sequential patterns locally. SAMapper reads all candidate sequential patterns from the outputs of the candidate computing job and uses a local map to aggregate occurrence

frequencies. It then outputs pairs of <candidate sequential pattern, its local supports>. These pairs with the same candidate sequential pattern are sent to the same SAReducer for final support accumulation and identification of frequent sequential patterns in the current Point of Interest (POI). Obviously, the scalability and performance of DPSP outperform other serial algorithms through experiments.

2.3.2.4 PPSP [53]

Because of the rapid increase in the size of data, the researchers implement a parallel version of Pisa [50] using the GPU platform called PPSP, or Parallel Progressive Sequential Pattern. The algorithm efficiently uses both the host CPU and GPU to distribute workload, achieving a speedup of more than 10× by leveraging coalesced memory accesses and SIMD execution on GPUs.

PPSP maintains a full PS-tree in the device memory. Each node in the PS-tree contains information about timestamps, sequence IDs, parent and child indexes, and valid lengths. The researchers proposed a different order to traverse the tree called a Reverse Level Wise (RLW) order. It starts from the bottom-most level and moves up to the root. At each timestamp, several threads equal to the number of active sequences are invoked. Each thread is responsible for updating the PS-tree corresponding to its sequence ID. Threads generate singular child nodes for their sequence ID, which are later merged to form the actual nodes of the PS-tree. The kernel running on the GPU accumulates the occurrence frequencies of candidate sequential patterns in the current period of interest (POI) and reports frequent sequential patterns to users. This kernel efficiently processes candidate sequential patterns generated on the host CPU and uses the computational power of the GPU for faster mining. PPSP optimizes memory transactions by ensuring that threads referencing memory in the same segment are serviced simultaneously,

maximizing bandwidth utilization. Asynchronous execution between the host and device allows for concurrent processing, ensuring that neither the host nor the device is idle at any given time. Synthetic data generation and analysis show significant performance improvements with the PPSP algorithm compared to sequential implementations.

2.3.2.5 HDCL [54]

Mining sequential patterns algorithms not only cannot handle large datasets but also face other challenges related to noisy and non-stationary data. Therefore, this paper suggests HDCL method that using the deep learning for mining sequential patterns from progressive databases. To deal with non-stationary data a mathematical technique is used for signal processing and data analysis called discrete wavelet analysis DWT. It converts non-stationary data into time series data and helps in separating the time-series components of the data by breaking down the original data into approximation and detailed coefficients representing low-frequency and high-frequency components, respectively. The convolutional neural network (CNN) is used to mine the frequent sequence pattern from the outcome of DWT [55]. The event in the sequences are arranged in priority with the help of constructing the candidate frequent 1-sequences in convolutional layer 1. For often occurring 1-sequences, the item in convolutional layer 1 whose support is larger than or equal to minimal support is chosen in max-pooling layer 1. Converting the two frequent sequences, S1 and S2, into two sequences (S1; S2) and (S2; S1) will produce the candidate frequent 2 sequences in convolutional layer 2. Once more, determine the support value for each event in max-pooling layer 2, then choose the two frequent sequences. The researchers also consider the time interval to discover the frequent patterns because having two patterns at an instance of time helps to find frequent items purchased by the customer for example. To reach this goal, Long Short-Term Memory (LSTM) is used to find the frequent time interval among each sequence pattern.

The time interval is another factor that the researchers consider when looking for frequent patterns; for example, finding two patterns at one point in time can be useful in identifying often purchased items by a consumer. To accomplish this, the frequent time gap between each sequence pattern is found using Long Short-Term Memory (LSTM). The LSTM model takes the output from the preceding frequency step and time step in DWT to produce one output after the max-pooling layer 2 generates frequent 2-sequences. It then determines the associated frequent time interval from the frequent 2-sequences. The results of this approach emphasize the importance of deep learning models in processing complex data structures and improving the accuracy of pattern extraction in various domains.

	Sequential Patterns			Timed Sequential Patterns			Dynamic Sequential Patterns			Parallel Processing
	Mining ordering of items	**Minimi-zing # database scans**	**Reducing Size of candidates set**	**Handling timestamp data**	**Finding all possible occur-rences**	**Updating temporal relation**	**Insertion operation**	**Modifi-cation operation**	**Deletion operation**	
FASTUP [37]	Yes	Yes	Yes	No	No	No	Yes	No	No	No
ISM [38]	Yes	Yes	Yes	No	No	No	Yes	No	No	No
IUS [39]	Yes	Yes	Yes	No	No	No	Yes	No	Yes	No
ISE [40]	Yes	Yes	Yes	No	No	No	Yes	No	No	No
GSP+, MFS+ [41]	Yes	Yes	Yes	No	No	No	Yes	No	No	No
I-Apriori, I-PrefixSpan [10]	Yes	Yes	Yes	Yes	No	No	No	No	No	No
IncSP [43]	Yes	Yes	Yes	No	No	No	Yes	No	No	No
FS-Miner [44]	Yes	Yes	Yes	No	No	No	Yes	No	No	No
T-pattern [9]	Yes	Yes	Yes	Yes	No	No	No	No	No	No
T-PrefixSpan [33]	Yes	Yes	Yes	Yes	No	Yes	No	No	No	No
T-CSpan [34]	Yes	Yes	Yes	Yes	No	Yes	No	No	No	No

SID-PrefixSpan [11]	Yes	Yes	Yes	Yes	No	Yes	No	No	No	No
SP-TCSpan [12]	Yes	Yes	Yes	Yes	No	Yes	No	No	No	No
Pisa [50]	Yes	Yes	Yes	No	No	No	Yes	No	Yes	Yes
BSPinc [45]	Yes	Yes	Yes	No	No	No	Yes	No	No	No
MI-Apriori, MI-Prefix [24]	Yes	Yes	Yes	Yes	No	No	No	No	No	No
Wtd_Seq_Pat [51]	Yes	Yes	Yes	No	No	No	Yes	No	Yes	No
DPSP [52]	Yes	Yes	Yes	No	No	No	Yes	No	Yes	No
STARM-TS [25]	Yes	Yes	Yes	Yes	No	No	Yes	No	No	No
CAI-PrefixSpan [31]	Yes	Yes	Yes	Yes	No	No	No	No	No	No
MR-INCSPM [47]	Yes	Yes	Yes	No	No	No	Yes	No	No	No
PPSP [53]	Yes	Yes	Yes	No	No	No	Yes	No	Yes	Yes
NIA [46]	Yes	Yes	Yes	No	No	No	Yes	No	No	No

Alzahrani's Algo. [32]	Yes	Yes	Yes	Yes	No	No	No	No	No	No
ti-pattern model [4]	Yes	Yes	Yes	Yes	No	No	No	No	No	No
ISPTAR [48]	Yes	Yes	Yes	No	No	No	Yes	No	No	No
HDCL [54]	Yes	Yes	Yes	No	No	No	Yes	No	Yes	No
IncSeq [49]	Yes	Yes	Yes	No	No	No	Yes	No	Yes	Yes
Minits+	Yes	Yes	Yes	Yes	No	Yes	No	No	No	Yes
Minits-AllOcc [56]	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes
MinitsDays	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes

Table 2.2: Feature comparison of discovering Timed/Dynamic Sequential Patterns techniques

2.4 Summary

This chapter presents state-of-the-art techniques for mining time-interval sequential patterns. Additionally, we surveyed existing techniques for mining sequential patterns that have incorporated temporal relation in the patterns and techniques for mining sequential pattern mining from incremental and progressive databases. However, none of the discussed techniques fully address all the challenges associated with mining timed sequential patterns from static or dynamic sequence databases. To address this gap in the literature, three innovative algorithms are proposed in this dissertation. Minits+, Minits-AllOcc, and MinitsDays.

Minits+ (MINIng Timed Sequential patterns) is the first approach described in this dissertation. It is designed to identify frequently occurring sequential patterns in a static discretized timed sequence database while incorporating temporal relations between events in a pattern. These intervals represent the time range required to transition between events. An enhanced version, MMinits+, has been developed for multi-core CPUs, offering improved performance for large-scale sequence databases.

The second technique is Minits-AllOcc (MINIng Timed Sequential Patterns for All-Time Occurrences). This single-core CPU algorithm is designed to perform the following tasks. First, it incorporates temporal relations between events in a sequential pattern to indicate the time required to transition between events. Second, it identifies the transition time between events in a timed sequential pattern based on all occurrences of the pattern in the discretized timed sequence database. Third, it discovers patterns from a static discretized timed sequence database. To address the issue of increased computational cost as the database size grows, a multi-core CPU variant called MMinits-AllOcc has been developed, offering enhanced efficiency and scalability.

MinitsDays (MINIng Timed Sequential Patterns in DYnamic Sequence Databases) is the third technique introduced. This algorithm is designed to incorporate temporal relations between events in a sequential pattern, and to discover patterns from a dynamic discretized timed sequence database accounting for changes in its size and content.

To enhance scalability and efficiency, particularly in dynamic environments where the database frequently changes, a multi-core CPU variant called MMinitsDays has been developed. Table 2.2 presents a feature comparison analysis of the algorithms discussed in this chapter against the challenges associated with mining timed sequential patterns from static and dynamic timed sequence data, as outlined in Chapter 1, Section 1.5. Explanations for each cell in the table can be found in the discussion section for the respective algorithms. In the table, a “Yes” indicates that the technique on the left addresses the problem listed at the top, while a “No” indicates that the technique does not address the problem.

CHAPTER 3
THE PREPROCESSING APPROACHES AND PROPOSED ALGORITHMS

This chapter begins by introducing the foundational concepts necessary for the reader to understand the ideas presented in this dissertation. It then provides an overview of the preprocessing steps required to prepare raw data for the sequential pattern mining task. Following this, a summary of our proposed algorithms is presented, along with a detailed discussion of each. Our proposed algorithms are as follows: Minits+ for discovering the complete set of timed sequential patterns from a static Discretized Timed Sequence Database; Minits-AllOcc for discovering all possible occurrences of timed sequential patterns from a static Discretized Timed Sequence Database; and MinitsDays for discovering all possible occurrences of timed sequential patterns from a dynamic Discretized Timed Sequence Database.

3.1 Preliminaries

This section recalls the main concepts of sequential pattern mining [1] and introduces the new terminologies that are related to the main topic of this dissertation, timed sequential pattern mining.

3.1.1 Sequential Pattern Mining Problem Definitions

This section reviews the terminology of the sequential pattern mining problem, which is defined as follows [1]:

Definition 3.1 (Itemset): a set of items, such that $I \subseteq X$, where $X = \{x1, x2, ... xl\}$ is a set of items in the database.

Definition 3.2 (Sequence s): is an ordered list (based on timestamps) of itemsets. The length of a sequence is the number of itemsets in it.

Definition 3.3 (Super Sequence): A sequence A = < $\{a_1\}$, $\{a_2\}$, … $\{a_n\}$ > is contained in another sequence B = < $\{b_1\}$, $\{b_2\}$, … $\{b_m\}$ > and A is a subsequence of B, or 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 $a_1 \subseteq b_{j1}$, $a_2 \subseteq b_{j2}$, ….., $a_n \subseteq b_{jn}$.

Definition 3.4 (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.

Definition 3.5 (Support): The support of a sequence A in a sequence database is the percentage of the number of sequences in the database that contains A, such that sup(A) = (number of sequences that contain A/number of sequences in a database) × 100.

Definition 3.6 (Sequential Pattern): 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.

Definition 3.7 (Event relation e-relation): Given two items x and y, it is said x and y have an e-relation between them denoted as < {x, y}> if x and y occur in the same itemset at the same timestamp.

Definition 3.8 (Sequence relation s-relation): Given two items x and y, it is said that x and y have an s-relation between them denoted as < {x}, {y}> if x and y occur in two different itemsets at different timestamps, and the itemset of x occurs before the itemset of y.

3.1.2 Timed Sequential Patterns Mining Problem Definitions

In this section new definitions for the timed sequential pattern mining problem are introduced as follows:

Definition 3.9 (Event): is a pair e = (t, I), where I is an itemset that occurs at the timestamp t. Notations e.t and e.I are used to indicate respectively the timestamp t and the itemset I associated with the event e.

Figure 3.1 illustrates a sample Discretized Timed Sequence Database (DTSDB). For simplicity, letters are used to refer to items that represent different properties of objects in the database (e.g., temperature and speed of wind), and integer numbers are used 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 {5, a, b,}, which occurred at timestamp 5, followed by the second event {12, d, g}, which occurred at timestamp 12.

Definition 3.10 (Timed Sequence): The list of events that is sorted in the timestamp order. TS = $< \{e_1\}, \{e_2\}, ..., \{e_n\} >$, such that $e_i.I.x \subseteq I$ $(1 \leq i \leq n)$.

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}>

Figure 3.1: An example of Discretized Timed Sequence Database DTSDB

Definition 3.11 (Timed Sequence Database TSDB): is a set timed sequences < TS_id, TS >, where TS_id is a timed sequence identifier and TS is a timed sequence. A Discretized Timed Sequence Database (DTSDB) is a timed sequence database in which continuous numerical data has been transformed into discrete ordinal values using discretization methods to enable timed sequential pattern mining.

Definition 3.12 (All-time Occurrences): 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 Discretized Timed Sequence Database DTSDB is defined as a list of all possible ordered set of indices $1 \leq j_1 < j_2 < \ldots < j_n \leq m$, such that: $I_1 \subseteq e_{j1}.I$, $I_2 \subseteq e_{j2}.I$, ……, $I_n \subseteq e_{jn}.I$.

Definition 3.13 (Delta): *Δ* is defined as the time difference between two events in an ordered set of indices such that $\Delta(e_i, e_j) = t_j - t_i$ for I < j, where t_i and t_j are the timestamps associated with events ei and ej, respectively.

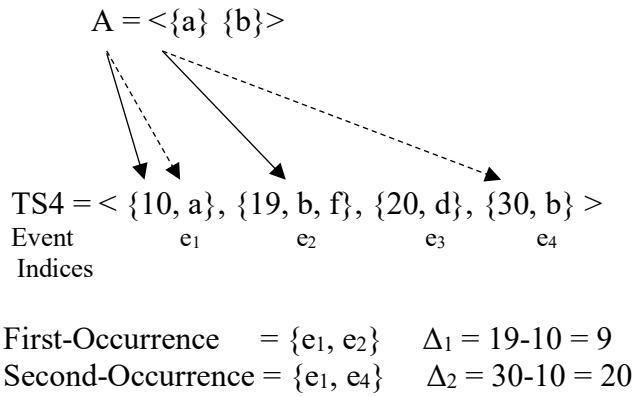


Figure 3.2: All-time Occurrence of A in TS4

Let sequence A = < {a}, {b} > and timed sequence TS4 = < {10, a}, {19, b, f}, {20, d}, {30, b} >, as shown in Figure 3.2. The first ordered list of indices of the events for the first occurrence of sequence A in TS4 is $\{e_1, e_2\}$, as shown by the solid arrow in Figure 3.2. The delta **Δ** is the difference between the timestamps of these two consecutive events, which is $e_1.t = 10$ and $e_2.t = 19$. Thus, the **Δ** = 19 – 10 = 9. Then, the second occurrence of sequence A in TS4, as shown by the dotted arrow in Figure 3.2, has the indices of the event $\{e_1, e_4\}$. The delta **Δ** is the difference between the timestamps of these two consecutive events, which is $e_1.t = 10$ and $e_4.t = 30$. Thus, the **Δ** = 30 – 10 = 20.

Definition 3.14 (Timed Sequential Patten TSP): 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, median, maximum, and minimum as range of transition time between itemsets, calculated based on the values of the delta **Δ**. TSP is denoted as: TSP = < $\{I_0\}$ $[\tau_1]$ $\{I_1\}$ $[\tau_2]$ $\{I_2\}$ … $[\tau_n]$ $\{I_n\}$ >.

Assuming the min_sup = 50%, since the support of sequence A = < {a}, {b} > is 50%, the sequence is a sequential pattern. We assumed 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 temporal relation between itemsets.

3.2 Data Preprocessing

Preparing raw data for analysis is referred to as data preprocessing in data mining. It involves cleaning, transforming, and organizing the data to make it suitable for analysis and

mining. This stage is crucial because inconsistencies, errors, missing values, and noise commonly found in raw data can significantly affect the effectiveness of data mining algorithms.

Sequential pattern mining can only handle categorical data, which is grouped into categories rather than measured numerically. Therefore, when we collected real datasets for the experiments, we first discretized them using well-known techniques relevant to their respective fields. For example, the Oklahoma Mesonet dataset was used [22, 23], gathered through collaborative efforts by OU and OSU scientists through weather monitoring stations. In this dataset, one of the attributes is wind, which was discretized using the well-known Beaufort scale, a standard in meteorology [57]. However, we could not find a standard and precise discretization technique for the T-Drive dataset [58, 59], a collection of trajectories gathered by Microsoft Research. Consequently, we conducted the study below to identify the most suitable discretization method for different trajectory datasets.

3.2.1 Introduction to Trajectory Sequential Patterns

The widespread use of mobile devices and advancements in location-based services have resulted in the collection of vast amounts of spatio-temporal data across various applications. This data captures the movements of objects (e.g., vehicles, people, or animals), including their latitudes, longitudes, and timestamps at specific locations. Consequently, the sequence of spatial coordinates paired with timestamps, referred to as a trajectory, can be recorded using these devices.

Understanding the movement behavior of objects provides valuable insights, enabling us to analyze, understand, and predict their frequent patterns. As a result, discovering frequent sequential patterns from trajectory data has become a prominent area of research [13], [22], [47], [60, 61]. For instance, two trajectories, T1 and T2, consist of the following sequences of locations l_i:

$$T1: l_1 \rightarrow l_2 \rightarrow l_8 \qquad T2: l_1 \rightarrow l_2 \rightarrow l_4 \rightarrow l_8$$

We can see that these two trajectories have a common sequence $l_1 \rightarrow l_2 \rightarrow l_8$ of visiting order [60]. If the percentage of the tuples in the database that contain this pattern, which is called support, exceeds the given minimum support threshold (min_sup), the pattern is reported as a frequent sequential pattern.

The spatial and temporal information extracted from trajectories can enhance the effectiveness and usefulness of many applications. For instance, in tourism, identifying frequently repeated paths in a specific geospatial region can improve recommendation systems, allowing tourists to visit locations and follow suggested sequences of popular destinations [62]. Given the complex nature of trajectory data, dividing them into smaller, disjointed, and less complex sub-trajectories based on specific criteria can facilitate the extraction of trajectory sequential patterns. This process is known as trajectory segmentation.

The following example demonstrates the advantage of using this mechanism. Suppose we have three trajectories representing the movements of three people heading to a café (depicted as a dotted circle in Figure 3.3). Although all three individuals are at the same café, each occupies a different location within it because they are seated at different tables, and their timestamps reflect when they arrived. As a result, while visiting the café is a common activity for these individuals, the actual data points associated with the café in their trajectories differ in both coordinates and timestamps. Consequently, a sequential pattern mining algorithm would fail to identify the repeated activity of visiting the café, as each occurrence is recorded with distinct data points (coordinates and timestamps). However, if the general location (the café's coordinates) were recorded instead of precise individual positions, the algorithm could more easily identify the repeated pattern of visiting

the café. Thus, while the specific coordinates of these points may differ, they can still be recognized as belonging to the same general place [63].

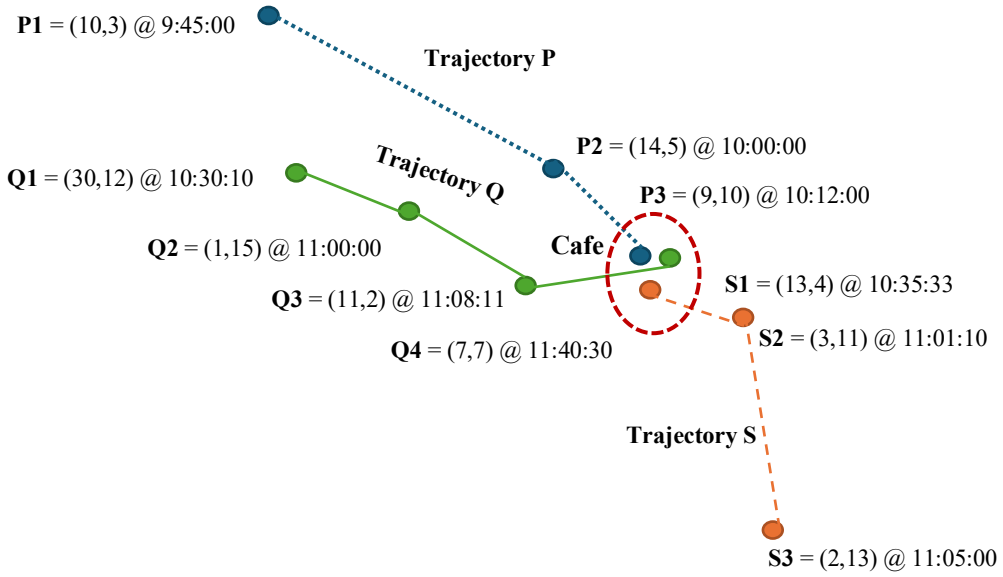


Figure 3.3: Drifting issue of GPS

To address this issue, researchers employ trajectory segmentation techniques to group data points belonging to the same place of interest into a single region. Among the various segmentation approaches, we focus on density-based [29] and grid-based [29] techniques for this comparative study, as we are not targeting a specific application. Using one of these techniques, the sequence points of a trajectory are transformed into a sequence of regions. Frequent sequential patterns are then discovered from these sequences of regions. While trajectory segmentation is a critical step in preparing trajectories for analysis, it can sometimes inhibit the discovery of accurate frequent sequential patterns or even lead to the omission of some patterns.

3.2.2 Trajectory Segmentation Approaches

This section discusses two popular trajectory segmentation techniques, density-based and grid-based, and examines their impacts on discovering trajectory sequential patterns.

3.2.2.1 Density-Based Trajectory Segmentation

Using unsupervised methods, trajectories (or sub-trajectories) are grouped into clusters based on similar features, such as time intervals or locations. In this process, the final number of clusters and the objects that belong to each cluster are initially unknown. Once the clusters are determined, all data points within the trajectories are labeled with the corresponding cluster IDs. Trajectories within the same cluster are expected to be very similar to each other, whereas trajectories from different clusters should exhibit significant differences based on the chosen features.

Various types of clustering algorithms exist, differing in how they define and identify clusters. One such type is density-based algorithms, which define a cluster as a dense region of objects[64]. A threshold for the density of data points within a region is used to determine the relevant data that belong to a single cluster. A notable example of a density-based clustering algorithm is Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [64]. Before using the DBSCAN algorithm, two input parameters need to be initialized: epsilon (ε) and a minimum number of points (MinPts). Epsilon (ε) defines the radius area around a point. DBSCAN randomly selects a point, say A. If other points are located within the radius ε(A) around point A, they are considered neighbors of point A. MinPts is used to determine whether point A is a core point. If the number of points within ε, including the point itself, is greater than or equal to MinPts, point A is labeled as a core point. If the number of points is less than MinPts, but the point lies within the neighborhood of a core point, it is labeled as a border point.

The main challenge lies in determining appropriate values for the two parameters, ε and MinPts, as they significantly influence the rigor and accuracy of the results. Thus, understanding the nature of the data is crucial for DBSCAN to produce meaningful and reliable outcomes.

Using this density-based technique to segment trajectories based on dense areas allows us to group all points that belong to the same location into a single cluster. This process transforms raw trajectories from sequences of points into sequences of cluster IDs, as illustrated in Figure 3.4(a).

Figure 3.4(b) shows three trajectories, each containing a different number of points, as recorded in the second column of Figure 3.4(a). After applying DBSCAN with MinPts = 6 and min_sup = 2, three clusters are identified, each represented by a dotted circle. Subsequently, the sequence of points within the first cluster is replaced with the ID of the first cluster (C1), the second cluster with C2, and the third cluster with C3.

The number of points (i.e., the density of an area) is a crucial parameter for identifying a cluster. If there are insufficient points, some clusters may be omitted, leading to missed trajectory patterns. One challenge when working with trajectory data is the variation in sampling rates between different trajectories. The sampling rate is defined as the time interval between two consecutive points. For example, in Figure 3.4(b), Trajectory 1 has a higher sampling rate than the other two trajectories because its points are collected every minute, while Trajectory 2 has the lowest sampling rate, with points collected every 5 minutes. Another challenge is the speed of the moving object. Suppose Trajectory 1 represents the movement of an elderly person, while Trajectory 2 represents the movement of a young person. Even if the points are collected at the same sampling rate (e.g., every 10 seconds), the number of points recorded may differ. This is

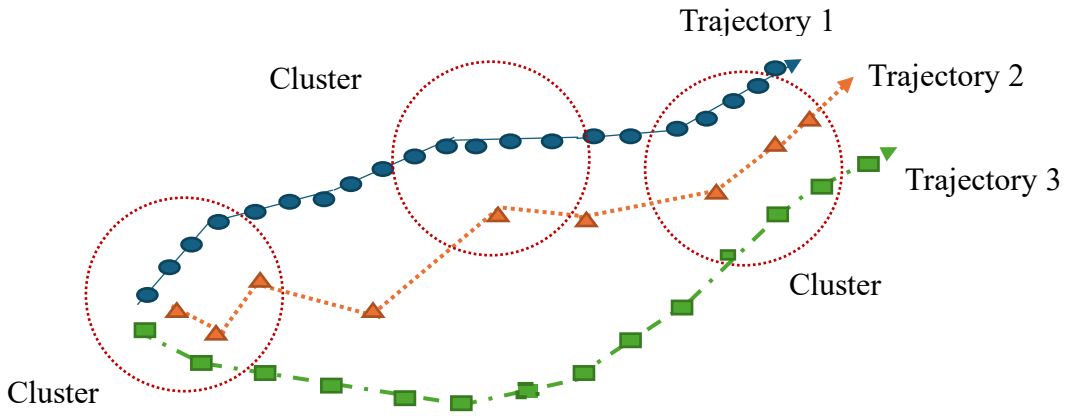
evident in the same circular area, as an elderly person walks more slowly than a young person, resulting in fewer points being collected for the same time interval.

Now, let us assume the user defines MinPts = 8 and min_sup = 2. Due to the sampling rate issue, after applying DBSCAN to the sequential database in Figure 3.4(a), only two clusters are identified: Cluster 1 (C1) and Cluster 3 (C3). Cluster 2 (C2) is missed because it has only six points, which is below the MinPts threshold. This results in the trajectory sequential pattern C1 → C2 being overlooked.

This example illustrates how using a density-based technique like DBSCAN to group trajectories without accounting for factors such as sampling rate differences and movement speed can negatively affect the discovery of trajectory patterns.

Trajectory ID	Sequence of Points	Sequence of Clusters
T1	$P_1, P_2, P_3 \ldots\ldots\ldots\ldots, P_{143}$	C1 ,C2 , C3
T2	$P_1, P_2, P_3 \ldots\ldots\ldots\ldots, P_{186}$	C1 , C2 , C3
T3	$P_1, P_2, P_3 \ldots\ldots\ldots\ldots, P_{276}$	C1 , C3

(a)



(b)

Figure 3.4: Density-based Trajectory Segmentation

3.2.2.2 Grid-Based Trajectory Segmentation

Another approach to grouping points that belong to the same geographical area is by using a regular grid. In this method, the space containing the trajectories is divided into a number of cells based on a user-defined parameter called cell size, where each cell represents a region. Each cell is assigned an ID, as shown in Figure 3.5. Trajectories are then mapped onto the grid, transforming raw trajectories from sequences of points into sequences of cell IDs [29]. However, this straightforward method of grouping points can lead to missing some trajectory patterns. If the points of a trajectory fall into different adjacent cells, some sequential trajectory patterns may be overlooked. For example, as illustrated in Figure 3.5, we have three trajectories:

1. The first row in the table and the black squares in the grid represent Trajectory 1 (T1).
2. The second row in the table and the white squares in the grid represent Trajectory 2 (T2).

3. The third row in the table and the cross symbols in the grid represent Trajectory 3 (T3).

After applying the grid segmentation technique, all points belonging to a specific cell are replaced by the cell's ID (denoted as C), as shown in the second column of the table in Figure 3.5. When mining the database to discover trajectory patterns, and assuming min_sup = 3, the pattern C6 → C8 is missed. This happens because a black square point from Trajectory 1, located within the dotted circle, falls into cell C9 instead of C6. If the cell size had been larger, that black square

Trajectory ID	Sequence of points	Sequence of Cells
T1	P_1, P_2, P_3……., P_{143}	C2 ,C9 , C8 , C4 , C2
T2	P_1, P_2, P_3……., P_{186}	C1 , C2 , C3 , C6 , C8 , C4 , C1
T3	P_1, P_2, P_3……., P_{276}	C2 , C3, C6 , C8 , C7 , C4 , C2

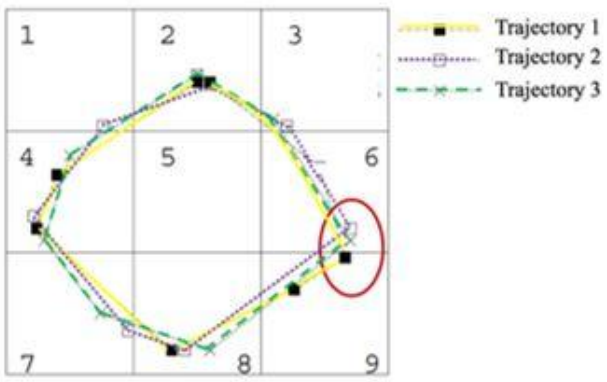


Figure 3.5: Grid-based Trajectory Segmentation

point could have been grouped into C6, which would increase the frequency of the pattern C6 → C8 to meet the min_sup threshold. Consequently, this pattern would have been identified as frequent. This example highlights how the cell size, defined by the user, plays a critical role in discovering frequent sequential trajectory patterns. An inappropriate cell size can lead to missing important patterns.

3.2.3 Discovering Frequent Trajectory Sequential Patterns

We use a well-known algorithm for discovering sequential patterns called SPADE [8] and adopt its implementation from a package in R called cSPADE. As shown in Figure 3.6, the trajectory sequence database contains of tuples of trajectories and each trajectory contains a

Trajectory ID	Sequences
T1	<{A},{B}>
T2	<{A},{C},{D}>
T3	<{D},{B},{D}>
T4	<{D},{B},{E}>
T5	<{A},{B},{C}>

Figure 3.6: Trajectory Sequence Database

sequence of points. For simplicity, we use letters to identify each point (its timestamp, latitude, and longitude); thus, Trajectory 1 is rewritten to be <{A}, {B}>.

SPADE [8] uses the vertical format of a sequential database, where each item (location) is

A		B		C		D		E	
SID	EID	SID	EID	SID	EID	SID	EID	SID	EID
1	1	1	2	2	2	2	3	4	3
2	1	3	2	5	3	3	1,3		
5	1	4	2			4	1		
		5	2						

Figure 3.7: Vertical Representation of the Trajectory Sequence

associated with an ID-list. The ID-list consists of two columns: Sequence ID (Trajectory ID) identifies the sequence in which the item appears and event ID (Timestamp) represents the position of the item within a specific sequence. Figure 3.7 illustrates the vertical database representation of the trajectory sequence database shown in Figure 3.6. Let us consider item D to demonstrate how

its ID-list is built. The algorithm constructs an ID-list containing the following information: Sequence 2: Item D appears at position 3; sequence 3: Item D appears at positions 1 and 3; and sequence 4: Item D appears at position 1.

By scanning the vertical database, the support count of all items can be computed efficiently. For instance, location A appears three times in the database. If the min_sup = 40%, the frequent 1-sequences will be: {A, B, C, D}. To find 2-sequences, the algorithm uses only the ID-lists of the frequent 1-sequences. It joins the ID-lists of two frequent sequences that share the same SID (Sequence ID) if the event ID of the first item occurs before that of the second item (i.e., the timestamp of the first item is earlier than the timestamp of the second item). The result of this join is stored in the ID-list for the new 2-sequence. For example, locations A and B share two common SIDs: 1 and 5. In both sequences, A always appears before B. The join result is shown in Figure 3.8, and the frequent 2-sequences after this step are: <{A},{B}>, <{D},{B}>, <{A},{C}>. Again, from the ID-lists of the 2-sequences, the frequent sequences can be identified and used to generate the frequent 3-sequences. This process is repeated until the algorithm enumerates all frequent sequences.

<{A},{B}>			<{A},{C}>		
SID	EID (A)	EID (B)	SID	EID (A)	EID (C)
1	1	2	2	1	2
5	1	2	5	1	3

Figure 3.8: Id-lists for some 2-sequences

In Chapter 4, we will present our experiments to compare the performance of the density-based and grid-based segmentation techniques, and their impacts on trajectory pattern mining.

3.3 Minits+: An Algorithm for Discovering Timed Sequential Patterns in Static Discretized Timed Sequence Database

In this section, we introduce a technique called Minits+ for MINIng Timed Sequential patterns from a static Discretized Timed Sequence Database.

3.3.1 Motivation of Minits+

While timestamps are used to order events within a sequential pattern, the temporal relation between itemsets is not retained. Previously proposed algorithms aimed to improve the efficiency of methods for discovering frequent sequential patterns but neglected to incorporate temporal information. However, in many applications, it is essential to know the time range required to move from one event to another. Sequential patterns that include temporal information allow us to answer questions such as “In what order do the measurements for train aerodynamic phenomena frequently occur?” and “When will the next measurements of train aerodynamics occur?” As illustrated in Figure 3.9, we have historical weather data (temperature T and wind speed W) recorded by four sensors. Each sensor is denoted as S with a unique ID, as shown in the first column. The time of each measurement is recorded alongside the data, as shown in the second column.

Since sequential pattern mining algorithms are not designed to handle continuous data, we must apply a partitioning technique to segment the data into classes with similar features or those that fall within the same group. To achieve this, an additional column is added next to each column to represent the equivalent class ID. For the timestamp, the first day of data collection is

determined, and the last day of data collection is also determined to specify the time window for data collection. Then, the timestamp is discretized based on a unit such as minutes or hours, depending on the nature of the application. In the example in Figure 1.3, the first hour of data collection is (4/9/13 9:00:00), and the last hour of data collection is (4/11/17 10:45:00). After calculating the hour number for each timestamp, where the first recorded date is considered as hour 1, an additional column was added next to the Timestamp column called Timestamp Class. For wind speed (W) can be categorized into five levels [46], and each level refers to the damage caused: (1) minimal (74 ≤W<95); (2) moderate (96≤W≤110); (3) extensive (111≤W≤129); (4) extreme (130 ≤ W ≤ 156); and (5) catastrophic (W ≥ 157).

Sensor ID	Timestamp	Timestamp Class (hours)	Tempera-ture (T)	Tempera-ture Class (TC)	Wind Speed (W)	Wind Speed Class (WC)
S1	2/10/15 10:50:00	673	54	1	112	3
S2	4/9/13 3:15:00	2	45	1	75	1
S1	3/12/15 11:00:00	703	50	1	100	2
S2	4/9/13 9:00:00	1	41	1	115	3
S2	10/10/15 5:23:00	915	51	1	90	1
S3	5/13/13 2:20:00	35	47	2	125	3
S4	4/11/17 10:45:00	1464	55	1	118	3
S3	7/23/16 4:29:00	1202	50	1	99	2
S4	1/11/17 11:00:00	1374	48	1	127	3

Figure 3.9: Sensors' historical weather information and discretized data

In the first row of the last column, the wind speed is classified as Class 3 because the value 112 falls within the "Extensive" range for Class 3. The timed sequential pattern we aim to discover is a sequence of frequently occurring measurements among sensors along with the typical temporal relation between these measurements (measured in hours in this example). The format of a pattern discovered in this study would look like:

< {TC1, WC3} [30 hours, 36 hours] {WC5}>

The above pattern represents a timed sequential pattern consisting of two itemsets. Items within the curly braces { } appear at the same timestamp, while the square brackets [ ] indicate the temporal relation from the previous itemset to the next. In this pattern, the first itemset {TC1, WC3} indicates that the temperature from Class 1 and the wind speed from Class 3 occur together at the same time. The square bracket [30 hours,36 hours] specifies that, after 30 to 36 hours, the next itemset {WC5} occurs, representing the wind speed from Class 5.

3.3.2 Overview of Minits+

To discover the complete set of timed sequential patterns, the following steps are performed in our algorithm:

1. Read the following from a user: min_sup, Discretized Timed Sequence Database DTSDB, User-required statistics in the timed sequential patterns.
2. Scan DTSDB to determine distinct items.
3. For each distinct Item I_j, compute its support. If I_j is frequent, add it to the set of Frequent Items, denoted as FI and the set of timed sequential patterns TSPs.
4. For each frequent item I_j, build the pseudo-projected database, which is a projection of the original database. This projection collects all pointers referring to the sequences in the main memory as a pseudo-projection. Building projected databases effectively reduces their size, significantly minimizing the effort required for generating candidate subsequences.
5. Compute the occurrences of event-relation and sequence-relation between I_i and each frequent item in FI.
6. Append f_K, an element from FI, to I_j considering the type of relation (event or sequence relation), and add the pattern into TSPs after compute the statistics. If it is sequence relation and the

pattern is frequent, then the user required statistic is computed. The user can select one of the following statistics:

a. Transition time which contains the minimum time required to move from one itemset to next itemset in a timed sequential pattern and maximum time required to move from one itemset to next itemset in a timed sequential pattern.
b. Average time required to move from one itemset to next itemset in a timed sequential pattern.
c. Most frequent time required to move from one itemset to next itemset in a timed sequential pattern.
d. Median time required to move from one itemset to next itemset in a timed sequential pattern.

7. Update FI to include the new frequent items in pseudo-projected database.
8. Repeat steps 2, 3, 4, and 5 until no new timed sequential patterns (TSP) can be discovered.

The pseudo-code of the Minits algorithm is shown in Figure 3.10.

Algorithm 3.1: Minits +

Input: Discretized Timed Sequence Database (DTSDB), minimum support threshold (min_sup), user's required statistic

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

```
for each distinct item
  Compute its support.
  Add frequent item to Frequent Item set (FI);
end for
Add all elements of FI into FP;
Call Find _TSP (DTSDB, Prefix, FI); // Prefix = Ø during the first call
Function Find _TSP (DTSDB, Prefix, FI) {
 // Initialize statistics variables
  min=0, max = 0, avg=0, mod=0, med=0;
  for each item Ij in FI
.   Build Ij- pseudo projection DB from DTSDB;
.   if (Ij- pseudo projection DB is not empty) then
.     Continue
.   else
.     for each sequence s in the Ij- pseudo projection DB
.       for each event ej in a sequence
.         for each item fk in FI
.           if (there is an e-relation between Ij and fk in the event ej) then
.             e-relation support count ++
.           else if (there is a s-relation between Ij in event ej and fk in the event ej+1) then
.             s-relation support count ++
              //Compute following statistics values
.             minimum and maximum
.             average
.             most frequent value
.             intermediate value
.           end if
.         end for
.       end for
.     end for
.   end if
.   Update FI to contain new frequent events nl, where the support of Ij ∪ nl >= min_sup
.    for each values nl in FI
.     if (Ij and nl has an e-relation) then
```

. Prefix = < (I_j ∪ n_l)>
. Add Prefix into FP
. Call *Find _TSP (I_j- pseudo projection DB, Prefix, FI*)
. **else if** (v_i and n_l has a s-relation) **then**
. Prefix = v_i ∪ n_l
. Add < v_i ∪ [user's required statistic] ∪ n_l> into FP
. Call *Find _TSP (I_j- pseudo projection DB, Prefix, FI)*
. **end if**
. **end for**
. **end for**
. }

Figure 3.10: Pseudo-Code of the Minits+ Algorithm

3.3.3 Details of the Minits+

At the beginning, the support of each distinct item I_j is computed by scanning the database. The support is then compared against the minimum support threshold to determine if the item is

Sensor ID	Sequence
S1	< {5, TC1, WC3}, {12, TC1, WC2}>
S2	< {5, TC1, WC1}, {24, TC1, WC3}, {28, TC1, WC1}>
S3	< {6, TC2, WC3}, {19, TC1, WC2}>
S4	< {7, TC1, WC3}, {9, TC1, WC3}>

Figure 3.11: Simplified Discretized Timed Sequence Database

frequent. If it is frequent, I_j is added to FI set. Subsequently, the elements of FI are added to another set called FP (Frequent Patterns), which represents the complete set of timed sequential patterns TSPs discovered by the algorithm, as these items are TSP of length 1.

In the second step, the algorithm divides the search space into several partitions equal to the number of frequent items in FI and constructs a projected database for each I_j. The projected database is defined as a collection of postfixes of sequence in the database S with respect to a given prefix α, denoted as S|α [6]. For example, let prefix α = TC1, and let S be the database shown in

Figure 3.11. The projected database S|TC1 is shown in Figure 3.12. In the projected database, only frequent values are retained. According to the Apriori property [21], any infrequent value can be eliminated, as it will not contribute to generating frequent sequential patterns. Consequently, the algorithm generates candidates based solely on the frequent values detected in the most recent projected database. It calculates support by checking whether the non-empty projected sequences contain the candidates.

In the next step, the algorithm scans the projected database and counts the support of e-relation and s-relation between I_j and each distinct item f_k in FI. The purpose of defining these two types of relationships is to distinguish different connections between items and to identify the associated temporal relations. For example, consider the items TC1 and WC3 from two different data attributes: temperature and wind speed. Two possible links can exist between them. If TC1 and WC3 occur within the same event, they share the same timestamp, which is classified as an e-relation. If TC1 and WC3 occur in different events, they have distinct timestamps, which is classified as an s-relation. For each type of relation, different patterns can be discovered, allowing the algorithm to capture varied temporal connections between items.

Therefore, the algorithm counts the support for each possible pattern while considering the type of relationship. To achieve the second objective, finding the temporal relations, the timestamps must satisfy specific conditions based on the relationship type. For an e-relation, the timestamps of

Sensor ID	Sequence
S1	< {5, WC3}, {12, TC1}>
S2	< {5}, {24, TC1, WC3}, {28, TC1}>
S3	-
S4	< {7, WC3}, {9, TC1, WC3}>

Figure 3.12: Projected database for prefix TC1 - S|TC1

Ij and fk must be the same, resulting in the temporal relations always being zero. For an s-relation, the timestamps of Ij and fk must differ. In this case, the temporal relations are recalculated according to Line 24 of the algorithm, and the interval is updated to reflect the minimum and maximum times observed. At the end of this step, the support of each relation is compared against the min_sup threshold, retaining only the frequent relations. For example, it was found that TC1 occurs 75% of the time with TC1, forming an s-relation. Assuming the user chooses the transition time, it will be calculated as [2], [19] between the events {TC1} and {TC1}.

In its fourth step, the algorithm distinguishes frequent patterns by comparing their support against the min_sup threshold and identifies the type of relationship for each pattern.

As mentioned above, if the s-relation between {TC1} and {TC1} has support ≥75% and the min_sup is 75%, then the pattern <{TC1} [2,19] {TC1}> is identified as a timed sequential pattern TSP. Similarly, if the algorithm finds that TC1 also has an e-relation with WC3, occurring 75% of the time, then < {TC1, WC3}> becomes another TSP. The algorithm then updates FI to include the new frequent distinct values TC1 and WC3 for the prefix TC1, resulting in FI = {TC1, WC3}. In its final step, the algorithm repeats Steps 2, 3, and 4: it builds a projected database for each item in FI, computes the support for both e-relation and s-relation, and identifies new timed sequential patterns.

The goal of this step is to extend shorter patterns into longer patterns that share the same prefix. The algorithm terminates when no new TSP can be discovered, which occurs when the projected database becomes empty.

3.3.4 The Proposed Enhancement

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

3.3.4.1 Using the Pseudo-Projection Technique

The drawback of the previous technique is that building marginally smaller copies of the database for each projection and storing them in main memory until needed becomes prohibitively expensive. To address this issue, a more efficient technique called pseudo-projection was proposed in PrefixSpan paper. Instead of creating and maintaining copies of sequences for each projected database, the pseudo-projection technique keeps the original database in memory and uses a collection of pointers to represent each projected database. Each pseudo-projected database consists of the following of information: a pointer to a sequence in the original database and an offset indicating the starting position of the postfix within the sequence. This approach significantly reduces memory overhead and can improve mining efficiency.

For example, the pseudo-projection for the prefix TC1 contains four pointers, each referring to a sequence. In sequence S1, the offset is 3, indicating that the projection starts from this position. Similarly, the offsets for sequences S2, S3, and S4 are 3, 6, and 3, respectively. Since the PrefixSpan algorithm does not handle timestamps, it must be modified to work with Minits. The modified pseudo-projection includes the following components: a pointer to a sequence in the original database, an offset indicating the starting position of the postfix within the sequence, and the timestamp of the event that contains the prefix. In this approach, each timestamp is retained for calculating temporal relations.

Seq ID Pointer	index
S1	3, 5
S2	3, 5
S3	6, 19
S4	3, 7

Figure 3.13: Pseudo projection database for prefix TC1 - S|TC1

Figure 3.13 illustrates the modified pseudo-projection database for the prefix TC1. For example, the first row points to the first sequence S1 in the original database, where TC1 appears at event 1 and with timestamp 5, and the offset is 3, resulting in the index [3, 5].

We refer to the version of Minits+ that uses the physical projected database as Minits+-projected DB, while the modified version that replaces the physical projected database with the modified pseudo-projection database is simply called Minits+.

3.3.4.2 Using Multi-core CPUs

Another enhancement introduced is the use of multi-core CPUs to implement Minits+, referred to as MMinits+. The workload for extending patterns by recursively calling the function *Find_TSP()*, which in turn calls *Find_TSP( ),* is distributed among multiple threads. A queue of jobs is created to handle all possible candidates, determining their frequency, and these jobs are dynamically assigned to idle threads. By leveraging multiple threads working simultaneously, the execution time of Minits+ is significantly reduced.

Figure 3.14 illustrates the mechanism of MMinits+. When the algorithm first discovers that T1 and W3 are frequent TSP, the two patterns are inserted into the queue and wait for idle threads to process them. The threads work on extending these patterns and generating candidates. After calculating their support, the algorithm identifies <{TC1, TC1}>, <{TC1},{WC3}>, and <{WC3},{TC1}> as new TSP. These three patterns are then inserted into the queue, where additional idle threads are launched simultaneously to detect the new frequent timed patterns.

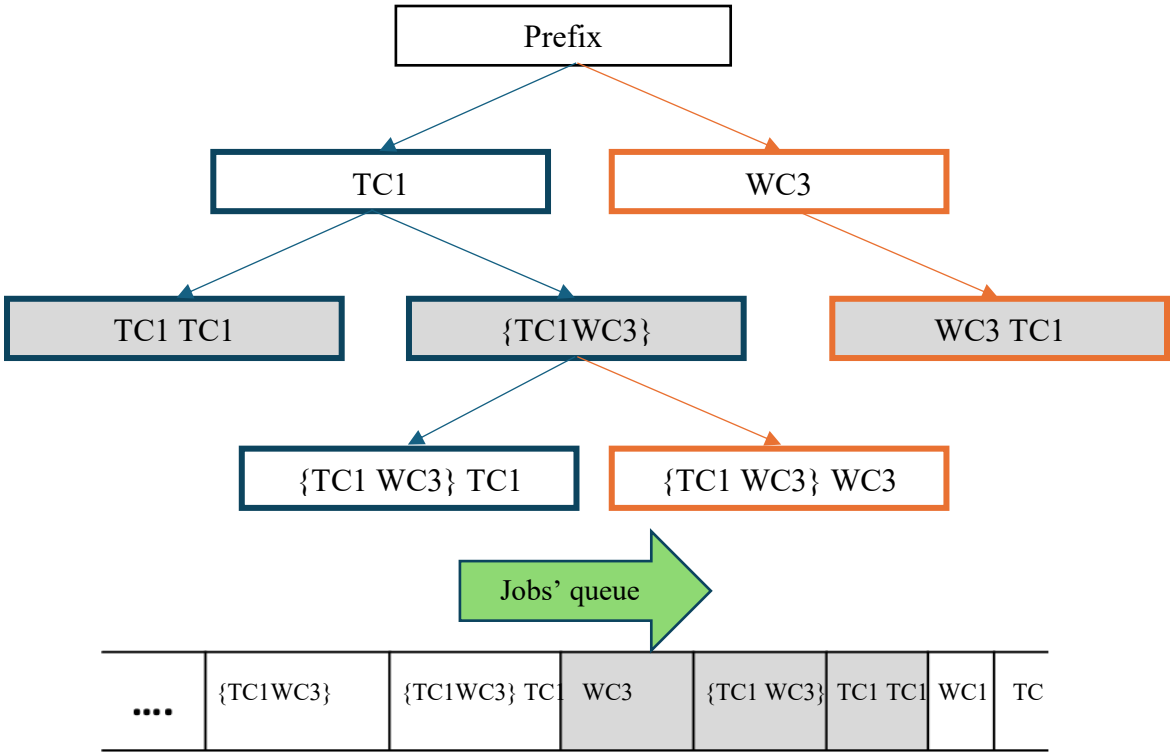


Figure 3.14: The mechanism of multi-core CPUs of Minits+

3.4 The Minits-AllOcc: An Algorithm for Mining Timed Sequential Patterns for All-Occurrences in Static Discretized Timed Sequence Databases

In this section, we present an algorithm for MINIng Timed Sequential patterns for ALL possible OCCurrences of a sequential pattern from static Discretized Timed Sequence Database called Minits-AllOcc.

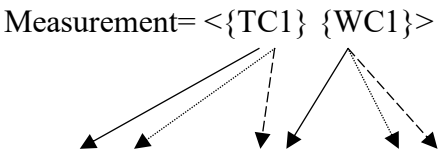
3.4.1 Motivation of Minits-AllOcc

Incorporating temporal information into sequential pattern mining introduces additional challenges. While both approaches used in sequential pattern mining and timed sequential pattern mining must determine whether a pattern occurs in some tuples of a database, timed sequential pattern mining must also identify how many times the pattern occurs in each tuple to compute the temporal relationships between the itemsets within the pattern. For example, consider a tuple in a database that contains all the measurements from a sensor over six months, where the following pattern frequently occurs: low temperature followed by high wind speed after some time. In timed sequential pattern mining, it is essential to determine all occurrences of the high wind speed in that tuple. Simply identifying the first occurrence of this measurement and reporting the temporal relation is insufficient, as the problem requires accounting for all occurrences to provide a complete and accurate temporal analysis.

For example, in Figure 3.15, we observe that the second sensor, S2, has the following measurements based on the timestamp order: Temperature from Class 1 and wind speed from Class 1 {TC1, WC1}, followed by temperature from Class 1 and wind speed from Class 3 {T1C, WC3}, followed by temperature from Class 1 and wind speed from Class 1 {TC1, WC1}. The tuple for this sensor is: S2=< {TC1, WC1}, {TC1, WC3}, {TC1, WC1}>. To find the temporal relation

between the two measurements {TC1} and {WC1} (for the pattern denoted as <{TC1} {WC1}>, the following steps are performed, as shown in Figure 3.15:

1. Find the timestamp difference t1 between the first occurrence of TC1 and the first occurrence of WC1 (solid arrows).
2. Find the timestamp difference t2 between the second occurrence of TC1 and the first occurrence of WC1 (dotted arrows).
3. Find the timestamp difference t3 between the first occurrence of TC1 and the second occurrence of WC1 (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].



S2 = < {TC1, WC1}, {TC1, WC1}, {TC1, WC1}>

Figure 3.15: All Possible Occurrences of the Measurement {TC1}{WC1}in S2

To find all possible occurrences of a pattern, the naïve approach is to scan each tuple in the database until the end. However, sequential pattern mining algorithms stop checking the rest of a tuple as soon as the pattern is found. In contrast, timed sequential pattern mining requires scanning all tuples in the database. First, it is essential to consider all possible occurrences of the pattern, along with the different timestamps of each occurrence, to calculate the temporal relation. Once the temporal relation is determined for one sensor, the same process must be repeated to find the

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

This leads to the second challenge of timed sequential pattern mining: updating 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 containing this pattern is greater than or equal to a user-defined threshold. However, when we extend the pattern to include additional measurements, it does not guarantee that the extended pattern will appear in the same tuples, as some tuples may no longer carry the updated pattern. Consequently, the temporal relation becomes invalid and must be updated based on the new timestamps of the updated tuples. For example, consider the timed sequential pattern < {TC1, WC3}[t1,t2]{TC1}>. From Figure 3.11, we can observe that S1 and S4 contain these measurements. Therefore, t1 and t2 are calculated based on the timestamps associated with these measurements in these tuples. When the pattern is extended to < {TC1, WC3} [t1, t2] {TC1, WC2}>, we observe that the tuple for S4 no longer contains this pattern, and only S1 has the required measurements. As a result, t1 and t2 must be recalculated based on the timestamps associated with the measurements in S1.

A brute-force approach would require rescanning the entire database to update the temporal relation for the extended pattern. Thus, for every pattern extension, the database would need to be scanned multiple times to ensure the correctness of the temporal relations, which significantly increases computational overhead.

We assume that users will consistently select time intervals as the preferred temporal relationship between itemsets in a pattern, given that most applications discussed in Section 1.5.2.2 rely on interval-based representations. Even when restricted to calculating only time intervals, Minits-AllOcc outperforms Minits+, as it effectively addresses the critical challenge outlined in

Section 1.5.2.2: Tracking All Possible Occurrences of a Sequence. Consequently, extending Minits-AllOcc to incorporate additional statistical measures would require only a minor modification.

3.4.2 Overview of Minits-AllOcc

We propose an algorithm called Minits-AllOcc (MINIng Timed Sequential Pattern for All-time Occurrences) to discover the complete set of timed sequential patterns from a Discretized Timed Sequence Database. To achieve this, we introduce a data structure called the occurrence tree (O-tree), which represents all possible occurrences of a pattern in a particular timed sequence within the database (DTSDB). This tree is a key component of the algorithm, as it enables the discovery of timed sequential patterns without repeatedly scanning the Discretized Timed Sequence Database. In the O-tree:

- The timed sequence ID (TSID) is stored as the root.
- Each node stores an event ID (eID) and its timestamp (eID.t).
- A node can have multiple parent nodes and multiple child nodes.
- The link between a parent node and a child node represents the time difference (Δ) between the timestamps of the two nodes (i.e., the parent and its child).

The structure of the O-tree is illustrated in Figure 3.16. For example, when TS3 in Figure 3.17 is scanned, three occurrence trees are created for the items a, b, and d, derived from the timed sequences < {2, a}, {19, a, b},{25,d}>. Since the candidate sequence <{a}>appears twice in TS3, its O-tree in Figure 3.18 has two nodes connected to the root. The first node represents the first occurrence at event e1 with its timestamp, and the second represents the second occurrence at event e2 with its timestamp. However, the sequence <{a} {a}> appears once in TS3 and has two nodes. One node is connected to the root, while the other is connected to the first node via a link

Δ. The link holds the time difference between the parent and child timestamps, calculated as 19−2=17. This structure allows the algorithm to efficiently discover patterns and compute temporal relationships without multiple database scans.

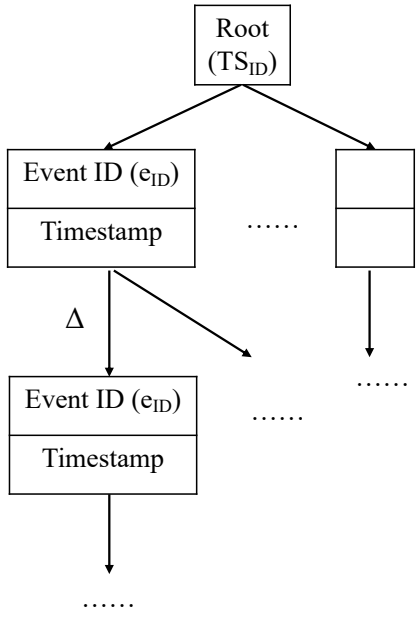


Figure 3.16: Occurrence Tree Data Structure

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}>

Figure 3.17: An Example of Discretized Timed Sequence Database

Since each sequence has an O-tree for each timed sequence in DTSDB that contains it, the sequence will have a collection of O-trees that identify its occurrence in the whole DTSDB. Thus, we give the following definition:

Definition 3.15 (A-Forest): Given a sequence A and Discretized Timed Sequence Database DTSDB, *A-Forest* refers to collection of all O-trees that identify all possible occurrences of sequence A in DTSDB. Figure 3.19 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 DTSDB.

Given a DTSDB and a min_sup threshold, the main goal of Minits-AllOcc is to find the complete set of the timed sequential patterns in the DTSDB, such that each pattern's support is greater than or equal to the min_sup threshold. To achieve this goal, Minits-AllOcc uses the forests to store all the required information from the DTSDB and uses the forests to mine the patterns without having to scan the DTSDB many times. The following steps are performed:

1. Scan DTSDB to build an I_j-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 two relations between itemsets are considered while merging the steps *event-relation* and *sequence-relation.*

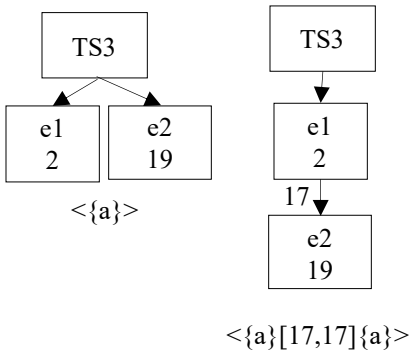


Figure 3.18: An O-tree for the Sequences <{a}> and <{a}{a}> in TS3

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 DTSDB for each candidate to calculate its support.

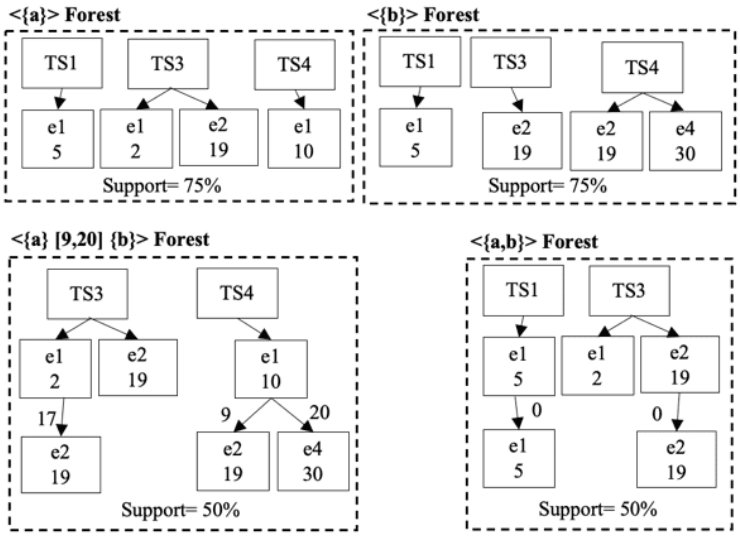


Figure 3.19: Merging O-trees of <{a}> and <{b}> to Generate <{a,b}>-forest and <{a}[9,20]{b}>-forest

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. Minits-All Occ's pseudo-code is presented in Figure 3.20.

3.4.3 Details of the Minits-AllOcc

This section describes the six steps presented in Section 3.4.2 in detail, using the running example shown in Figure 3.17. The algorithm scans the DTSDB tuple by tuple, constructing the associated forest for each item by adding occurrence trees (O-trees) (lines 1–19). For example, as shown in Figure 3.19, after the algorithm finishes scanning the TSDB, the <{a}>-forest contains three O-trees because the sequence <{a}> appears in three timed sequences: TS1, TS3, and TS4. Each O-tree captures all occurrences of an item along with their timestamps within a specific timed sequence.

In TS1, for instance, there is one node in the O-tree, indicating that item a appears in the first event, with a timestamp of 5. To calculate 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 (representing the distinct item's support) is below the threshold, the forest is removed (lines 20–27).

For instance, the sequences <{e}> and <{f}> are not frequent because their forests each contain only one O-tree, indicating that they appear only one timed sequence. As a result, their support is 25%, which is below the threshold.

Algorithm 3.2: Minits-AllOcc

Input: Discretized Timed Sequence Database (DTSDB), minimum support threshold (min_sup)
Output: Timed Sequential Patterns set (TSPs) 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 DTSDB
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 TSPs
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 3.20: Pseudo-Code of the Minits-AllOcc Algorithm

Consequently, two sets are formed: TSPs and 1-TSP. The first set, TSPs, contains the complete timed sequential patterns and is updated periodically as new patterns are discovered. The second set, 1-TSP, contains only the timed sequential patterns of length 1, which will be used as a seed set to extend the patterns in subsequent steps. Both sets, TSPs and 1-TSP, initially contain the values {<{a}, {b}, {d}, {g}>} (lines 24–25). The next step involves generating candidates by merging the O-trees of all 1-timed sequential patterns through the function find-TSP() (line 30). The mechanism for merging trees is as follows: if the relationship is an s-relation, the appended node must have an event ID e_i greater than the event ID in the parent node (i.e., the event IDs of the two nodes are compared) (line 57); then link between the parent and child nodes then holds the time difference between their timestamps (line 59).

In contrast, if the relationship is an e-relation, the appended node must have the same event ID e_i as its parent (line 53). For 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 3.19. 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}>-the 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 with 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. 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 timed sequential patterns, and they are added to the TSPs because their supports are 50% (line 69–76). The support is calculated using the following formula 3.1:

$$Support(candidatesequence) = \frac{O_{trees} \in theforest}{timedsequences \in TSDB * 100} \quad (3.1)$$

These two timed sequential patterns are added into TSPs = {<{a}>,<{b}>,<{d}>,<{g}>,<{a} [9, 20] {b}>, <{a, b}>} The algorithm repeats the same steps, by calling function find-TSP () recursively in line 37, to extend the pattern by merging O-trees,

generating candidates, finding TSP, and computing temporal relations until no more TSP can be found.

As shown in Figure 3.21, pattern <{a} [9, 17] {b} [1, 6] {d}> result from merging <{a} [9, 20] {b}>-forest and <{d}>-forest. The forest consists of only the O-trees 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 TSPs is updated to be {<{a}>, <{b}>, <{d}>, <{g}>, <{a} [9, 20] {b}>, < {a, b}>, <{a} [9, 17] {b} [1, 6] {d}>}. As 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 set 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 }>}.

3.4.4 Proposed Enhancement

In this section, we outline several effective mechanisms designed to enhance the efficiency of Minits-AllOcc.

3.4.4.1 Pruning the Forests

This technique is applied after merging the O-trees. When these O-trees are used in the subsequent step for generating candidates, they will carry only the necessary information. Which means this approach saves space by removing unnecessary nodes and saves time by avoiding traversal of redundant branches in the trees. Any branch in an O-tree that does not have a new appended node is removed after the merging step is executed. Figure 3.22 illustrates the pruning, where deleted branches in the O-trees are marked with a cross symbol. For example, consider the O-tree with a TS3 root, which results from merging the TS3 O-tree from the <{a}>- forest and

<{b}>-forest. Since no appended node exists in the right branch of the <{a}>-forest, that node is removed from the <{a}[9,20] {b}>-forest. These branches are no longer present in the resulting O-trees.

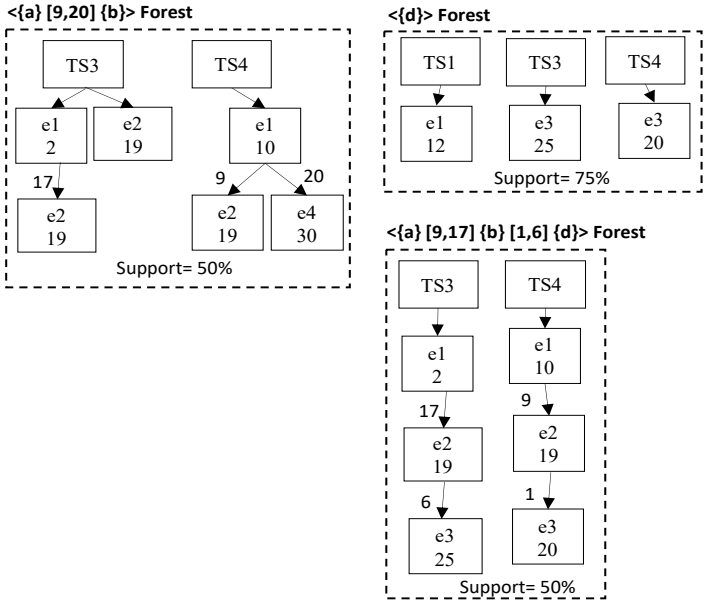


Figure 3.21: Merging <{a} [9, 20] {b}>-forest and <{d}> to Generate <{a}[9, 17]{b}[1, 6] {d}>-forest

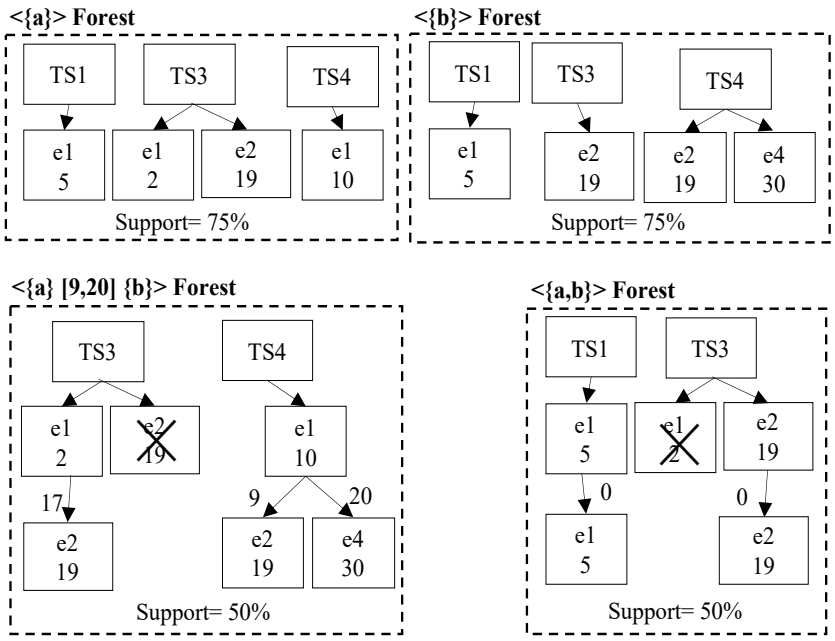


Figure 3.22: Pruning the Original <{a} [9, 20] {b} >-forest and < {a, b} >-forest

3.4.4.2 Using a Frequency Matrix

This technique avoids generating unnecessary candidates, thereby reducing the number of forests. For example, the algorithm uses the 1-sequence forests to generate 2-sequence candidates, retaining only frequent candidates and removing infrequent ones. Since all the required information is already available in the forests, a frequency matrix is built for each sequence to indicate the frequent candidates.

For instance, the frequency matrix for the <{a}> pattern is shown in Figure 3.23. The matrix considers two types of relations (event and sequence, represented as rows) and all 1-timed sequential patterns that can be combined with {a} (represented as columns). The cells under the <{b}> column represent the frequency of the two relations between <{a}> and <{b}>.

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

Figure 3.23: Frequency Matrix for <{a}>

This frequency is calculated from the forests of these patterns, as shown in Figure 3.22. For an s-relation, there are two O-trees (TS3 and TS4) where {a} and {b} occur at different timestamps within the same timed sequence. For an e-relation, there are two O-trees (TS1 and TS3) where {a}and {b} occur at the same timestamp within the same timed sequence.

From the frequency matrix, we can infer that <{g}> is not frequent under either s-relation or e-relation. Thus, there is no need to build forests for the sequences <{a} {g}> or < {a, g}>.

3.4.4.3 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 DTSDB to build the forest for each item and finds that <{a}>, and <{b}> are frequent. In the serial version, the algorithm starts with pattern <{a}> and keeps extending it until no more patterns can be found that have prefix <{a}>. Then, it starts with 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 3.24, and works on generating their candidates at the same time.

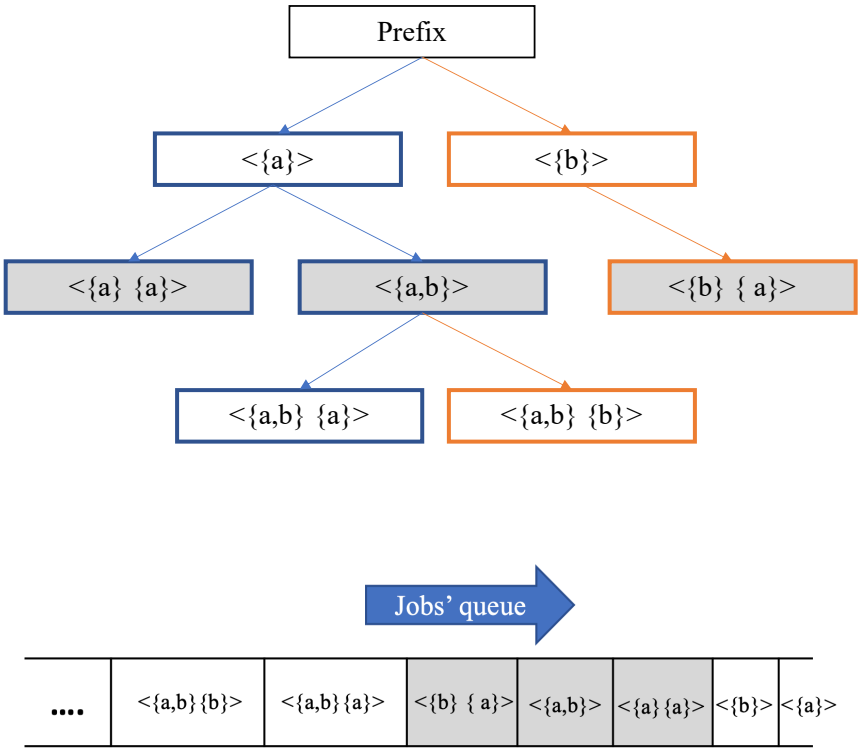


Figure 3.24: Multi-core Implantation

Then, the candidates <{a} {a}>, <{a} {b}>, and so on 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}>, and so on, and then inserting them into the queue. Those candidates wait to be assigned to an idle thread again. This process continues until no more jobs remain in the queue.

3.5 MinitsDays: An Algorithm for Mining Timed Sequential Patterns in Dynamic Discretized Timed Sequence Databases

In this section, we present an algorithm called MinitsDays for MINIng all possible occurrences of Timed Sequential patterns from a DYnamic timed Sequence database.

3.5.1 Motivation of the MinitsDays

Most sequential pattern mining algorithms, including the two previously discussed algorithms: Minits and Minits-AllOcc—assume that databases are static, meaning they do not

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}>

Figure 3.25: Discretized Timed Sequence Database for Patients

change over time. However, in the real world, databases are dynamic and capable of adapting and evolving in response to factors such as data updates, user interactions, or changing requirements. This dynamic nature necessitates the development of an algorithm that can efficiently detect and handle these changes while discovering the complete set of timed sequential patterns without requiring the entire database to be re-mined from scratch. The following are the different scenarios that can occur with a Discretized Timed Sequence Database:

Case 1: New timed sequences are inserted into the Discretized Timed Sequence Database DTSDB. For instance, Figure 3.25 shows the Discretized Timed Sequence Database for four patients. The patients' temperature (TC) and blood pressure (BPC) are measured every time they visit.

After one week, we may have a new patient, so we need to add a new timed sequence into the previous DTSDB, which represents a tuple of a new patient. The new version of timed sequence DB looks like the DB in Figure 3.26**.** If we continue to receive new patients, more timed sequences are added to the existing DTSDB.

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}>

Figure 3.26: Updated Discretized Timed Sequence Database for Patients

Case 2: New events are inserted into an existing timed sequence in the DTSDB. For instance, a company in oilfield services tries to predict the final failure of electric pump sensors by collecting data every day for temperature (TC), pressure (PC), and voltage (VC). Figure 3.27 is the DTSDB for 3 sensors (s).

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}>

Figure 3.27: Discretized Timed Sequence Database for Sensors

After one day we have new measurements for each sensor, so we need to insert each new event into the existing timed sequence in DTSDB. The new version is shown in Figure 3.28. Every day a new version of DTSDB is generated with same number of timed sequences but of different lengths.

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}**>

Figure 3.28: Updated Discretized Timed Sequence Database for Sensors

Case 3: An existing timed sequence is deleted from the Discretized Timed Sequence Database DTSDB. In business intelligence, it is possible to analyze stocks in detail to determine whether each stock is a viable investment. A snapshot of a stock's Discretized Timed Sequence Database is shown in Figure 3.29. The stock price at the opening is known as the open, and the stock price at the closing is known as the close. According to their price, opening and closing stocks are divided into three categories: low (open1 or close1), medium (open2 or close2), and high (open3 or close3).

Company	Timed Sequence
Apple	< {5, Open1}, {12, Close2}, {17, Open1}>
Amazon	<{21, Open1, Close1}, {24, Close3}, {39, Open1}>
Ford	<{2, Open2}, {19, Open1}>
Sears	<{4, Open1}, {9, Open1, Close3}, {13, Open1}>

Figure 3.29: Discretized Timed Sequence Database for Stocks

The fluctuations in the stock price of a company that filed for bankruptcy five years ago, Sears for example, might not significantly impact the current quotes of other equities. Thus, we do not need to keep that information in the DTSDB. Figure 3.30 shows the updated DTSDB.

Company	Timed Sequence
Apple	< {5, Open1}, {12, Close2}, {17, Open1}>
Amazon	<{21, Open1, Close1}, {24, Close3}, {39, Open1}>
Ford	<{2, Open2}, {19, Open1}>

Figure 3.30: Updated Discretized Timed Sequence Database for Stocks

Case 4: The schema of the event in the DTSDB is changed. Let us have the same DTSDB from case 1 in Figure 3.25, which contains the temperature and blood pressure measurements of four patients. The schema of an event in the DTSDB is *(Timestamp, Temperature, Blood pressure).* Suddenly, the doctors decide to observe the effectiveness of one more measurement, cholesterol level. So, the schema of an event in the DTSDB becomes *(Timestamp, Temperature, Blood pressure, Cholesterol level).* In this case there are two possible scenarios:

Case 4.1: Changing the schema of the new coming events. Starting from the current timestamp, the new schema will be used, and all old events are ignored because the doctors are interested in the new relation between the three new symptoms. The updated DTSDB is shown in Figure 3.31.

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}>

Figure 3.31: Updated Discretized 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}**>

Figure 3.32: New Discretized Timed Sequence Database for Case 4.1

We can treat this case as if we have a new DTSDB that contains the data shown in Figure 3.32. Then, we need to execute MinitsDays on this DTSDB to discover the new set of timed sequential patterns among the three symptoms.

After that, if we have a new patient P5, a new timed sequence will be inserted into the previous DTSDB and considered to be **Case 1**.

Case 4.2: Modify the schema of all previous events. We can assume that the data is there, but it was disregarded as the doctors were not interested in knowing the patients' cholesterol levels at the time. Later, doctors decide to study the relationship between TC, BPC, and CC. It means we need to re-read the existing raw data, discretize it, and re-build the timed sequence DB as shown in Figure 3.33. All the previous steps are considered preprocessing steps. The algorithm executes when the new version of the DTSDB is ready. Then, we can treat the DTSDB as a completely new DTSDB and perform the algorithm from the beginning to discover the set of timed sequential patterns that explain the relationship between the new symptoms.

Case 5: An existing event is deleted or modified. In the KDD process, we have a step called preprocessing. So, any missing, erroneous, low-quality, or redundant data is removed or fixed. Thus, we don't need to remove or modify an event after this step. We assume that the result of the preprocessing stage is correct and there is no need to delete or modify an existing event.

Our proposed the algorithm MinitsDays can deal with CASE 1, CASE 2, and CASE 3. In other words, it can discover the timed sequential patterns if the DTSDB has been updated by inserting or deleting a timed sequence without re-mining the entire DTSDB.

We assume that users will consistently select time intervals as the preferred temporal relationship between itemsets in a pattern, given that most applications discussed in Section 1.5.2.2 rely on interval-based representations. Even when restricted to calculating only time intervals, Minits-AllOcc outperforms Minits+, as it effectively addresses the critical challenge outlined in Section 1.5.2.2: Tracking All Possible Occurrences of a Sequence and Section 1.5.3.1: Dealing with Different Types of Database Modifications. Consequently, extending MinitsDays to incorporate additional statistical measures would require only a minor modification.

3.5.2 Overview of MinitsDays

MinitsDays is an algorithm designed to discover the complete, updated set of timed sequential patterns from a dynamic database. The approach leverages historical information from

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}**>

Figure 3.33: Updated Discretized Timed Sequence Database for Case 4.2

previous executions of the algorithm and efficiently updates this information to reflect the changes that have occurred in the database.

In [45], a novel methodology called backward mining was introduced as the opposite of the forward-mining methodology. In forward mining, when a k-pattern is discovered, one item is appended at the end of the pattern to generate a k+1-candidate. In contrast, backward mining appends one item at the beginning of the pattern to generate a k+1-candidate.

Backward mining has the property of stable sequences, which are subsequences whose support counts remain unchanged in the updated database. By identifying and eliminating stable sequences, the support counting process is optimized, resulting in more efficient pattern maintenance. This approach prunes all stable sequences and their super sequences, minimizing the need for database projections and thereby reducing computational overhead while improving the overall efficiency of pattern mining in incremental databases. The correctness of this property was validated in [45]. MinitsDays adopts this property, utilizing two data structures to store and organize all required data from the current DTSDB and traversing these structures to update the set of timed sequential patterns. The 1-sequence data structure contains all distinct 1-items in the Discretized Timed Sequence Database. Each 1-item is associated with its support count and a list

of all timed sequence IDs that contain the item along with their timestamps, as illustrated in Figure 3.34.

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

Figure 3.34: 1-sequence Data Structure for an Item

The 1-sequence data structure for the timed sequence database in Figure 3.35 is shown in Figure 3.36. 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 Figure 3.36.

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}>

Figure 3.35: Dynamic Discretized Timed Sequence Database

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}

Figure 3.36: 1-sequence Data Structure for all Items in Discretized Timed Sequence Database Shown in Figure 3.35

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 min_sup condition. The structure of a node in the TSS-tree is identical to that of the 1-sequence data structure. Each node includes the pattern, its support count, and all occurrences of the pattern in the database. The structure of the TSS-tree is shown in Figure 3.37. MinitsDays takes the following inputs:

- Minimum support threshold (min_sup): A user-defined parameter that specifies the minimum frequency for a pattern to be considered significant.
- 1-Sequence Data Structure (1SD): A data structure that stores all necessary information about distinct itemsets throughout the lifetime of the timed sequence database.
- Previous Timed Sequential Pattern Suffix Tree (Prev_TSS_tree), A tree containing all timed sequential patterns discovered from the previous version of the Discretized Timed Sequence Database.
- List of updated timed sequences TS (Updated_TS_List): A list of timed sequences that have been either inserted into or deleted from the DTSDB.
- Type of update (Update_Type), Specifies whether the update operation is an insertion or deletion of timed sequences in the DTSDB.

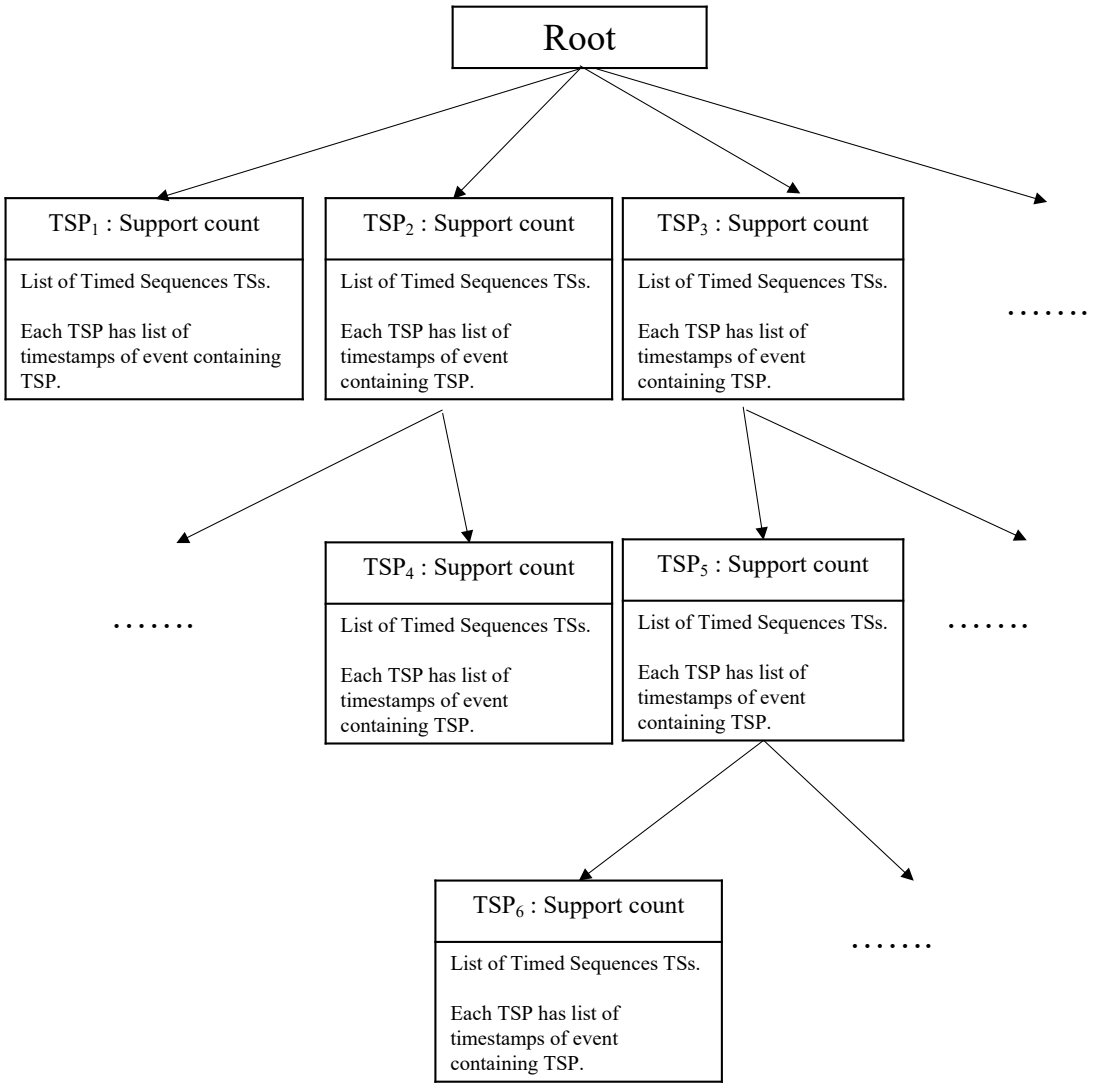


Figure 3.37: TSS-Tree Structure

At the first execution of MinitsDays, the database is treated as a static version. MinitsDays reads the list of updated timed sequences (Updated_TS_List) as the first version of the DTSDB, builds the 1-Sequence and TSS-tree from scratch, and identifies the timed sequential patterns. For all subsequent executions, MinitsDays handles a dynamic DTSDB, leveraging the data already

stored in the 1-Sequence Data Structure (1-SD) and Prev_TSS_tree. The following steps are performed:

1. Determine the type of changes: Identify whether the Update_Type is an insertion or deletion operation.
2. Check the existence of data structures: If the required data structures have not been created, this indicates the first execution of the algorithm, and they must be initialized.
3. Read the list of updated timed sequences (Updated_TS_List):
    - If the operation is an insertion, the *handle_insertion()* function is called.
    - If the operation is a deletion, the *handle_deletion()* function is called.
4. Update the content of the data structures based on the type of changes:
    - If a new distinct item appears due to the addition of new timed sequences, it is added to the 1-Sequence data structure.
    - If an existing distinct item becomes infrequent due to the deletion of some timed sequences, it is removed from the TSS-tree along with all its derived patterns.
5. Traverse the updated TSS-tree: The updated tree is traversed to provide the complete and updated set of timed sequential patterns.

   The pseudo-code for MinitsDays is presented in Figure 3.38.

Algorithm 3.3: MinitsDays

Input: min_sup - 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_sup
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_sup, 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_sup, 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_sup)

50.        return 1SD, New_TSS_tree
```

```
51.  Algorithm Handle_Deletion(min_sup, 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_sup)

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_sup:
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_sup:
86.                 Updated_TSS_tree.remove(pattern)
87.             endif
88.         endfor
89.     endfor
90.     return Updated_TSS_tree

91. Algorithm Get_Descendants(min_sup, 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_sup:
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.           return  Get_Descendants(min_sup,  Updated_TS_List,  New_TSS_tree  ,  1SD,
     New_TSS_tree .root.children(), NTSP)
159.       Endif
```

Figure 3.38: Pseudo-Code of the MinitsDays Algorithm

3.5.3 Details of the MinitsDays

The five steps presented in the previous Section 3.5.2 will be explained in detail using the DTSDB in Figure 3.35. For the first running of the algorithm, 1SD, and Prev_TSS_tree are empty (lines 1, 2). The updated timed sequences (TS) list is currently the first version of the Discretized Timed Sequence Database in Figure 3.35. The algorithm deals with this case as it deals with a static Discretized Timed Sequence Database. The algorithm scans the Updated_TS_List and builds the 1SD, which is shown in Figure 3.36. 1-SD shows all distinct items either they are frequent or not in the DTSDB and its support count, and its occurrence positions (lines 12–27). Then, the support count of each distinct item is compared against the minimum support min_sup and added to NTSP and Timed Sequential Pattern Suffix tree TSS-tree only if the support is greater than or

equal min_sup as shown in Figure 3.39 (lines 28–34). Also, frequent items are added into new set of Timed Sequential Patterns NTSP as 1- timed sequential patterns (lines 28–34). Next, each frequent item is considered as suffix and all 2-candidates are generated by the calling function *Get_Descendants()* in line 35. Let assume the min_sup = %50, the TSS-tree is as shown in Figure 3.39 and NTSP={<{a}>, <{b}>, <{d}>, <{g}>}.

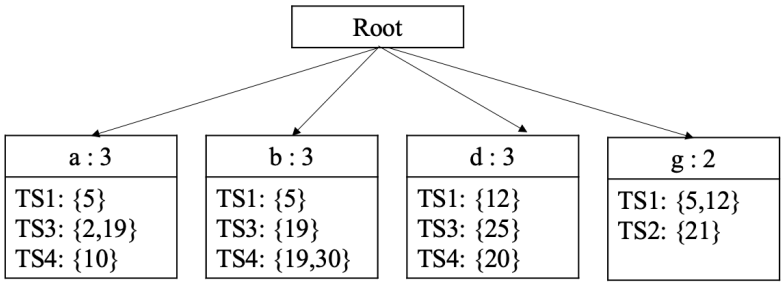


Figure 3.39: TSS-Tree for 1-Timed Sequential Patterns

Now, to decide which of the 2-candidates, which generated by the *Get_Descendants()* function (lines 91–115), is a Timed Sequential Pattern; the algorithm tests these candidates against min_sup (line 116). If it a frequent pattern, it will be added to the NTSP and TSS-tree after the temporal relation is calculated (lines 116–129). For example, after generating all 2-candidates from NTSP, the following patterns are 2-sequential patterns: <{a} {b}>, <{a, b}>, <{b}{d}>, and <{a}{d}>. Next, the temporal relation for sequential patterns that have a sequence relationship is calculated. For instance, the temporal relation for <{a} {b}> is computed as follows according to the DTSDB in Figure 3.35:

- TS3 contains {a} with timestamp 2, and {b} with timestamp 19. The difference between timestamps is 19–2 = 17.

- TS4 contains {a} with timestamp 10, and {b} with timestamp 19. The difference between timestamps is 19–10 = 9.
- TS4 contains also {a} with timestamp 10, and {b} with timestamp 30. The difference between timestamps is 30–10 = 20.
- The minimum is 9 and maximum is 20, 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 Figure 3.40 .

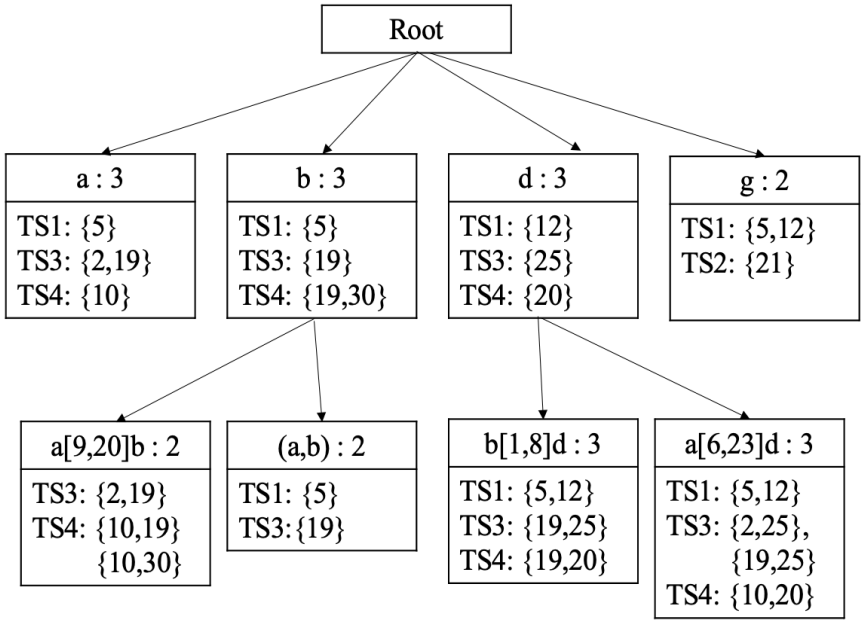


Figure 3.40: Updated TSS-Tree

After that, the algorithm generates 3-candidates, determines the 3-timed sequential patterns, calculates the temporal relation between itemsets in each frequent timed sequential patterns, adds them into the TSS-tree, and excludes infrequent timed sequential patterns.

The algorithm keeps doing the same previous procedure until no more candidates can be generated. When this case happened, the complete set of timed sequential patterns is reported, and 1-SD and TSS-tree are finalized. The final TSS-tree is shown in Figure 3.41 and NTSP = {<{a}>, <{b}>, <{d}>, <{g}>, <{a} [9], [20] {b}>, <{a,b}>, <{a} [1], [8] {d}>, and <{b} [6,23] {d}>}.

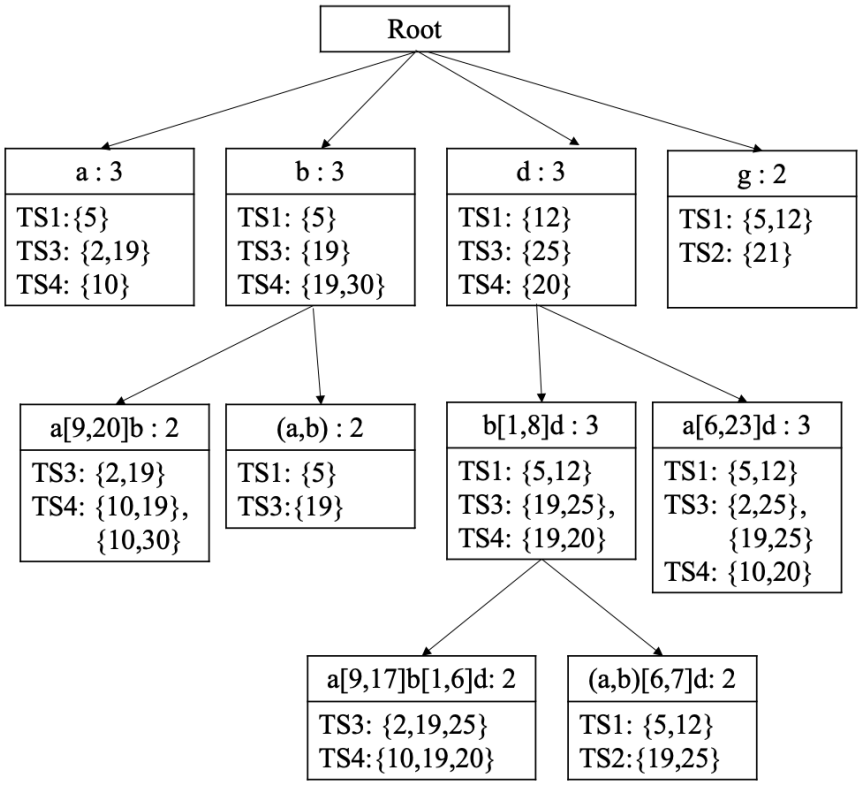


Figure 3.41: Final TSS-Tree

After the first-time execution, the 1SD and TSS-tree were built and no longer empty. Let us assume the DTSDB in Figure 3.35 has been updated by adding more new timed sequences as shown in Figure 3.42. The inputs for the algorithms are:

- Min_sup = 50%
- 1SD as shown in Figure 3.36

- Previous TSS-Tree as shown in Figure 3.41.
- Updated TS_List which contains only the new time sequences that have been added into DTSDB; in this example, updated_TSS_List = {TS5, TS6, TS7}
- Updated_Type = insertion.

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}>

Figure 3.42: UTSDB for Discretized Timed Sequence Database Shown in Figure 3.35

MinitsDays starts reading updated_TSS_List and Updated_ type and update 1SD data accordingly. For example, since the type of updating operation is insertion, the function *Handle*_Insertion( ) is called (line 37).

a: 3	b: 5	d: 3	e: 1	f: 5	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}				

Figure 3.43: 1-Sequence Data Structure for Discretized Timed Sequence Database Shown in Figure 3.42

All the added timed sequences do not contain item *a*, thus, the information of it in 1SD does not change. However, item b exists in the list, so its information is updated as shown in Figure 3.43.

Next, the TSS_tree must also 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, it will be removed from the TSS-Tree and it is all children because all have become infrequent according to the Apriori principle. For example, *g* is no longer frequent because the new support count is 3. Therefore, item *g* is removed from the TSS_tree as shown in Figure 3.44.

Another scenario is if the support count does not change for an item; then, its backward extension is stable and does not change according to the backward mining outlined in[45]. For example, support count for items *a* and *d* do not change, thus all their children remain the same as shown in Figure 3.44.

However, if the support count of an item is changed, the content of that item in the TSS_tree and all its extension will be updated. For example, the support count of item *b* is changed from 3 to 5. Accordingly, all the backward extensions of item *b* will be re-checked, and new candidates will be generated by the calling function *Get_Descendants ()* in line 35. MinitsDays compares the support count of each candidate against min_sup. If a candidate is frequent, the same steps of calculating the temporal relation for sequential pattern <{a} {b}>, as mentioned above, are followed to find the time intervals for each candidate. For example, previous timed sequential patterns of suffix *b* are not frequent because their support count is 2; this does not change even after the DTSDB has new timed sequences. Also, the support count for new candidates does not satisfy the min_sup condition. Thus, no backward extensions are appended to the TSS_tree and current extensions are removed, as shown in Figure 3.44.

The last scenario is when an infrequent distinct item becomes frequent. This leads us to add this item into the TSS_tree and generate all 1-candidates, 2- 1-candidates, and so on until no more candidates can be produced. In our example, item *f* was infrequent because its support count is 1. After adding the three new timed sequences, its support count is 4. MinitsDays updates the TSS_tree by the calling function *Update_TSS_tree ()* in line 49, so item *f* and its discovered timed sequential patterns are inserted into the TSS_tree as shown in Figure 3.44. When no more candidates can be generated, MinitsDays provides the set of new timed sequential patterns NTSP, new TSS_tree, and new 1SD. In this example, the outputs are NTSP = {<{a}>, <{b}>, <{d}>, <{f}>, <{a} [1, 8] {d}>, <{b} [6,23] {d}>, <{b} [6, 23] {d}>}. The new 1-SD is shown in Figure 3.44 and new TSS_tree is shown in Figure 3.44.

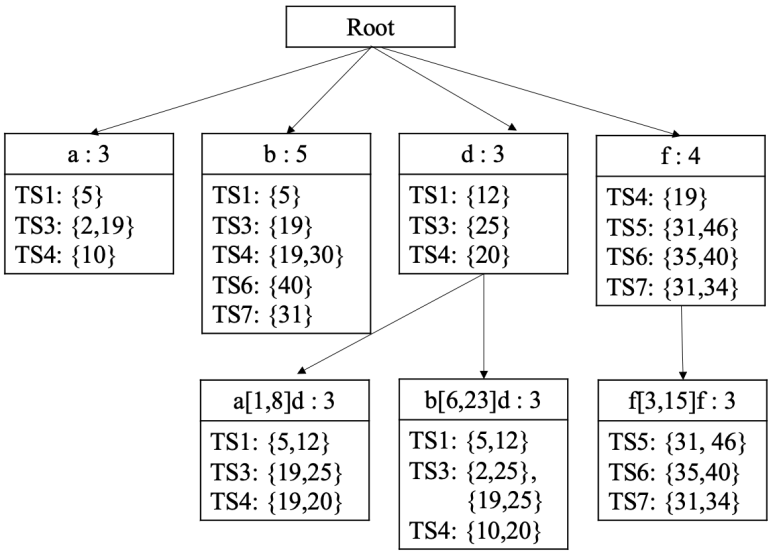


Figure 3.44: Final TSS-Tree

3.6 Summary

In this chapter, we presented our efforts in studying the impacts of two trajectory segmentation techniques: density-based clustering and the grid-based technique—as preprocessing steps for discovering sequential patterns from trajectories. Additionally, we introduced the concept of incorporating temporal relationships between itemsets in frequent sequential patterns.

We also introduced the Minits+ algorithm, which was inspired by PrefixSpan. Since the physical projected database has a very high cost, Minits+ used a technique that replaced that projected database with the modified pseudo projection that saves a lot of the previous cost.

We then presented Minits-AllOcc, an algorithm designed to discover timed sequential patterns TSP. These patterns capture not only sequential relationships but also the transition times between all possible occurrences of events across the Discretized Timed Sequence Database (DTSDB). Both Minits and Minits-AllOcc operate under the assumption that the database is static. However, in real-world scenarios, databases are dynamic, evolving in response to user interactions, updated data, and changing requirements. To address this, we developed MinitsDays, an algorithm capable of recognizing and handling these changes. Without the need to re-mine the entire database, MinitsDays efficiently updates and discovers the complete set of timed sequential patterns when modifications are made to the database.

CHAPTER 4
PERFORMANCE ANALYSIS

In this chapter, we present our analysis of the worst-case computational complexity and provide proof of correctness of the proposed algorithms. Additionally, we include extensive experimental studies to evaluate their performance compared to state-of-the-art algorithms.

4.1 Theoretical Analysis of the Proposed Algorithms

In this section, we analyze the time complexity and provide proof of correctness of the discovered timed sequential patterns for each of the three proposed algorithms: Minits+, Minits-AllOcc, and MinitsDays.

4.1.1 Time Complexity Analysis and proof of correctness of Minits+

4.1.1.1 Time Complexity Analysis

Figure 3.10. in Chapter 3 shows the pseudocode of Algorithm 3.1: Minits+. The time complexity variables for Minits+ are listed as follows:

- N: Number of timed sequences in the discretized timed sequence database (DTSDB).
- M: Number of distinct individual items across all sequences in DTSDB
- L: Maximum length of the timed sequence in the database.

We analyze the time complexity by evaluating the time cost of the major steps for the algorithm and adding them together as the result.

The algorithm begins by scanning the database and computing support, where the algorithm identifies the support of each distinct item across the dataset (Lines 1-4). This involves iterating over M unique items and scanning N sequences, each containing up to L events, leading

to a time complexity of O(M*N*L). This step ensures that only frequent items are considered in subsequent computations, thereby reducing the search space.

Following the identification of frequent items, the algorithm proceeded to build pseudo-projected databases. For each frequent item (Line 5), a pseudo-projection is constructed by maintaining pointers to relevant positions in the original sequences (Lines 6-12). This method reduces memory overhead while still necessitating a scan over the database to locate occurrences of the item. For each of the M frequent items, N sequences of maximum length L are traversed, yielding a time complexity of O(M*N*L).

With the pseudo-projected databases established, the next stage involves computing event-relations (e-relations) and sequence-relations (s-relations). These relational computations (Lines 12-29) analyze how frequent items co-occur and follow each other within sequences. For each of the M frequent items, their interactions with all other frequent items are examined, involving a scan through the projected databases of length up to L, resulting in a time complexity of $O(M^2*N*L)$.

To further enrich the understanding of sequence dynamics, the algorithm then computes temporal statistics for sequence-relations (s-relations) (Lines 21-24). These statistics, such as minimum, maximum, average, median, and mode, are computed during the detection of s-relations. While most of these statistics are updated incrementally with no additional asymptotic cost, computing the median requires sorting and incurs a time complexity of O (l log l), where l denotes the maximum length of the timed sequential patterns.

The algorithm subsequently enters the pattern extension (Line 30) and recursion phase (Line 35 and Line 39), wherein it recursively expands existing patterns by appending new frequent items. Each recursive call involves scanning the database and computing support for extended

patterns through the invocation of the *Find_TSP()* procedure (Lines 7–42) . This recursive process continues until no further frequent patterns can be identified (Line 31). Each call has a time complexity of O(M*N*L), and the number of such calls depends on the depth L and the number of frequent items M.

Finally, the total number of recursive calls is a critical factor in the overall complexity. The recursive nature of pattern extension leads to a combinatorial explosion of possibilities, resulting in up to M^L recursive calls. Consequently, the time complexity of this stage becomes $O(M^L)$, dominating the computational cost for larger values of L.

By aggregating the computational costs of all steps, the overall time complexity of the Minits+ Algorithm is given by: $O((M*N*L) +(M*N*L) + (M^2*N*L) + (l \log l) + (M*N*L) *M^L))$. As the last term dominates for $L \geq 2$, the final asymptotic time complexity of the Minits+ algorithm is: $O(M^L*N*L)$.

4.1.1.2 Proof of Correctness

We prove the correctness, completeness and termination of the Minits+ algorithm.

Lemma 4.1: Every output pattern P is a valid TSP with support ≥ min_sup, and accurate temporal statistics.

Proof: We prove the correctness of the Minits+ algorithm that it outputs only valid and frequent timed sequential patterns (TSPs) using mathematical induction on the pattern length k.

Step 1: Base Case (k=1)

Let D denote the input database consisting of N sequences. In the initial step, Minits+ performs a scan over D to compute the support of each distinct item $x \in X$, where X is the set of all unique items in the database. The support of item x is defined as: $sup(x) = |\{S \in D \mid x \in S\}|$.

Only those items satisfying the minimum support threshold, i.e., $sup(x) \geq min\_sup$, are retained. Thus, all 1-length patterns $P=\langle x \rangle$ included in the output are frequent and valid by definition.

Step 2: Inductive Hypothesis

Assume all patterns of length k, denoted $Pk = \langle x_1,x_2,\ldots,x_k \rangle$, discovered and output by the algorithm are correct i.e., they are frequent and conform to valid temporal constraints

Step 3: Inductive Step (k→k+1)

We must show that all patterns of length k+1, formed by extending Pk, are also valid and frequent.

For each frequent prefix $\alpha = \langle x_1,x_2,\ldots,x_k \rangle$, the algorithm constructs a projected database $D\alpha \subseteq D$ consisting of all suffixes of sequences in which α occurs. The algorithm then attempts to extend α by appending a frequent item $x_j \in X$, such that the resulting sequence $\alpha' = \alpha \bullet x_j$ satisfies one of the following two relations:

1. Event-relation (e-relation): $x_j \in e_t$ where $e_t = e_\alpha$.

   This indicates that x_j appears in the same event (i.e., having the same timestamp) as the last item in α, and the support is verified over Dα.

2. Sequence-relation (s-relation): $x_j \in e_{t'}$ where $t'>t$, and $\Delta t = timestamp(x_j) - timestamp(x_k)$.

   This indicates that x_j appears in a subsequent event, and the time difference Δt is recorded as the transition time between the two items.

The extended pattern $\alpha'=\langle x_1,\ldots,x_k,x_j \rangle$ is retained if and only if $sup(\alpha') \geq min\_sup$.

By the inductive hypothesis, α is valid. Since the extension is made only when α′ is frequent and satisfies a valid temporal relation (e or s), it follows that α′ is also valid. Hence, all patterns of length k+1 produced by Minits+ are correct where the temporal relation is where the temporal relation is one of the statistics selected by the user: minimum, maximum, average, most frequent

value, or median. For each s-relation, the algorithm computes precise transition times based on exact timestamps. Let Δt_1, Δt_2,…,Δt_n be the set of time intervals between items in each valid instance of the s-relation. The algorithm aggregates temporal statistics over this multiset: min(Δt), max(Δt), avg(Δt), median(Δt), mode(Δt). These are computed during the traversal of the projected databases and maintained incrementally, except for the median which requires sorting. Since all-time differences are computed from valid occurrences of s-relations in the dataset, the temporal statistics are guaranteed to be accurate.

Lemma 4.2: No frequent TSP is omitted from the output.

Proof: We now prove that the Minits+ algorithm is complete, i.e., it discovers all valid timed sequential patterns (TSPs) that satisfy the minimum support threshold. We proceed by proof by contradiction.

Assume, for the sake of contradiction, that there exists a frequent TSP P′ of length L that is omitted from the output of Minits+. Let us denote this pattern as: $P'=\alpha \bullet x_j$, where α is the prefix of P′ consisting of the first L−1 items, centered dot (•) is a concatenation operation, and x_j is the final item. Since P′ is assumed to be frequent, we have: $sup(P') \geq min\_sup$.

By the Apriori property of sequential pattern mining, every subsequence of a frequent pattern must also be frequent. Therefore, α is frequent: $sup(\alpha) \geq min\_sup$.

According to the bottom-up approach of the Minits+ algorithm to generate patterns, all frequent patterns of length k<L, including α, are discovered before attempting any extension. Thus, the algorithm must have built the projected database Dα, consisting of all suffixes of sequences that contain α. Given that $sup(P') \geq min\_sup$, the extension $\alpha \bullet x_j$ would have been considered, and the algorithm would have verified its frequency.

If $\sup(\alpha \bullet x_j) \geq \min\_sup$, the algorithm would retain P′ in the output set, since all such frequent extensions are explicitly tracked and included. Thus, the assumption that P′ was omitted contradicts the design of the Minits+ algorithm.

Therefore, the initial assumption is false, and we conclude that no frequent timed sequential pattern is omitted from the output of Minits+.

Lemma 4.3: Minits+ terminates after finite steps.

Proof: The termination of the Minits+ algorithm can be proven by a direct argument based on the bounded nature of the recursion and the finiteness of the branching factor. Let M be the number of distinct frequent items in the database and L be the maximum allowable pattern length. First, we observe that the maximum recursion depth of the algorithm is bounded by L, since pattern extensions are only performed up to length L. That is, the recursive procedure *Find_TSP(α)* can extend a prefix pattern α of length k only if k<L. Hence, the recursion depth is upper bounded by L, i.e., $\text{Depth}_{max} \leq L$.

Second, at each recursive call, the algorithm considers at most M possible extensions by appending one of the M frequent items to the current prefix, subject to temporal and support constraints. Therefore, the branching factor per call is at most M, and the total number of recursive branches across the entire search space is finite. Specifically, the maximum number of recursive calls is bounded above by the following formula 4.1:

$$\sum_{k=1}^{L} M^k = \frac{M^{L+1}-M}{M-1} \quad (4.1)$$

Since both the recursion depth and the number of extensions at each step are finite and bounded, the search space explored by the algorithm is finite. Therefore, Minits+ is guaranteed to terminate after a finite number of steps.

Theorem 4.1

Minits+ discovers all and only frequent timed sequential patterns (TSPs) from a discretized timed sequence database D with support ≥ min_sup, while correctly computing transition time statistics.

Proof: By Lemma 4.1, Lemma 4.2, and Lemma 4.3, Minits+ discovers a correct and complete set of timed sequential patterns and successfully terminates.

4.1.2 Time Complexity and proof of correctness of Minits-AllOcc

4.1.2.1 Time Complexity Analysis

Figure 3.20 in Chapter 3 shows the pseudocode of Algorithm 3.2: Minits-AllOcc. The variables of the time complexity analysis of Minits-AllOcc are listed as follows:

- N: Number of timed sequences in DTSDB
- M: Number of distinct individual items across all sequences in DTSDB
- L: Maximum events per sequence (or maximum sequence length)

The first step in the algorithm involves building initial forests (referred to as Ij-forests) by scanning the entire DTSDB to create O-trees for all 1-length patterns (Lines 1- 19). This step requires examining every item in every event of every sequence (Line 2). Given that there are N sequences, each with up to L events (Line 4), and each event can contain up to M items, the time complexity of this initial construction phase is O(N*L*M). Following the construction of the initial forests, the algorithm proceeds to eliminate infrequent 1-length patterns that do not meet the user-defined minimum support threshold (Lines 20–27). Since this operation simply requires checking the support of each of the M distinct items, the filtering process incurs a time complexity of O(M).

The next phase of the algorithm is concerned with pattern extension, where candidate patterns are generated by combining previously discovered frequent patterns. This process begins

with the generation of all pairs of frequent 1-length patterns (Lines 28-29). If the number of frequent 1-length patterns is less than or equal to M, then the total number of such pairs is upper bounded by M^2, leading to a complexity of $O(M^2)$ for this step. Pattern extension is carried out recursively using the *find_TSP()* function (Line 30), which attempts to grow each pattern up to the maximum allowable depth, corresponding to the number of itemsets in the longest potential pattern (Lines 33–42). This recursive behavior leads to a complexity proportional to the recursion depth, yielding a time complexity of O(L) per recursive call. A significant portion of the computational load arises from the Merge-Trees () function (Lines 43–48). To determine the support (Lines 47–48) and temporal relationships (Lines 55–59) of candidate patterns, this function merges the O-trees of prefix and suffix patterns. Since each pattern can appear in up to N sequences, its forest contains up to N O-trees. Each O-tree contains up to L nodes. The merging process requires comparing all pairs of O-trees from the prefix and suffix forests, and for each such pair, all combinations of nodes are evaluated to establish valid temporal relationships (Lines 70–75). This results in a worst-case complexity of $O(N^2*L^2)$ for each merge operation. Considering the combinatorial growth of candidate patterns, up to M^L in the worst case, the total time complexity of the pattern extension step becomes $O(M^L*N^2*L^2)$. When combined with the earlier phases of initial forest construction and pruning, the overall worst-case time complexity of the Minits-AllOcc algorithm is expressed as: $O((N*L*M)) + O(M) + O(M^L*N^2*L^2))$. Among these, the dominant term is $O(M^L*N^2*L^2)$, which reflects the core computational burden of the algorithm.

4.1.2.2 Proof of Correctness

We prove the correctness, completeness and termination of the Minits-AllOcc algorithm.

Lemma 4.4: Every output timed sequence pattern TSP produced by Minits-AllOcc is valid with support ≥ min_sup, and all temporal intervals in TSP are calculated from all valid occurrences of TSP in DTSDB.

Proof: We refer to the Line numbers in Figure 3.20 in Chapter 3 that shows the pseudocode of algorithm 3.2: Minits-AllOcc. To formally prove the correctness of the Minits-AllOcc algorithm, we use mathematical induction on the length k of timed sequential patterns TSP=⟨I_1 [τ_1] I_2 [τ_2]… [τ_{k-1}]I_k⟩, where each I_j is an itemset. The goal is to show that the algorithm outputs only patterns with correct support ≥ min_sup and the associated temporal intervals for every pattern of length k, provided that all patterns of length k-1 are processed correctly.

Step 1: Base Case (k = 1)

In the base case, we consider all patterns of 1-length ⟨x⟩, where x ∈ M (the set of all distinct items). The algorithm begins by scanning the entire database (Lines 1–19) and constructs an occurrence tree (O-tree) for each item x, capturing all occurrences of x across all sequences in the database. By construction, every instance of item x in each sequence is recorded, and thus the support of ⟨x⟩, denoted support(⟨x⟩), is computed exactly. Subsequently, the algorithm prunes all infrequent items (Lines 20–27), retaining only those patterns for which support (⟨x⟩) ≥ min_sup. As a result, all patterns in the set of TSP, satisfy the support condition. Since all patterns of length 1 contain only a single itemset, there are no inter-itemset temporal intervals to compute, and therefore the temporal relation is correct. Hence, the base case holds.

Step 2: Inductive Hypothesis

Assume for all patterns of length k-1; that is, for every such pattern P′= P=⟨I_1 [τ_1] I_2 [τ_2]… [τ_{k-2}]I_{k-1}⟩, the algorithm has correctly:

1. Constructed O-trees that capture all valid occurrences of P′ across all sequences in the database.

2. Computed the correct support(P′).

3. Derived the exact temporal interval between two consecutive itemsets in P′.

Step 3: Inductive Step (k > 1)

We must now show that the algorithm correctly constructs the pattern $P = P' \bullet I_k = \langle I_1 [\tau_1] I_2 [\tau_2] \ldots [\tau_{k-1}] I_k \rangle$,

Part 1: Correct Support Computation for P

By the inductive hypothesis, the O-trees for P′ contain all valid occurrences in each sequence S_i. The algorithm extends P′ by appending I_k, checking whether a valid extension exists in the same sequence. For each O-tree of P′ in S_i, it evaluates all occurrences of I_k that satisfy the temporal constraint: $t_{jk} > t_{jk-1}$.

This extension process is implemented in Lines 53–61 of the algorithm. If such a temporal relationship is found, the algorithm appends I_k to the existing occurrence in the O-tree and calculates the time difference $t_{jk} > t_{jk-1}$. The support of pattern P is then computed as the number of distinct sequences in which such valid extensions exist. Since all occurrences of P′ are already captured, and all valid positions for I_k following I_{k-1} are considered, the support of P is calculated exhaustively and accurately. The pattern P is output only if support(P) ≥ min_sup ensuring that only frequent patterns are retained.

Part 2: Exact Temporal Interval Computation for P

For each valid occurrence of pattern P in sequence S_i, denoted $(j_1, j_2, \ldots, j_k)$ the algorithm computes the temporal difference between the last two itemsets as $\Delta = t_{jk} - t_{jk-1}$. This value is stored in the corresponding O-tree node during the merge process (Lines 55 and 59). After collecting all such Δ values across all O-trees of pattern P, the algorithm aggregates them to compute the final interval: [min Δ, max Δ]. This computation occurs in Lines 71–74. Since all valid occurrences of

P are derived from complete and correct O-trees of P′ (by the inductive assumption), and every eligible I_k that follows I_{k-1} temporally is evaluated, no valid occurrence of P is missed. Thus, the temporal relation assigned to P is mathematically complete and correct.

Lemma 4.5: No valid pattern is missed.

Proof: To demonstrate the completeness of the Minits-AllOcc algorithm, we employ a proof by contradiction. Specifically, we aim to show that the algorithm does not miss any valid pattern that satisfies the user-defined minimum support threshold min_sup. We assume the contrary that there exists at least one valid timed sequential pattern that is not included in the final output set.

Let us assume that there exists a pattern $\sigma*=\langle I_1\ [\tau_1]\ I_2\ [\tau_2]\ldots\ [\tau_{k-1}]I_k\rangle$, such that:

- $\sigma*$ is a valid pattern (i.e., support($\sigma*$) ≥ min_sup), and
- $\sigma*\notin$TSP-set, where TSP-set denotes the complete set of patterns output by the algorithm.

We now analyze this assumption by cases based on the length k of $\sigma*$.

Case 1: k = 1. In this case, $\sigma*$ is a 1-length pattern consisting of a single itemset $\langle I_1\rangle$. According to the algorithm, all sequences in the database are scanned (Lines 1–19), and occurrence trees (O-trees) are built for all distinct items. Then, in Lines 20–27, the algorithm checks whether the support of each 1-length pattern is greater than or equal to min_sup. If it is, the pattern is added to the set TSP. Since $\sigma*$ is a valid 1-length pattern by assumption, it must satisfy the condition support($\sigma*$) ≥ min_sup. Therefore, it should have been retained during the filtering step. If $\sigma*\notin$TSP, this would imply that the algorithm incorrectly pruned a frequent 1-length pattern, which contradicts the logic and correctness of the pruning step in Lines 20–27. Hence, our assumption leads to a contradiction.

Case 2: k > 1. Now consider the case where $\sigma*$ is a pattern of length k > 1. Let $P^*=\langle I_1\ [\tau_1]\ I_2\ [\tau_2]\ldots\ [\tau_{k-2}]I_{k-1}\rangle$, be the prefix of $\sigma*$. Since $\sigma*$ is valid, its prefix P^* must also be valid by definition (i.e., a subsequence of a valid pattern must itself be valid). By inductive reasoning, $P^*\in$TSP, since if it were missing, it would contradict Case 1.

The algorithm attempts to extend P^* by combining it with any frequent pattern of length 1 $\langle x_k\rangle\in$TSP using the Merge-Trees () function (Lines 43–82). This merge operation is initiated in Lines 28–42 through the recursive Find-TSP () function, which considers all combinations of valid prefixes and suffixes. Since:

- $\langle x_k\rangle$ is a frequent 1-length pattern (otherwise $\sigma*$ would be invalid),
- $\sigma*$ occurs in the database (by assumption), and
- The O-trees of P^* and $\langle x_k\rangle$ share at least one common sequence ID (TSID),

the Merge-Trees () function should have successfully combined them and computed the support and temporal intervals for $\sigma*$.

If the support of $\sigma*$ meets or exceeds min_sup, then the algorithm must include it in the TSP-set. If $\sigma*\notin$TSP-set, this implies that the merge process either failed to process a valid extension or incorrectly computed the support both of which contradict the correctness of Lines 43–82.

Lemma 4.6: Minits-AllOcc terminates after finite steps.

Proof: To prove that the Minits-AllOcc algorithm terminates, we provide a direct proof based on the bounded nature of its search space and iteration depth. The algorithm operates on a static Discretized Timed Sequence Database (DTSDB), which contains a finite number of timed sequences. Let L denote the maximum number of events in any sequence TS∈DTSDB. Then, the

maximum possible length of any pattern is also bounded above by L, as no pattern can contain more itemsets than the longest sequence in the database. Formally, we state:

$$Maximum\ pattern\ length \leq max_{TS \in DTSDB} \mid TS \mid = L \quad (4.2)$$

Since L is finite (because the database is finite), the number of pattern extensions is likewise bounded in depth. Furthermore, at each level k, the algorithm attempts to generate extensions of frequent patterns by joining them with frequent patterns of length 1. Let F_1 denote the set of frequent patterns of length 1, such that $|F_1| < \infty$. During each recursive call or iteration, each pattern can be extended in at most $|F_1|$ ways (i.e., by appending each item in F_1). Therefore, the total number of new candidate patterns generated at each level is finite and upper-bounded by $|F_1|$.

The recursive pattern extension process thus resembles a finite tree, where each node has at most $|F_1|$ children and the depth does not exceed L. As both the breadth and depth of this extension tree are finite, the total number of possible candidates the algorithm may generate is bounded by:

$$\sum_{k=1}^{L} |F1|^k = \frac{|F1|^{L+1} - |F1|}{|F1| - 1} \quad (4.3)$$

This sum is finite. Since the algorithm only processes a finite number of candidate patterns, and each pattern is handled in finite time (i.e., through merging O-trees, counting support, and computing intervals), the overall algorithm must terminate. Thus, by the finiteness of both the maximum pattern length and the number of extensions at each iteration, we conclude that the Minits-AllOcc algorithm terminates after a finite number of steps.

Theorem 4.2

Minits-AllOcc discovers all and only frequent timed sequential patterns (TSPs) from a discretized timed sequence database D with support ≥ min_sup, while correctly computing transition time statistics.

Proof: By Lemma 1, Lemma2, and Lemma 3, Minits-AllOcc discovers a correct and complete set of timed sequential patterns and successfully terminates.

4.1.3 Time Complexity and proof of correctness of MinitsDays

4.1.3.1 Complexity Analysis

Figure 3.38 in Chapter 3 shows the pseudocode of algorithm 3.3: MinitsDays. The variables of the time complexity analysis of Minits-AllOcc are listed as follows:

- N: Number of timed sequences in the Discretized Timed Sequence Database (DTSDB),
- 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 DTSDB
- 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.

4.1.3.2 Proof of Correctness

We prove the correctness, completeness and termination of the MinitsDays algorithm.

Lemma 4.7: Every output timed sequence pattern TSP produced by the MinitsDays algorithm is valid with support ≥ *min_sup,* and accurate temporal statistics.

Proof: We prove this lemma by mathematical induction on the length k of the pattern P.

Step 1: Base Case (k = 1)

The algorithm scans the entire DTSDB once and builds the 1-Sequence Data Structure (1SD), which tracks every occurrence of each item a ∈ Σ along with its timestamps. The support of each item a is computed as support(a) = |{TSi| a ∈ TSi}|. Only those items a for which support(a) ≥ min_sup are inserted into the initial set of patterns: NTSPs←{⟨{a}⟩|support(a) ≥ min_sup}. Hence, all patterns of length 1 in NTSP are guaranteed to be frequent, with accurate support values derived directly from 1SD.

Step 2: Inductive Hypothesis

Assume for all patterns P ∈ NTSP of length k−1, the followings are true:

- support(P)≥min_sup
- Temporal intervals between events in P are accurately computed.

Step 3: Inductive Step (k−1):

Now, we show that every k-length pattern P′ generated from P is also valid and frequent. The algorithm constructs candidate k-length patterns by backward extension:

- Prepending a frequent item a to an existing pattern P, resulting in a new candidate P′ = a•P.

To determine whether P′ is frequent, MinitsDays intersects the timestamp lists of a and P in 1SD. It checks whether the temporal constraint between the new prefix a and the existing pattern P is satisfied. That is, for each occurrence: $t_P - t_a \in$ [min, max]. If these conditions hold for several distinct sequences ≥ min_sup, then, support (a • P) ≥ min_sup. The new pattern P′ is added to NTSP, and its temporal statistics are recorded.

Therefore, by induction, all patterns of length k in NTSP are frequent and their temporal intervals are computed accurately from the timestamps.

Lemma 4.8: MinitsDays ensures the correctness and completeness of the discovered timed sequential patterns when DTSDB undergoes insertions or deletions of timed sequences.

Proof: We show by direct argument how each update case ensures that only frequent patterns are retained, and all newly frequent patterns are discovered when necessary.

Case 1: Insertions. When new timed sequences are inserted into the DTSDB, the algorithm first updates the 1-Sequence Data Structure (1SD) by integrating the new sequences and updating the support counts and timestamp lists for all items. Let the number of newly added sequences be U. For each item a present in the inserted sequences, its support is recalculated as $sup(a) = sup_{old}(a) + sup_{new}(a)$, where $sup_{old}(a)$ is the number of previously sequences before the update containing a, and $sup_{new}(a)$ is the number of newly inserted sequences containing a.

The Timed Sequential Pattern Suffix Tree (TSS-tree) is lazily updated; that is, only the patterns whose support count changed due to updates are recomputed. This approach avoids

redundant work and improves efficiency. If a pattern P becomes newly frequent due to the insertion (i.e., sup(P)≥min_sup), it is discovered through re-computation:

- Its support is re-evaluated using the updated 1SD.
- All possible backward extensions from affected patterns are explored to generate and validate new candidate patterns.
- Any new pattern that satisfies the frequency threshold is added to NTSP.
- Thus, the algorithm ensures no valid new pattern is missed, preserving completeness, and all inserted patterns are validated before inclusion, preserving correctness.

Case 2: Deletions. When sequences are removed from DTSDB, the algorithm decrements the support counts of the corresponding items in 1SD. Specifically, for each item a that appeared in a deleted sequence:

sup(a) ← sup(a)−1, the TSS-tree is updated by identifying and pruning any pattern P for which sup(P) < min_sup. These patterns are removed from NTSP, along with any of their super-patterns, in accordance with the Apriori principle, which states that all super-patterns of an infrequent pattern must also be infrequent.

Since no pattern is removed unless its updated support falls below the minimum support threshold, the algorithm avoids false negatives and maintains correctness. Additionally, all remaining patterns continue to satisfy the frequency condition, and no frequent patterns are omitted—thereby preserving completeness.

Lemma 4.9: The MinitsDays algorithm always terminates after a finite number of steps.

Proof: To show that MinitsDays terminates, we observe that the algorithm operates on a finite input space and performs a bounded number of operations, including recursive calls.

First, consider the structure of the input:

- The Discretized Timed Sequence Database (DTSDB) contains a finite number of timed sequences, denoted N.
- Each sequence has a finite number of events, bounded by L, where L is the maximum sequence length.
- The database is constructed over a finite item alphabet Σ, with |Σ| = M distinct items.
- As such, the total number of possible patterns of length 1 is at most M, where M denotes the number of distinct items in the database. Each item can independently appear as a pattern of length 1. The total number of possible sequential patterns (i.e., combinations of items with order and timing) is upper bounded by:

$$\sum_{k=1}^{L} M^k = \frac{M(M^L-1)}{M-1} \quad (4.4)$$

This is finite since both M and L are finite.

Furthermore, recursive calls in the Get_Descendants() function are made to extend patterns by one item at each level. As the maximum length of a pattern is bounded by L, the depth of recursion is also bounded by L, i.e., depthmax ≤ L. In each recursive step:

- Only valid extensions of frequent patterns are considered.
- The support count for each candidate is computed using precomputed data (1SD).
- Unpromising branches (i.e., those with support below min_sup) are pruned.

Thus, there is no infinite recursion, and the pattern tree cannot grow indefinitely. The total number of possible pattern candidates explored during the entire execution is finite, and each step involves deterministic operations on finite data structures.

Theorem4.3:

Minits+ discovers all and only frequent timed sequential patterns (TSPs) from a discretized timed sequence database *D* with support ≥ min_sup, while correctly computing transition time statistics.

Proof: By Lemma 1, Lemma2, Lemma 3, and Lemma 4, MinitsDays discovers a correct and complete set of timed sequential patterns and successfully terminates.

4.2 Experimental Analysis

We conducted extensive experiments to evaluate the impact of different techniques for discretizing sequence data and to assess the performance of our proposed algorithms in terms of execution time based on min_sup and sequence database size. To present the results, this section is divided into four parts: (1) experimental analysis of Trajectory Segmentation; (2) experimental analysis of Minits+; (3) experimental analysis of Minits-AllOcc; and (4) experimental analysis of MinitsDays.

4.2.1 Experimental Analysis of Trajectory Segmentation

In this section, we present our experiments to compare the performance of the density-based and grid-based as discretization techniques for sequence data and their impacts on trajectory pattern mining results.

4.2.1.1 Experimental Setup

All experiments were performed on a computer with a 2.7 GHz Intel Core i5 processor with 8 gigabyte main memory, running MacOS High Sierra version 10.13.4. The grid-based algorithm was implemented in R using RStudio version 1.1.383. In contrast, the implementations of both DBSCAN [1], where it is a density-based algorithm and SPADE [2], where it is a sequential patterns

mining algorithm, were adapted from the existing packages in R called dbscan and arulesSequences [3].

4.2.1.2 Dataset

For the experiments, we used two real-life trajectory datasets: GeoLife and Deer. GeoLife was collected by Microsoft Research Asia [4] [5], and [6] by tracking the movements of 182 individuals in Beijing, China, over a five-year period (2007–2012). Deer was collected as part of the Starkey project [7], which recorded the movements of 32 deer (20,000+ points) in northeastern Oregon from 1993 to 1996. To prepare the datasets for the experiments, we removed unnecessary attributes and retained only the following: object ID, latitude, longitude, and timestamp. To facilitate interpretation of the results, we re-scaled the trajectory data values to represent how many standard deviations a value deviated from its mean. Consequently, the coordinates of the points were standardized.

4.2.1.3 Determining Optimal Values of Parameters

The two segmentation techniques required values of several parameters before we started performing them. These values may have a significant impact on the performance, but they may not be known in advance, or some available values chosen by other studies were not suitable because of the nature of the datasets. Therefore, we reviewed other studies and used several tools to help us choose the optimal values for these parameters. The first technique, DBSCAN, requires two parameters: minimum number of points MinPts, and epsilon ε. For MinPts, there is no automatic method for choosing its optimal value because the value depends on the nature of the dataset. As a result, we reviewed the existing literature [47] , [1] , [9], [10] and adopted the same value in the other studies for this parameter. [8] used DBSCAN for the GeoLife dataset and selected MinPts = 60 and ε = 0.001. For the Deer dataset, we chose the default value for MinPts for two-

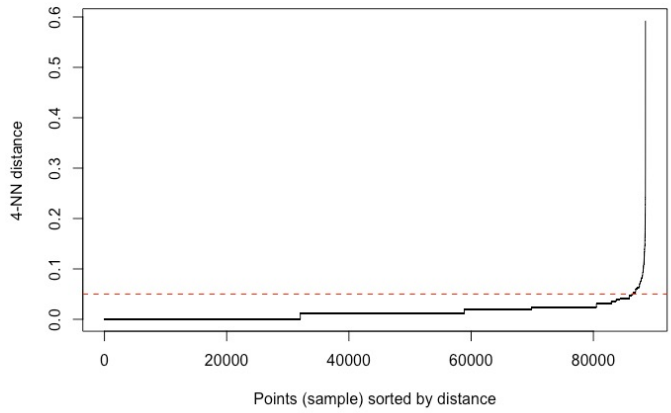


Figure 4.1: Best value of (ε) for deer dataset

dimensional data, which is 4 [1], [9]. For ε, the k-nearest neighbor distance method [10] can be used to estimate its value. The idea behind the knee point is that the k-distance for the core and border points is expected to be within a particular range, while the `points' k-distance is much greater.

Thus, we can detect a change in a curved shape. In Figure 4.1, the knee is visible at around a 4-NN distance of ≅ 0.05, which represents the value of ε for the Deer dataset. The second technique, the grid-based, requires one parameter: cell size. We chose different sizes for the cell, as shown in Table 4.1, and calculated the overall SSE of all cells by using the following formula (4.1):

$$SSE = \sum_{j}^{k} \sum_{i=1}^{n} (x_i - \bar{x})^2 \qquad (4.1)$$

Cell size	SSE	# Cells
1 × 1	35025.3073	45
1.5 × 1.5	5623373.2063	20
2 × 2	11605.6514	7
2.5 × 2.5	20168.0529	6
3 × 3	11613.6555	5
3.5 × 3.5	2319.593	1
4 × 4	0.021037	1

Table 4.1: Cell size vs. SSE measurement for GeoLife dataset

where x_i is an actual point, $\bar{x}$ is the mean of points, n is the number of points of a cell, and k is the number of cells. We chose the cell size to be = 3 × 3 because it gave a low SSE as shown in Table 4.1. Although the next two cell sizes (3.5 × 3.5 and 4 × 4) have a lower SSE, they produced only one cell. This does not help us discover the frequent sequential patterns at the same procedure for Deer and chose the cell size to be equal to 2 × 2.

4.2.1.4 Experimental Parameters

We describe the dynamic parameters, their ranges, and their default values used in this analysis, as summarized in Table 4.2. For each experiment, various values within a parameter's range were tested while assigning default values to the other parameters. The ranges were tailored to each dataset, as described below: For Epsilon ε, the default value was set to the optimal value determined in Section 4.2.1.3, which was also the median of its range. For GeoLife, the range was from 40 to 80, with a default value of 60, based on the literature review in Section 4.2.1.3. For Deer, the range was from 1 to 8, with a default value of 4, similarly informed by the literature. For the

Parameter Name	**GeoLife Dataset**		**Deer Dataset**	
	Range of values	Default values	Range of values	Default values
Epsilon ε	0.0001 – 0.1	0.001	0.01 – 0.1	0.05
MinPts	40 – 80	60	1 – 8	4
Cell size	1 × 1 – 4 × 4	3 × 3	1 × 1 – 4× 4	2 × 2
Number of trajectories	1 – 182	91	1 – 33	16
min_sup	20% – 80%	50%	20% – 80%	50%

Table 4.2: List of parameters for experiments

cell size, the range was set from 1 × 1 to 4 × 4, as discussed in Section 4.2.1.3. The default value was 3 × 3 for GeoLife and 2 × 2 for Deer, based on the technique described in Section 4.2.1.3. The ranges for the number of trajectories depended on the data available for each dataset. For GeoLife, the range was from 1 to 182, with a default value (mean) of 91. For Deer, the range was from 1 to

32, with a default value (mean) of 16. The support (min_sup) had a range from 20% to 80% with the default value of 50%, which was the median of the interval.

4.2.1.5 Evaluation Metrics

We evaluated the impact of trajectory segmentation techniques on discovering trajectory sequential patterns by measuring two key metrics: execution time (ET) and the number of discovered patterns (#patt). Execution Time (ET): Measured in seconds, it was recorded from the moment the dataset was read until SPADE produced the patterns. Number of Discovered Patterns (#patt): A higher number of patterns discovered by SPADE after applying a segmentation technique indicates more accurate and complete results, as SPADE is designed to identify all frequent sequential patterns with a frequency greater than or equal to min_sup [26]. We employed two segmentation techniques, density-based and grid-based, to generate sequential patterns with varying levels of granularity. Since these patterns were not directly compatible due to their differing granularities, we transformed them into a common lower granularity level (point level) for comparison. This allowed us to determine whether the patterns generated by the two techniques were identical. Additional details are provided in Section 4.2.1.6.6.

4.2.1.6 Experimental Results

Five sets of experiments were conducted to study the impacts of the five parameters: ε, MinPts, cell size, number of trajectories, and min_sup on the performance of the two techniques. Table 4.3 presents the average performance of the density-based and grid-based segmentation techniques for GeoLife and Deer datasets.

Dataset	Density-based segmentation technique		Grid-based segmentation technique	
	ET (sec)	NumOfPatt	ET (sec)	NumOfPatt
GeoLife	5.2822	1	20.3453	3
Deer	1.5897	9	3.8445	2071

Table 4.3: Average of performance results

The density-based technique requires less time to discover the frequent sequential patterns than the grid-based technique. DBSCAN first labels all points as either core, border, or noise, then eliminates the noise points. That means the number of points is now fewer than the original dataset. After that, DBSCAN goes through core and border points and assigns a cluster ID to each point. However, in our implementation, the grid-based technique first labeled all points with its row and column index and did not remove any points. That means the number of points was still the same as the original dataset. Then, the algorithm determined the cell ID for each point. Depending on this, when the algorithm converted the original sequence database in the new form of database in terms of cluster ID or cell ID, the length of sequences could be different. For the density-based technique, some of the points did not belong to any cluster since some were categorized as noise. Thus, they were removed from the sequence and the length of that sequence was shortened. In contrast, for the grid-based technique, each point must belong to a cell, so the length of the sequence

will be longer than the sequence in the density-based sequence database. Therefore, SPADE needs more time to mine frequent trajectory patterns from the grid-based sequence database. Nevertheless, the grid-based technique outperformed the density-based technique in terms of how many patterns were discovered. The first reason is because the grid-based technique does not discard any points, and the strongest relationship appears when there are many points. Also, the length of the sequences in the grid-based technique are longer than the length of sequences in the density-based technique, especially if many points belong to adjacent cells. For instance, if we have a sequence < P_1, P_2, P_3...., P_{10}> and each point belongs to the three adjacent cells alternately, the transformed sequence will appear as <C_1,C_2,C_3,C_1,C_2,C_3,C_1,C_2,C_3,C_1>. Thus, the length of the sequence is larger, which allows SPADE to discover more patterns. The second reason depends on how the algorithm treats the points. The grid-based technique depends only on one attribute (cell size), which corresponds to the radius (ε) parameter in the density-based technique. DBSCAN must also handle additional parameters (MinPts); after eliminating the noise points, two types of points remain: the core and border points. Usually, the number of clusters relies on the core points, and the density of each cluster relies on the border points. When we performed the experiment using the default values of the parameters, we always had the same number of clusters or cells. However, the density of clusters was always going to be different because of a different implementation of DBSCAN. Each implementation could employ a different strategy to decide the cluster of a border point. Some may arbitrarily assign it to any cluster, while others could assign the border point to the closest core point, if there are many of them around the border point and they belong to different clusters. Therefore, a border point could belong to a different cluster every time the DBSCAN is executed. Thus, the frequency of a cluster ID occurrence in a sequence database varies from time to time, so some patterns could be lost.

In order to understand how each technique would behave, in some experiments we separated the execution times, which are measured in seconds (sec), for each technique into four types: time required for reading input data (denoted as T_1); time needed by DBSCAN to assign cluster IDs to points, or by cells partition to assign cell IDs to points (T_2); time to convert the original sequence database to the new one in terms of cluster or cell IDs (T_3); and time to execute the sequential pattern mining algorithm, SPADE (T_4). Since each of the techniques required a user-defined parameter, we investigated the effect of changing the values of these inputs on discovering the frequent sequential patterns at the end.

4.2.1.6.1 Impact of Cluster Radius (ε)

When we increased the value of Epsilon (ε), we observed that total execution time also increased as shown in Figure 4.2. To understand the reasons, we studied many factors such as number of clusters and sub-execution times, as discussed in Section 4.2.1.6. We found that the number of clusters decreases because when the radius of a cluster increases, the number of core points will also increase due to the possibility of finding points that are equal to or more than MinPts. Therefore, more core points will be close enough to be in the same cluster. Also, the number of noise points reduces because the radius of a cluster is getting larger and larger, which means a noise point would be either a core point if it has the same or more points within (ε), or a border point that could be a neighbor of a core point. Minimizing the number of noise points leads to an increase in the numbers of core and border points, which means that DBSCAN needs more time to work on these points to determine which clusters they belong in. The execution time of SPADE, T_4, relies on the number of clusters. When the number of clusters increases, the number of cluster IDs that are used to replace the points also increases. Hence, the number of ID-lists increases, the number of candidates increases, and the time to check candidates' frequency also increases.

Ultimately, we conclude that increasing the value of (ε) will also increase the overall execution time (ET).

For the number of patterns, we saw in Figure 4.3 that it is neither predictable nor stable.

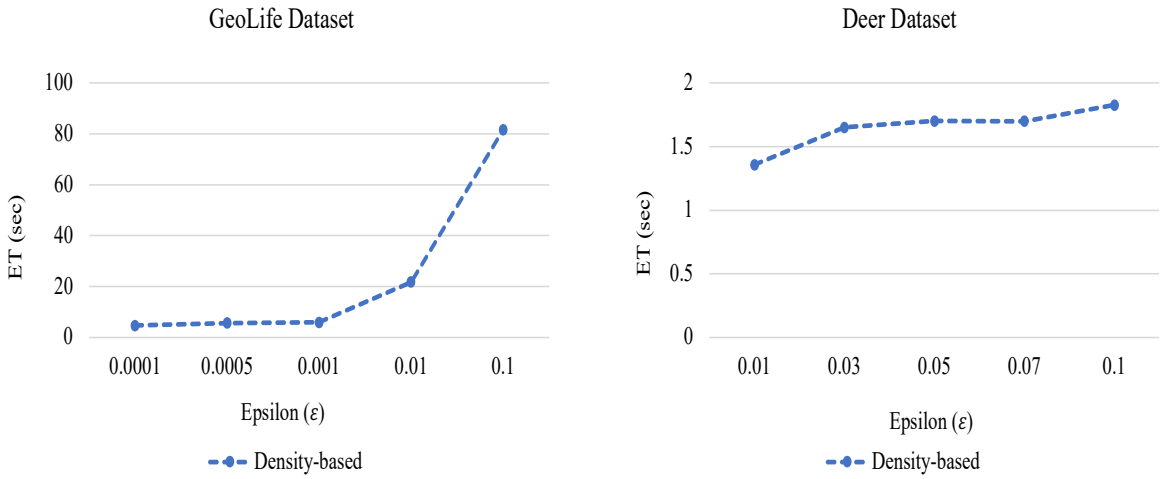


Figure 4.2: Impact of ε on Execution time (ET)

When the value of (ε) increases, the number of patterns could stay the same, increase, or decrease. In some cases, the number of patterns did not change even though the number of clusters decreased, because the density (number of points) of some clusters was lower than other clusters. If we lose these clusters, they do not have a significant impact on the number of discovered patterns because they do not appear often in the sequence database. However, in other cases, even when the number of clusters decreases, the number of patterns increases because some clusters merge with other clusters, which become denser. For instance, we discovered 3 patterns where ε = 0.03 and 9 patterns where ε = 0.05 for the Deer dataset. The number of clusters was 216 for the first case and 49 for the second case. When ε is bigger, the number of clusters is smaller, and thus, the number of points in each cluster is larger. When ε is increased, the radius covers a larger area, so some previous clusters

are merged with other clusters. This leads to more denser clusters that apparently occur frequently because many points now belong to fewer clusters. In the last case, the number of patterns decreases because the number of clusters decreases, so the length of a sequence in the database changes. For example, for the Deer dataset, when we set ε = 0.07, we had 16 clusters. Then, when we replaced the points with its associated cluster ID, we had sequences of varied length starting from 1 itemset to 60 itemsets. In contrast, when we set ε = 0.1, we had 7 clusters. Then, when we replaced the points with its associated cluster ID, we had sequences of varied length starting from 1 itemset to 40 itemsets. As long as we have long sequences in a database, we have a high chance of discovering more patterns that reflect more details regarding the tracking of an object's movement. In the end, we conclude that the number of discovered patterns will behave differently when increasing the value of (ε).

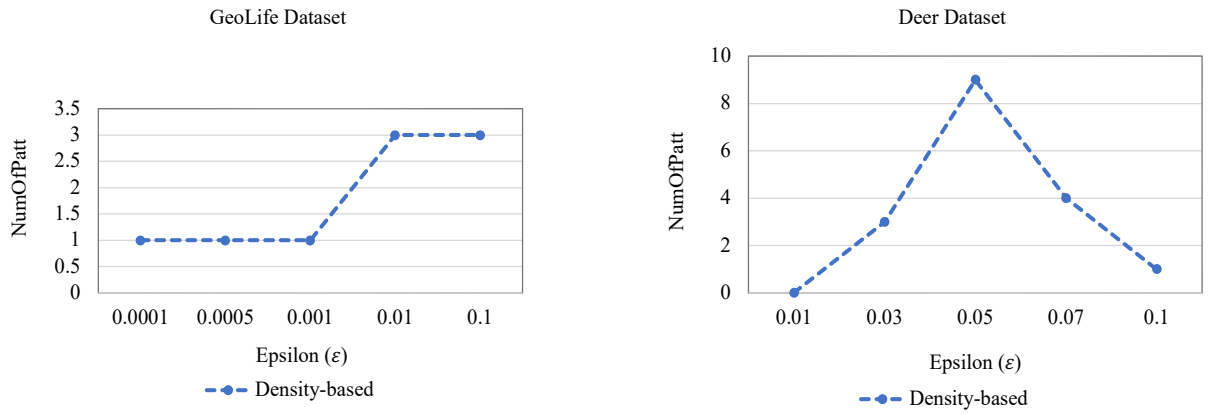


Figure 4.3: Impact of ε on number of patterns

4.2.1.6.2 Impact of Minimum Number of Points (MinPts)

We conclude that when the MinPts increases, the execution time (ET) decreases as shown in Figure 4.4. When we analyze the whole execution time, we found that the execution time of DBSCAN (T_2) dominates ET. For example, 80% of overall ET is taken up by T_2 for the GeoLife dataset. The number of clusters can be increased or decreased depending on the status of a core point and a border point. Increasing the number of MinPts means that we require a denser neighborhood for a point, which produces fewer core points because it becomes harder to satisfy the MinPts conditions when the radius (ε) of a cluster ID is fixed. If the core points are still close enough within (ε) from a former cluster, the number of clusters does not change because DBSCAN puts them in the same cluster. If the core points are close enough to some core points but no longer close to others, the number of clusters increases because each group of connected core points is assigned to a separate cluster. When a core point is not surrounded by a sufficient number of points which are equal to or more than MinPts, or when a point is not a border, the number of clusters decreases because this point is labeled as a noise point. For a border point, if it is still not surrounded by the acceptable number of points but is in the vicinity of a core point, it will belong to the cluster associated with the core point. If the border point is within a neighborhood of two core points, it will belong to either one. Usually, border points are assigned to the cluster they are first discovered from. Therefore, each time DBSCAN is run, the size of the cluster could change. However, the border point would not belong to any clusters because if its closest core point is not be classified as a core point any more or the border point cannot be reached from other core points, the point is labeled as noise point. Based on that, the number of noise points increases, because if a point does not have a sufficient number of points within (ε), it will not be a core point. Thus, it is less likely to also find a border point that is in a neighborhood of a core point within that fixed cluster size. Given

the increase in the number of noise points, the number of core and border points is lower. Thus, the algorithm needs less time to determine the cluster ID for each point. There is a direct relationship between the number of clusters and the performance of SPADE. As explained in Section 4.2.1.6.1, if the number of clusters increases, the execution time of SPADE also increases, and vice versa.

The number of patterns that can be discovered by SPADE may not change, or it may increase or decrease as shown in Figure 4.5. If the number of clusters decreases but the remaining ones still contain a huge number of points, that means these cluster IDs occur frequently in a sequence

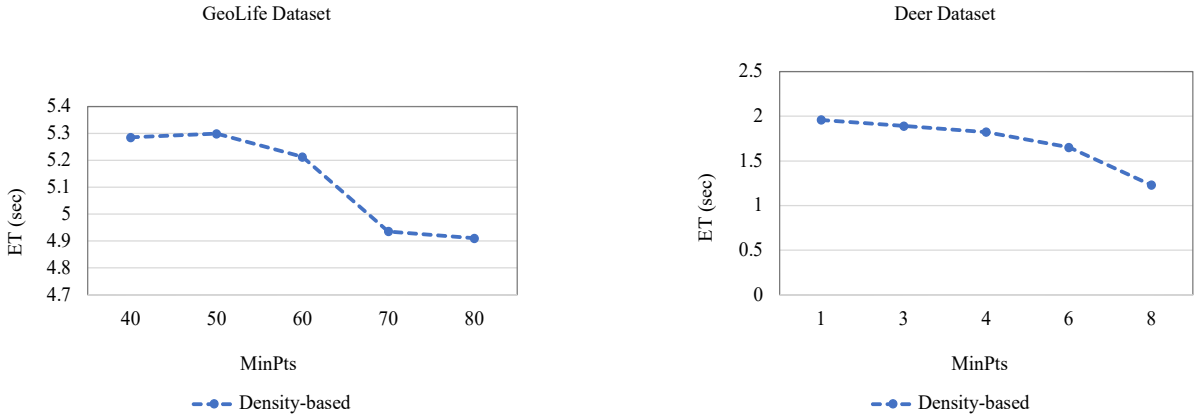


Figure 4.4: Impact of MinPts on Execution Time (ET)

database, so SPADE always discovers trajectory patterns from these clusters. In other words, the density of a cluster (number of points in cluster) also affects the number of patterns. Some of the clusters always have thousands of points, for example, versus some small clusters that have hundreds of points. The disappearance of small clusters does not impact the number of discovered patterns because they do not appear periodically in the sequence database. However, since the number of clusters decreases and the number of noise points increases, more points will be ignored. Thus, the possibility of finding the most frequent sequences decreases. For the last case, when the

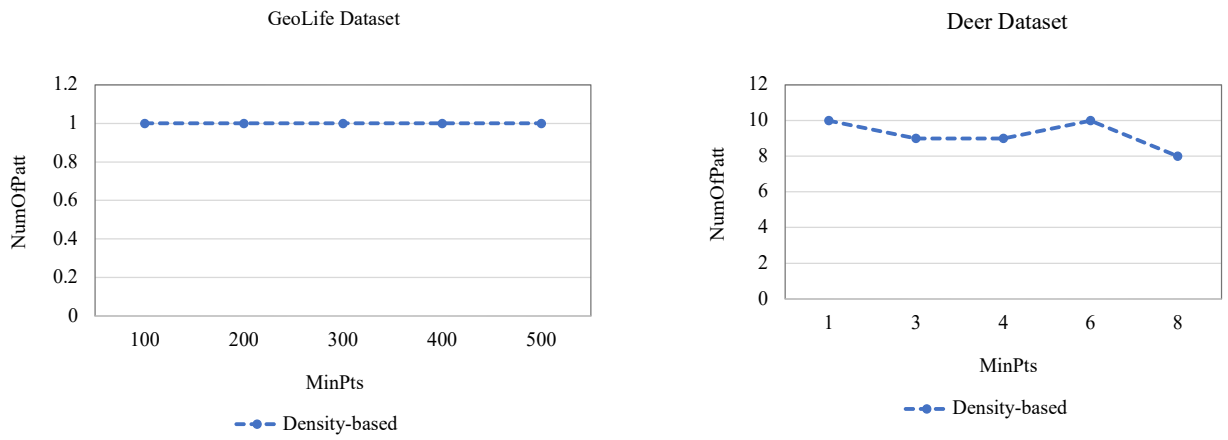


Figure 4.5: Impact of MinPts on Number of Patterns

number of patterns increases, the number of clusters decreases because other former clusters are merged with other clusters since they are not fulfilling the MinPts condition. Therefore, many clusters have a higher number of points than before, and their IDs will most likely arise in the database. In conclusion, we notice that increasing the number of minimum points had different impacts on the overall execution time (ET) and number of patterns (#patt).

4.2.1.6.3 Impact of Cell Size

When the size of the cell is increased, we observe in Figure 4.6 the total execution time (ET) decreases. After close examination, we concluded that SPADE execution time (T_4) affects the total execution time (ET). When the cell size is increased, the number of cells decreases, and the execution time of SPADE (T_4) also decreases. The reason is that the length of sequences decreases because the number of points is replaced by the smaller number of cells. However, the execution time of cells partitioning T_2 does have a huge impact on ET, for example, 90% of ET is taken by T_2 but the differences in term of times between different experiments are negligible when the cell size is increased. In other words, we can consider T_2 as constant for all experiments because it depends on the number of the points, which is static. For this technique, there are no longer any

noise points because each point must belong to a specific cell. For the task of discovering sequential patterns, we observed that changing the cell size produces a different number of patterns. The patterns will not change and SPADE will always discover the same number. Even though the grid-based technique considers all points, some cells may not contain any points, so these cell IDs do not show up in the database at all. Increasing cell size would provide a bigger cell, but there still may not be any points. Therefore, the database will contain the only cells that have points and SPADE finds the same patterns every time.

On the other hand, the number of patterns could increase or decrease as shown in Figure 4.7. In some scenarios, the patterns increased because this technique suffers from the issue of adjacent

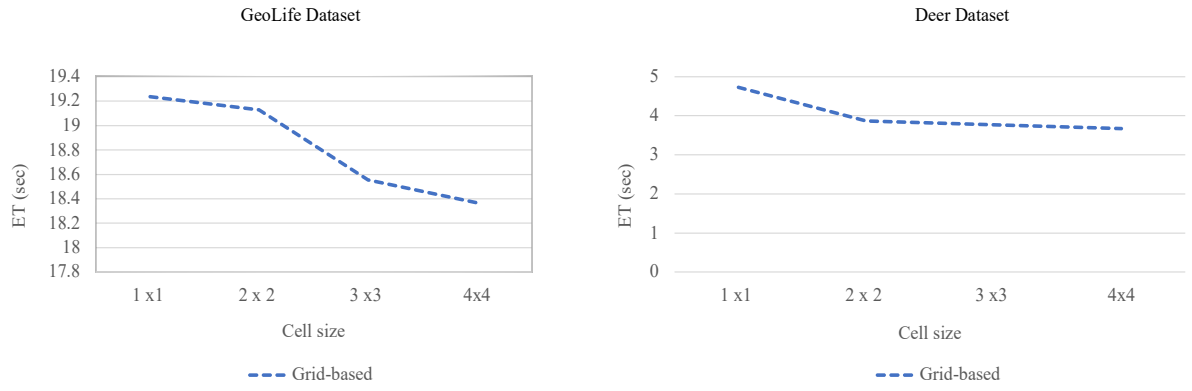


Figure 4.6: Impact of cell size on Execution Time (ET)

cells that belong to different cells, as discussed in Section 3.2.2.2. When we have many small adjacent cells that consist of points, some of the points belong to $cell_i$ and others to $cell_{i+1}$, and at the end, all these points are considered to belong to the same cell. In other scenarios, the number of patterns decreases because it also depends on the length of sequences in the database. As discussed in Section 4.2.1.6.1, since the cluster or cell number decreases, the IDs become fewer. So, the length

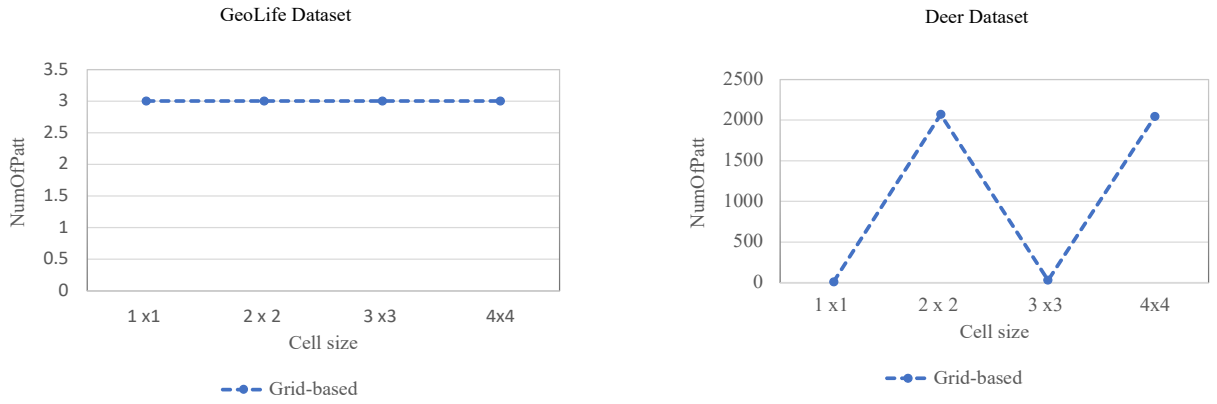


Figure 4.7: Impact of cell size on Number of Patterns

of sequences lessens, and SPADE cannot find sequential patterns longer than the longest sequence (in the worst case). Finally, increasing the size of a cell will reduce the overall execution time (ET) and produce an unpredictable number of patterns.

4.2.1.6.4 Impact of the Number of Trajectories

In this set of experiments, we compared the execution when the number of trajectories was increased. From Figure 4.8, we can observe that overall execution time (ET) increases. This is acceptable because when we increase the number of trajectories, the number of points also increases. Accordingly, the time of reading input (T_1), time for grouping points into a cluster or cell (T_2), time to convert sequence database (T_3), and time to apply SPADE also increase. In addition, we notice that the time for grouping points into a cluster or cell (T_2) dominates the overall execution time (ET). For the GeoLife dataset, when the density-based technique was applied, DBSCAN took around 65% of ET, and for the grid-based cell, segmentation took 85% of ET. For the Deer dataset, when the density-based technique was applied, DBSCAN took around 50% of ET, and for the grid-based cell, segmentation took 60% of ET. Comparing the performance of the two techniques, the grid-based technique took more time than the density-based technique, which was expected since

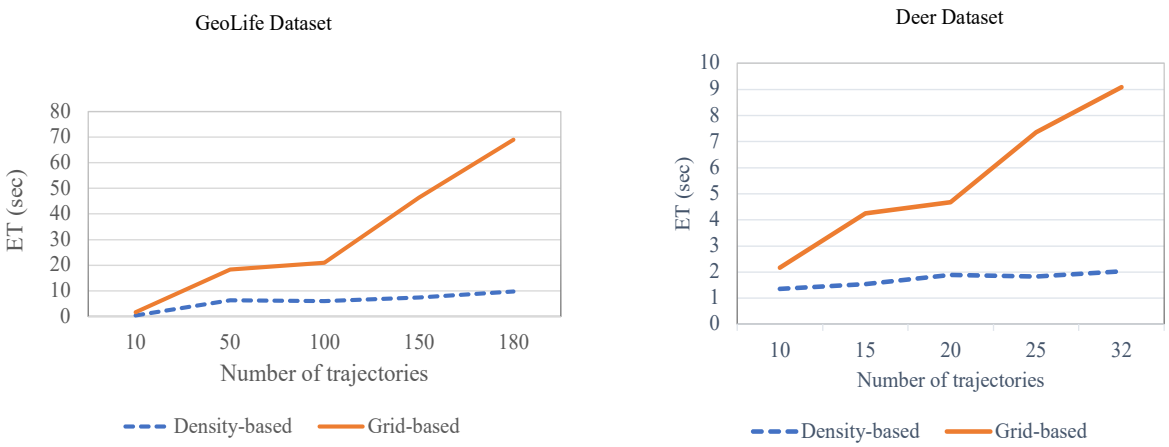


Figure 4.8: Impact of the number of trajectories on Execution Time (ET)

T_3 dominates the overall execution time. Given the increasing number of points, the execution time of the grid-based technique is always higher. By using DBSCAN, some points are eliminated because they are noise points, while all the points are retained by the grid-based technique.

From Figure 4.9, the number of the discovered patterns exhibits variation in behavior because, as discussed previously, it depends on the number of clusters/cells and the number of points in each cluster/cell. If some former clusters/cells become denser when the number of trajectories is increased while others do not, the number of patterns remains the same. If increasing the number of trajectories creates more clusters or makes some previous cells denser, the number of patterns will increase. If the cluster/cell IDs decrease, or the length of sequences shortens, the number of patterns will decrease. The details of all cases are discussed in Sections 4.2.1.6.1, 4.2.1.6.2, and 4.2.1.6.3.

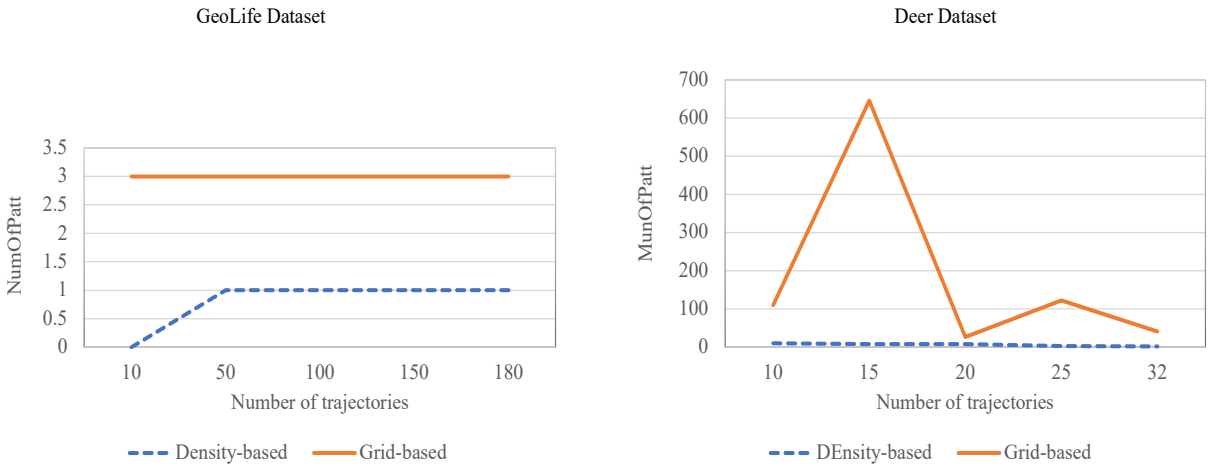


Figure 4.9: Impact of the number of trajectories Number of Patterns

4.2.1.6.5 Impact of Minimum Support

In this set of experiments, we compared (ET) and (#patt) for different values of minimum support threshold. From Figure 4.10, we can see that when the minimum support was increased, the execution time of both techniques decreased (note that the plot has straight lines because the difference in ET was about two to three seconds). The reason is that the SPADE algorithm generates fewer candidates when the min_sup is high. In other words, SPADE needs less time to eliminate infrequent sequences by scanning the ID-lists of candidates and testing each support count of a candidate against the min_sup. Since all other parameters (ε, MinPts, cell size, and the number of trajectories) are steady, the SPADE execution time (T_4) does not change that much and does not control the ET. Again, the execution time of segmentation dominates the overall ET. Therefore, we can observe that the ET of the grid-based technique is less than that of the density-based technique for the GeoLife dataset and vice versa for the Deer dataset; this was discussed in Section 4.2.1.6.4.

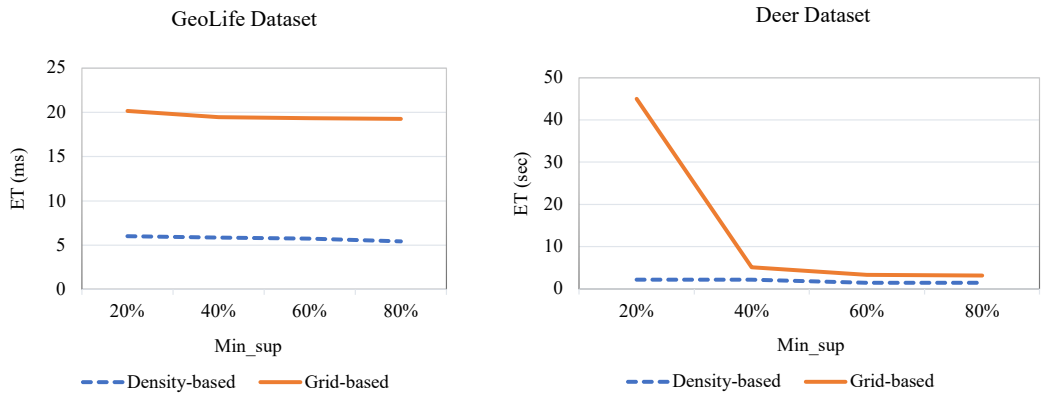


Figure 4.10: Impact of minimum support (min_sup) on Execution time (ET)

The other observation was based on the number of frequent sequences, as shown in Figure 4.11, generated by these segmentation techniques. As mentioned above, when the min_sup is increased, the number of patterns that are discovered by SPADE is going to be less because finding large patterns that satisfy the min_sup condition is difficult. For example, if a sequence database has 100 tuples, the number of patterns that exist at least in 30 tuples (min_sup=30%) is larger than the number of patterns that exist at least in 80 tuples (min_sup=80%). As [2] stated, SPADE discovers all frequent sequential patterns that satisfy the condition that their frequency is greater than or equal to min_sup, so every time we ran the algorithm, the grid-based technique provided either the same number or more than the density-based technique. For example, when min_sup was set to 40% for the GeoLife dataset, the grid-based technique discovered three patterns, while the density-based technique discovered one pattern. Therefore, the grid-based technique needs more

time to mine the ID-lists of all candidates that are generated by SPADE. Thus, there is a necessary trade-off between getting more accurate patterns and minimizing the execution time of SPADE.

4.2.1.6.6 Matching Sequential Patterns

Since we used different techniques with different parameters and strategies to segment trajectories, we wanted to make sure that these two techniques discovered the same patterns. Therefore, after assigning the cluster ID and cell ID to each point, we used the data structure HashMap and linked the list map to find the matching areas between them, as well as maintained the order of sequences based on time stamps. Then, we found the matching points between each cluster and cell. For example, in one experiment using GeoLife, when the cell size was larger than cluster size, we had 2 clusters and 20 cells. Then, we found how many matching points were between each cluster and cell. In the case that a cell matches a cluster with zero points, it means that they do not overlap. After that, for each cell that had some matching points with a cluster, we changed the cell ID so that it was similar to the cluster ID. Next, we compared each sequential pattern of cells with the sequential patterns of clusters and made sure that all patterns discovered by the grid-based technique matched all the patterns discovered by the density-based technique. In contrast, if the cluster radius was larger than the cell size, we found the matching points between the clusters and cells. Then, we changed the cluster ID to be similar to the cell ID and compared the discovered sequential patterns. We found that all the sequential patterns discovered by the density-based technique were also discovered by the grid-based technique, but not conversely. The patterns

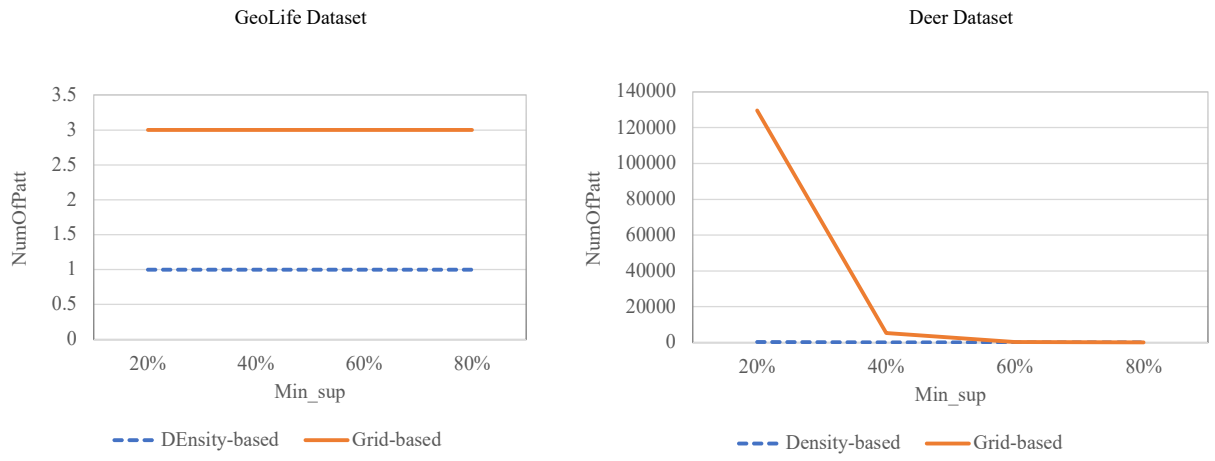


Figure 4.11: Impact of minimum support (min_sup) on Number of Patterns

discovered by the grid-based technique were not discovered by the density-based technique. From this result, we can conclude that the grid-based technique helps discover more hidden sequential patterns than the density-based technique, and as mentioned, SPADE discovers all frequent sequential patterns.

4.2.2 Experimental Analysis of Minits+

In this section, we present the results of experiments conducted to evaluate the performance of Minits+ on both single-core and multi-core CPUs, using real and synthetic datasets. After extensive testing, we found that Minits+-pseudo DB on a single-core CPU significantly outperformed Minits+-projected DB on the same hardware. As a result, for subsequent experiments, we selected Minits+-pseudo DB and re-implemented it for multi-core CPUs. From this point forward, when we refer to Minits+, we specifically mean the Minits+-pseudo DB version. We distinguish between the single-core CPU and multi-core CPU implementations by using the terms Minits+ and MMinits+ respectively.

4.2.2.1 Experimental Setup

All experiments were performed on a computer with a 2.10 GHz Intel Xeon(R) processor with 62-gigabyte main memory, running Ubuntu 18.04.1 LTS. The Minits+ algorithm was implemented in Java 1.8.

4.2.2.1.1 Competing Algorithm

In Chapter 2, Table 2.1 presents most of the existing techniques for discovering timed sequential patterns from static Discretized Timed Sequence Databases. However, all the algorithms listed in that table—except for T-Prefix [11], T-CSpan [12], SID-PrefixSpan [13], and SP-TCSpan [14] produce a different pattern format that does not align with the objectives of this dissertation. While T-Prefix, T-CSpan, SID-PrefixSpan, and SP-TCSpan discover patterns like our timed sequential patterns, they fail to address certain research challenges discussed in Chapter 1. Specifically, these algorithms do not adequately handle the following issues: tracking all possible occurrences of a sequence, dealing with different types of database modifications, and handling large-scale sequence databases as shown in Table 2.2. As stated in Chapter 2, this work focuses on techniques that generate the complete set of sequential patterns. Among the algorithms T-Prefix, T-CSpan, SID-PrefixSpan, and SP-TCSpan, we excluded T-CSpan and SP-TCSpan because they generate only a partial set of sequential patterns rather than the complete set. We selected SID-PrefixSpan as the competing algorithm since it is more recent than T-Prefix.

4.2.2.1.2 Datasets and Experimenal Parameters

We used one real-life dataset and one synthetic dataset. The real dataset was T-Drive [15] [16], and the synthetic dataset was generated using a tool provided by the SPMF Library [17]. Also, we set several parameters to conduct the experiments on each dataset. There are two types of parameters, static and dynamic. The values of the static parameters did not change in all

experiments, while the values of the dynamic parameters changed from one experiment to another. Essentially, we had two different parameters. The first was minimum support threshold (min_sup). It is a user-defined threshold applied to find all the frequent sequential patterns in an n-dimensional events sequence database. The second was the number of sequences in the event database (# sequences in DB). Since each sequence is represented as a tuple in the database, this parameter refers to the number of tuples in the database. We conducted the experiments on all datasets and the results were consistent. We will explain the type of parameter, and for the dynamic parameters we will explain their ranges and their default values of this analysis as summarized in Tables 4.4 and Table 4.5 for the T-Drive and synthetic dataset, respectively. When an experiment was conducted, we chose various values of one parameter within its range and assigned the default value to the other parameters.

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 week. The dataset contains the following attributes: User ID, timestamp, latitude, and longitude as shown in Figure 4.12**.** 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) [1]; the results of the discretized sequences are shown in Figure 4.13. Taxi 1 now has a sequence that contains events in terms of cluster IDs. 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 readers to [18].

One of the dynamic parameters is (min_sup), which has a range from 20% to 80% with the default value = 50%, which is the median of the interval. Then, the number of sequences ranges from 1 to 2000 and its default value (median) is 1000, as shown in Table 4.4.

Parameter Name	Range of values	Default value
min_sup	20% – 80%	50%
# sequences in DB	1 – 2000	1000

Table 4.4: Parameters list for T-Drive

Parameter Name	Range of values	Default value
min_sup	20% – 80%	50%
# sequences in DB	1 – 100,000	50,000

Table 4.5: Parameters list for synthetic dataset

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 4.12: Sequential database for the T-Drive dataset before discretization by DBSCAN

As mentioned earlier, the synthetic dataset was generated by using the implementation tool in [17]. We study the impacts of all two parameters shown in Table 4.5. The support (min_sup) parameter had a range from 20% to 80% with a default value = 50%, which is the median of the interval. The range of the parameter (number of sequences in a dataset) was chosen to be from 1 to 100,000 and its median value of 50,000 to be the default value.

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 4.13: Sequential database for the T-Drive dataset after discretization by

4.2.1.2.3 Evaluation Metrics

The evaluation metrics include two measurements: (1) the execution times (ET) of the two algorithms (Minits+ and MMinits+); and (2) the number of patterns (# patterns) generated by these two algorithms.

4.2.2.2 Experimental Results

In this section, we present the performance of the two algorithms, Minits+ and MMinits+, in terms of execution time (ET) and number of discovered patterns (# patterns) for the T-Drive and synthetic datasets. Again, Minits+ is inspired by PrefixSpan, which is a well-known algorithm for mining sequential patterns. Even though there are other algorithms that outperform PrefixSpan, such as FAST [19], there is no proof of correctness of the algorithm.

4.2.2.2.1 Accuracy

To validate that Minits+ always gives the same patterns in terms of numbers and contents excluding the time intervals as those produced by PrefixSpan [20], a tool was implemented in Java 1.8.0_91 using Eclipse version 4.5.2.

First, all temporal relations were removed from the patterns generated by Minits+. Then, these patterns were compared to the patterns generated by PrefixSpan to make sure that each pattern generated by one algorithm had a match generated by the other algorithm. For example, a pattern X was generated by PrefixSpan X= < {TC1}, {WC3}, {TC1WC3} > and a pattern Y was

generated by Minits+ Y= < {TC1} [2,5] {WC3} [3,7] {TC1WC3} >, we took away the temporal relations from pattern Y and compared them with pattern X. When the order of at least one value was different, pattern X did not match pattern Y. For instance, Z= < {WC3} [2,5] {TC1} [3,7] {TC1WC3} > did not match pattern X because the value {WC3} occurred before {TC1}. However, within the last event {TC1WC3} the order does not matter because all the values appeared at the same timestamp. At the end of this experiment, we found that both algorithms, Minits+ and MMinits+, discovered the exact same patterns that were produced by PrefixSpan. In other words, both algorithms produced accurate sequential patterns.

4.2.2.2.2 Execution Time

The execution time (in seconds) was recorded starting from the moment that a dataset was read to the moment that the algorithm produced the patterns. Table 4.6 shows the average performance of the two algorithms for the T-Drive and synthetic datasets. Table 4.6 shows that the execution time (ET) of MMinits+ decreased by about 74% compared to that of Minits+.

Datasets	Minits+		MMinits+	
	ET (sec)	# patt	ET (sec)	# patt
T-Drive	81.413	126	47.604	126
Synthetic data	25.32	3756	6.60	3756

Table 4.6: Average Execution times and # patterns of the algorithms on the T-Drive and synthetic dataset

4.2.2.2.3 Impact of Miniumum Support

In this set of experiments, we compared the execution time (ET) and the number of patterns (#patt) for different minimum support thresholds (min_sup). From Figure 4.14 and Figure 4.15, we observe that as the minimum support min_sup increases, the execution time of all algorithms

decreases. This is because the algorithms generate fewer patterns at higher min_sup values, as fewer sequences satisfy the min_sup condition. With a large amount of data and a significant number of discovered patterns, MMinits+ outperformed both Minits+ and SID-PrefixSpan, as shown in Figure 4.14 and Figure 4.15. Therefore, utilizing multi-core CPUs is more beneficial when the sequence database and the number of expected patterns is large. In addition, Minits+ outperformed SID-PrefixSpan, particularly at higher min_sup values. As mentioned earlier, SID-PrefixSpan uses a physical projected database, whereas Minits+ employs a pseudo-projected database. The pseudo-projected database is significantly smaller in size compared to the physical projected database, as it contains only identifiers rather than complete data. This size reduction minimizes the cost of projection, leading to faster execution times and improved efficiency.

The multi-core CPU version also proved to be efficient at lower min_sup values. However, as seen in Figure 4.14 and Figure 4.15 the execution times of all algorithms were very close when the min_sup exceeded 60%. This is because the number of candidates and, consequently, the number of patterns decreases, reducing the need for many threads. In Figure 4.14, Minits+ outperformed SID-PrefixSpan in terms of execution time, achieving a 9% improvement. Additionally, MMinits+ outperformed Minits+, demonstrating a 34% improvement in execution time. In Figure 4.15, Minits+ outperformed SID-PrefixSpan in terms of execution time, achieving a 36% improvement. Additionally, MMinits+ outperformed Minits+, demonstrating a 40% improvement in execution time.

Another observation was related to the number of frequent timed event patterns generated by these algorithms. Both algorithms discovered the same number of patterns, resulting in overlapping curves in Figure 4.14 and Figure 4.15. As the min_sup increases, the number of patterns generated by Minits+ decreases because fewer patterns satisfy the min_sup condition.

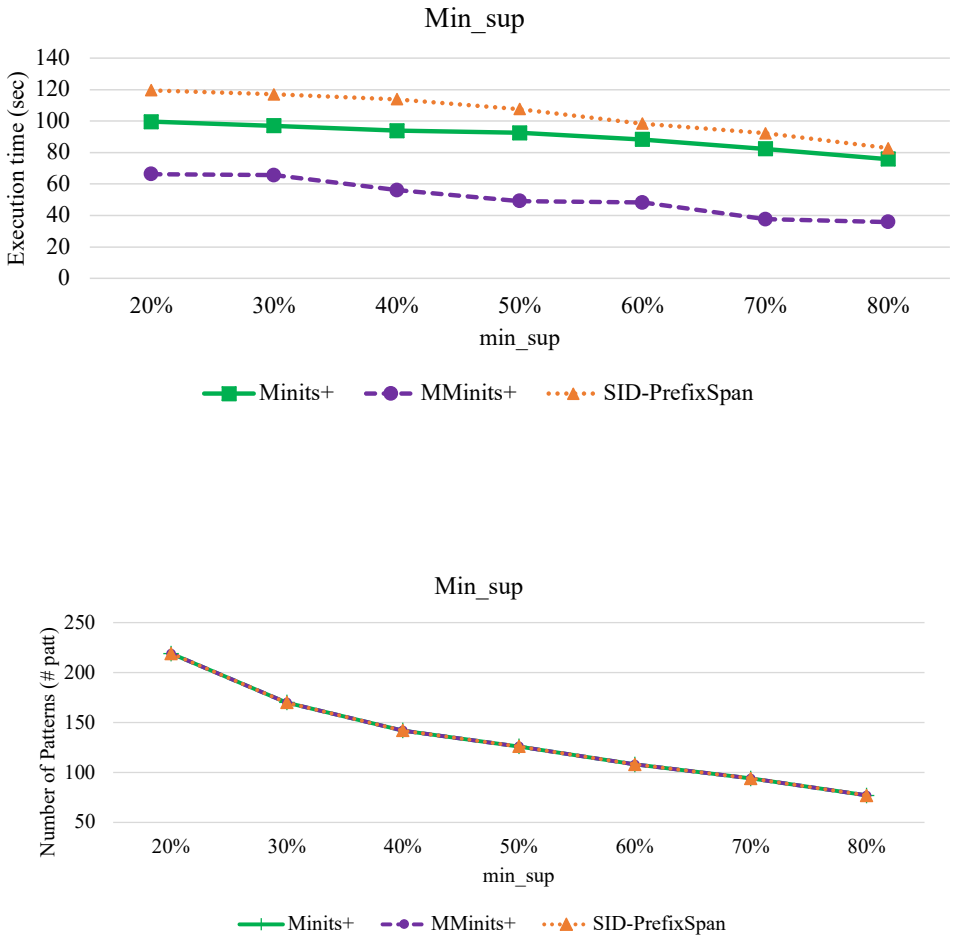


Figure 4.14: Impact of min_sup on execution time (ET) and number of patterns (#patt) using T-Drive dataset

Increasing the min_sup threshold raises the percentage of sequences in the database that a candidate sequence must match, meaning a sequence must occur more frequently to be returned by the algorithm. This becomes increasingly rare, especially as the min_sup threshold continues to rise.

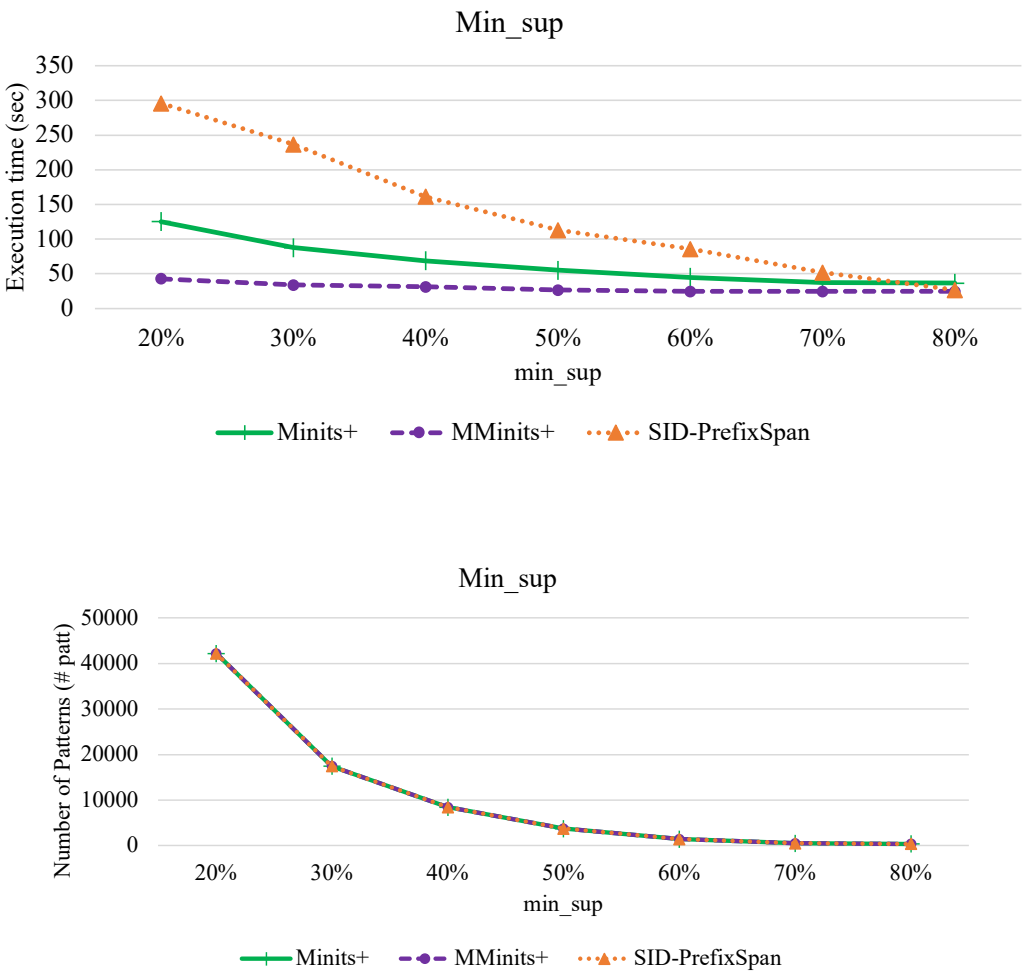


Figure 4.15: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Synthetic dataset

4.2.2.2.4 Impact of the Number of Sequences in the Database

In these experiments, we compared the execution time (ET) and the number of discovered patterns (#patt) for different database sizes (i.e., the number of sequences). From Figure 4.16, we observe that as the number of sequences increased, the execution times of both algorithms also increased. This is because the algorithms required more time to analyze the additional sequences in the database to determine whether they contained frequent timed sequential patterns.

Increasing the number of timed sequences in the database leads to a corresponding increase in the size of the projected database used by both Minits+ and SID-PrefixSpan. However, as previously mentioned, the size of the pseudo-projected database used by Minits+ is smaller than that of the physical projected database used by SID-PrefixSpan used by SID-PrefixSpan. This means that the cost of scanning the pseudo-projected database, and the memory required to store it are significantly reduced. These factors have a substantial impact on the performance of the algorithm, especially as the size of the original database continues to grow.

Another observation pertains to the number of frequent timed event patterns generated by these algorithms. As shown in Figure 4.17, when the number of sequences increased, the number of discovered patterns by Minits+, MMinits+ and, SID-PrefixSpan also increased. This is because the likelihood of finding more patterns in the new sequences that satisfy the min_sup condition (set to 50% as the default value) increases with the size of the database. In Figure 4.17, Minits+ outperformed SID-PrefixSpan in terms of execution time, achieving a 34% improvement. Additionally, MMinits+ outperformed Minits+, demonstrating a 60% improvement in execution time.

With an increase in the number of sequences, Minits+ checks if new patterns can emerge that were not present in the previous sequences. It then evaluates their support against the min_sup threshold. Additionally, the support of some previously infrequent patterns may become sufficient when new sequences are added to the database, thereby changing their status to frequent timed sequential patterns. This dynamic adjustment results in an increase in the number of newly discovered frequent patterns.

For example, in a synthetic dataset with 5,000 sequences, the number of patterns discovered was 3,758, while in a database with 20,000 sequences, the number of patterns increased to 3,780.

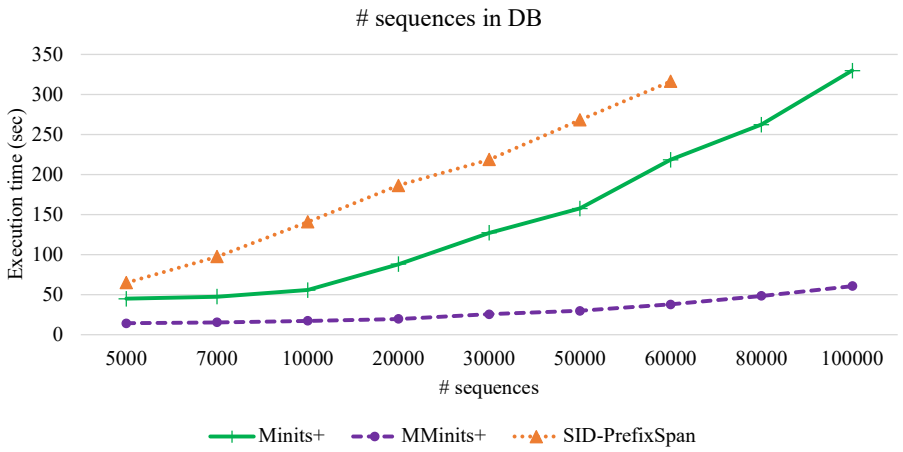


Figure 4.16: Impact of # sequences on execution time (ET) using Synthetic dataset

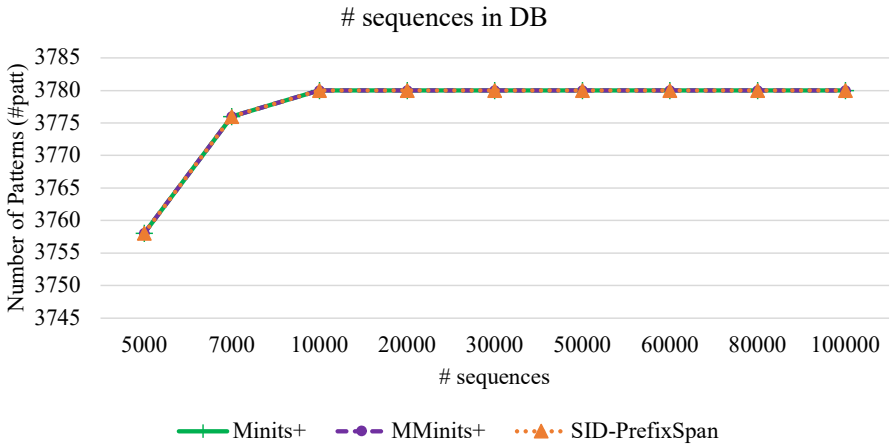


Figure 4.17: Impact of # sequences on number of patterns (#patt) using Synthetic dataset

4.2.3 Experimental Analysis of Minits-AllOcc

In this section, we describe the experimental environment and present the evaluation results of testing the following algorithms: single-core CPUs algorithm Minits-AllOcc, and multi-core CPUs MMinits-AllOcc. The experiments were conducted on both real and synthetic datasets, considering various parameters. After extensive testing, we observed that MMinits-AllOcc outperformed Minits-AllOcc.

4.2.3.1 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 were implemented in Java 1.8.

4.2.3.1.1. Datasets and Experimental Parameters

We use two real-life, T-Drive [15], [16] , and Oklahoma Mesonet [21], [22] , 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 4.12. 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

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 4.18: Sequential database for the Oklahoma Mesonet dataset before discretization

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 4.19: Sequential database for the Oklahoma Mesonet dataset after discretization

data, we discretized the data first by using a density-based clustering algorithm called Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [1], and the results of the discretized sequences are shown in Figure 4.13. 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. [18]. The second real data set is from Oklahoma Mesonet, a world-class network of environmental interventions by a group of scientists from the University of Oklahoma (OU) 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.

The dataset contains the following attributes: county ID, timestamp, air temperature, rainfall, wind, and moisture-humidity, as shown in Figure 4.18. We discretized the data first by using well-known scales in meteorology.

For air temperature, the index heat [23] 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 [24] 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 [25]: 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 4.19.

The synthetic dataset was generated by using a tool provided by the SPMF Library [17]. Also, we set several parameters to conduct the experiments on the dataset. There are two types of parameters: static and dynamic. The values of the static parameters were not changed in experiments. In contrast, the values of the dynamic parameters were changed from one experiment to another. In this experiment, we had four dynamic parameters. The first was the minimum support threshold (min_sup). It is a user-defined threshold that applies to finding all timed sequential patterns in a Discretized Timed Sequence Database (DTSDB). The second was the

number of timed sequences TS in DTSDB (#seq), which refers to the number of tuples in the database. The third parameter was the length of TS in DTSDB, which can also be represented as the number of events per TS (#events). The last parameter was 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 studied the effects of all four parameters shown in Table 4.7 on the synthetic dataset. However, for the T-Drive dataset, the only valid dynamic parameter shown in Table 4.7 is the min_sup. Thus, the other three parameters are all static. Now, we explain the range of the parameters and the default values of this analysis, as summarized in Table 4.7. 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 had a range of 20% to 80% with a default value of 50%, which is the median of the interval. The range of the number of timed sequences parameters was from 1 to 100,000, and its median value of 50,000 was the default value. For the number of events per sequence, the default value was 25 because the range was from 5 to 50. The number of items in the last parameter range was set at 1 to 10 items per event; thus, the default value was 5, which was the median.

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 4.7: Parameter list for the synthetic dataset

4.2.3.1.2 Evaluation Metrics

The evaluation metrics include two measurements: (1) execution time (ET) of algorithms (Minits-AllOcc and MMinits-AllOcc) and (2) number of patterns (#patt) that are generated by these algorithms.

4.2.3.2 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 (#patt) for the real and synthetic datasets.

4.2.3.2.1 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 [20]. 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 generated by Minits-AllOcc. Next, these patterns were compared to the patterns 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. If the order of at least one item set was different, pattern X did not match pattern Y. For instance, Z = < {b} [2, 5] {a} [3, 7] {b, a} > did not match 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.

4.2.3.2.2 Execution Time

The execution time was recorded from the moment that a dataset was read to the moment that an algorithm produced the timed sequential patterns. Table 4.8 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, compared to the execution time of Minits-AllOcc.

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

Table 4.8: Average execution time (ET) and #patt

4.2.3.2.3 Impact of Minimum Support

In this set of experiments, we compared execution time (ET) and the number of patterns (#patt) for different values of minimum support threshold (min_sup) for datasets Oklahoma Mesonet, T-Drive, and synthetic. From Figure 4.20, Figure 4.21, and Figure 4.22, we can see that when the minimum support increased, the execution time of all algorithms decreased. his is because the algorithms generate fewer timed sequential patterns when min_sup is high, as there

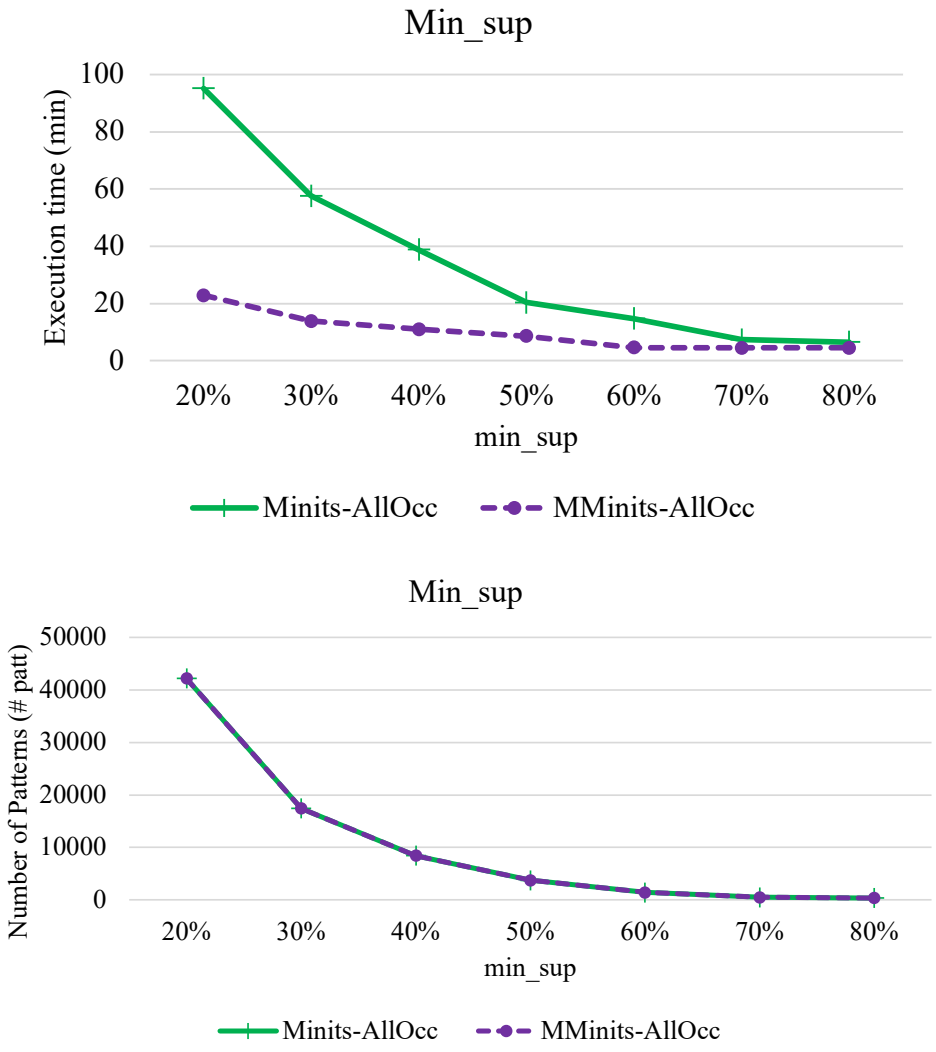


Figure 4.20: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Oklahoma Mesonet dataset

are 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 4.20, Figure 4.21, and Figure 4.22. Therefore, multi-core CPUs should be used when the size of the Discretized Timed Sequence Database is large.

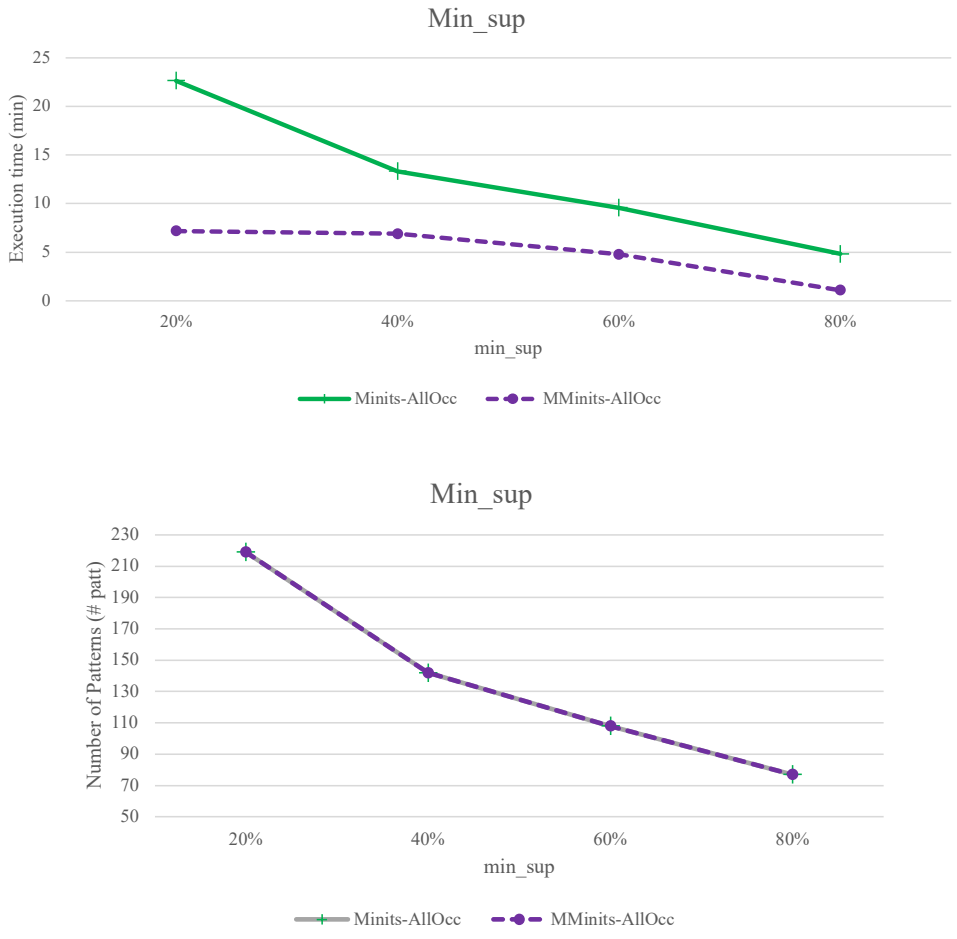


Figure 4.21: Impact of min_sup on execution time (ET) and number of patterns (#patt) using T-Drive dataset

As shown in Figure 4.20, Figure 4.21, and Figure 4.22, the ETs of both Minits-AllOcc, and MMinits-AllOcc were very close when the min_sup was greater than 60%. This is because the number of candidate sequences, and thus the number of timed sequential patterns, was decreasing, 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. In Figure 4.20, MMinits-AllOcc outperformed Minits-AllOcc in terms of execution time, achieving a 50% improvement.

In Figure 4.21, MMinits-AllOcc outperformed Minits-AllOcc in terms of execution time, achieving a 51% improvement. In Figure 4.22, MMinits-AllOcc outperformed Minits-AllOcc in terms of execution time, achieving a 55% improvement.

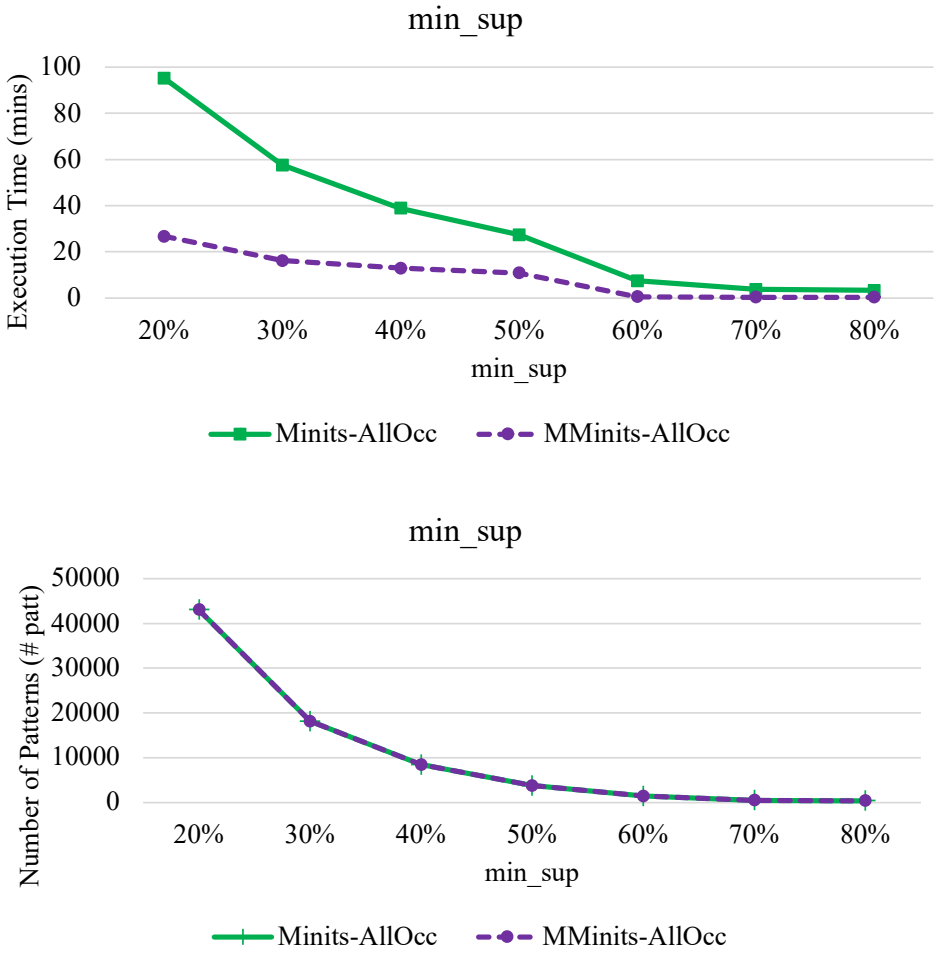


Figure 4.22: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Synthetic dataset

Another observation was made based on the number of timed sequential patterns generated by these algorithms. All algorithms discovered the same number of patterns; thus, their curves overlap in Figure 4.20, Figure 4.21, and Figure 4.22. 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 Discretized Timed Sequence Database that was supposed to contain a candidate sequence decreased, as shown in Figure 4.20, Figure 4.21, and Figure 4.22.

4.2.3.2.4 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 (#patt) according to the number of the timed sequences (#seq). From Figure 4.23, 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 Discretized Timed Sequence Database to decide if they contained a timed sequential pattern or not. In Figure 4.23, MMinits-AllOcc outperformed Minits-AllOcc in terms of execution time, achieving a 65% improvement.

We observed that the number of timed sequential patterns generated by these algorithms increased when the number of timed sequences increased, as shown in Figure 4.23. The number of timed sequential patterns discovered by the algorithms also increased because the possibility of finding more patterns in the new timed sequences that satisfied 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 whether 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 Discretized Timed Sequence Database, these patterns became timed sequential

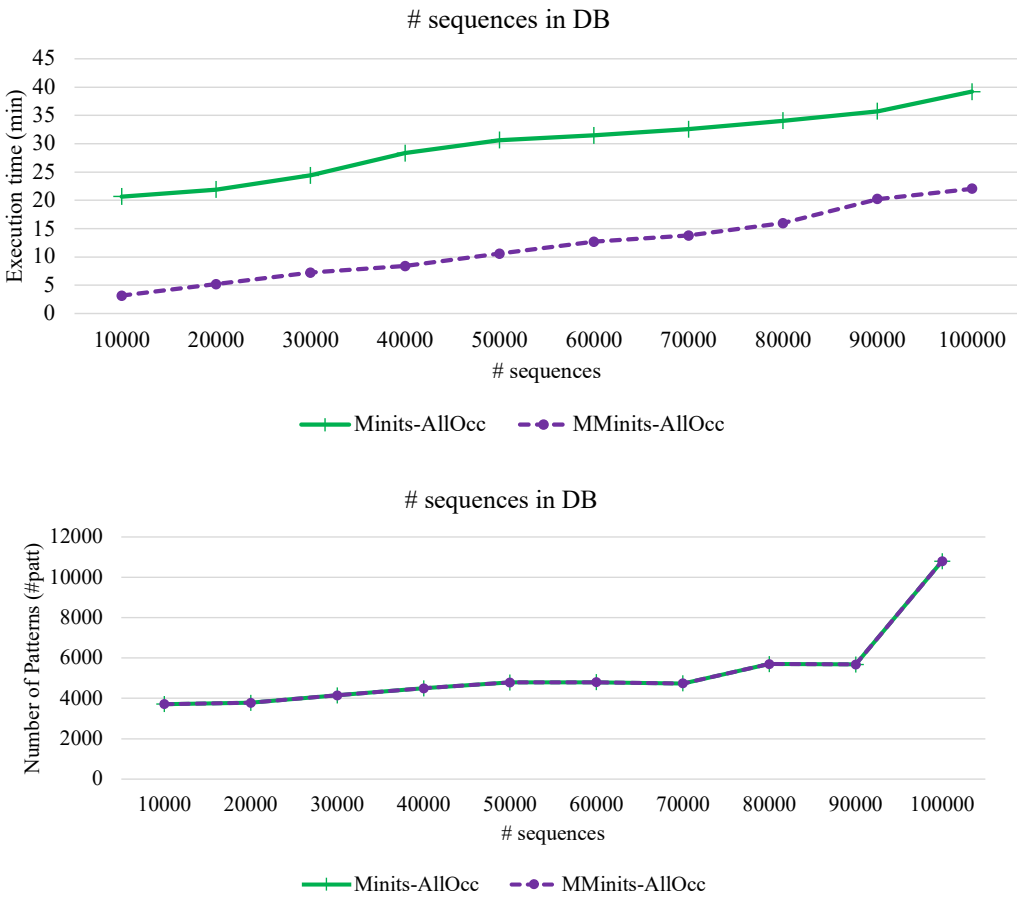


Figure 4.23: Impact of number of sequences on execution time (ET) and number of patterns (#patt) using Synthetic dataset

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 Discretized Timed Sequence Database had 10,000 timed sequences.

4.2.3.2.5 Impact of the Number of Events per Sequence

Figure 4.24 illustrates the impact of the number of events (#events) per timed sequence on execution time (ET) and the number of discovered sequential patterns (#patt). In Figure 4.24,

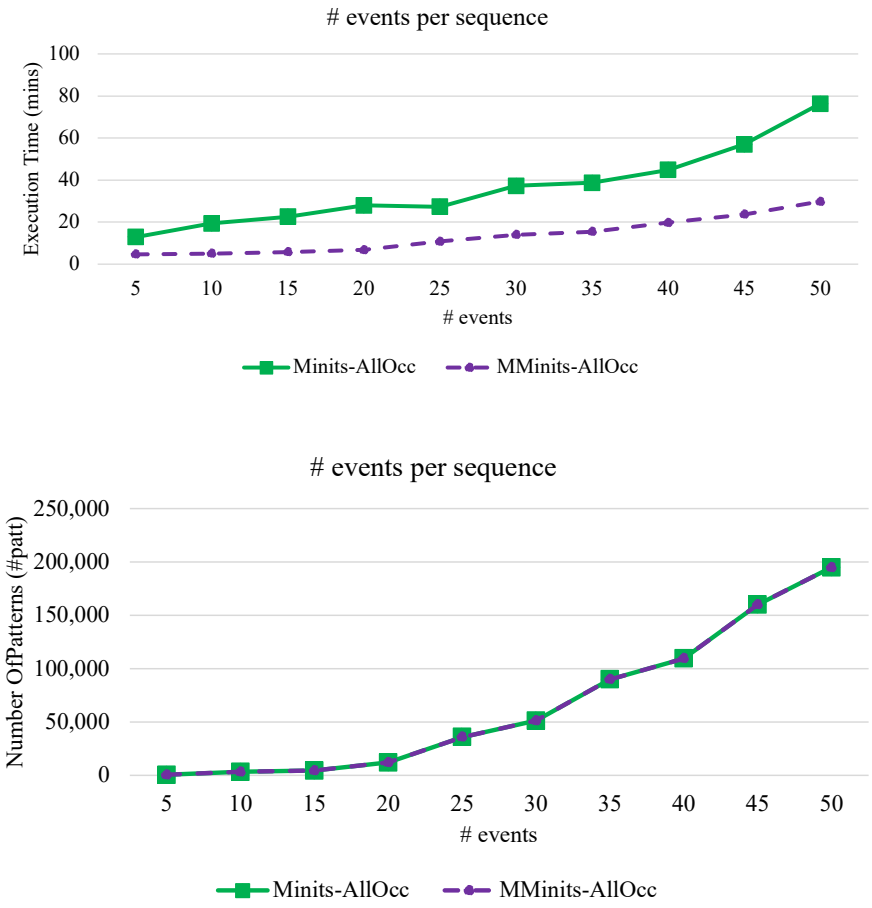


Figure 4.24: Impact of the number of events on execution time (ET) and number of patterns (#patt) using Synthetic dataset

MMinits-AllOcc outperformed Minits-AllOcc in terms of execution time, achieving a 55% improvement.

A strong relationship was observed between the length of a timed sequence and the number of discovered patterns. Increasing the length of timed sequences (#events) resulted in the discovery of more patterns, as the algorithm could extend a pattern up to the length of the timed sequence. If the maximum timed sequence containing n events, a set of timed sequential patterns can be

discovered with lengths varying from 1 to n. Consequently, the time required to discover these patterns increases, as shown in Figure 4.24.

4.2.3.2.6 Impact of the Number of Items per Event

In the last experiment, we increased the number of unique items in each event. This resulted in many new items appearing in the Discretized Timed Sequence Database (DTSDB), which in turn led to the detection of new timed sequential patterns. As the number of items increased, the number of possible combinations among these items to generate candidates also grew. Consequently, the number of patterns increased, as shown in Figure 4.25.

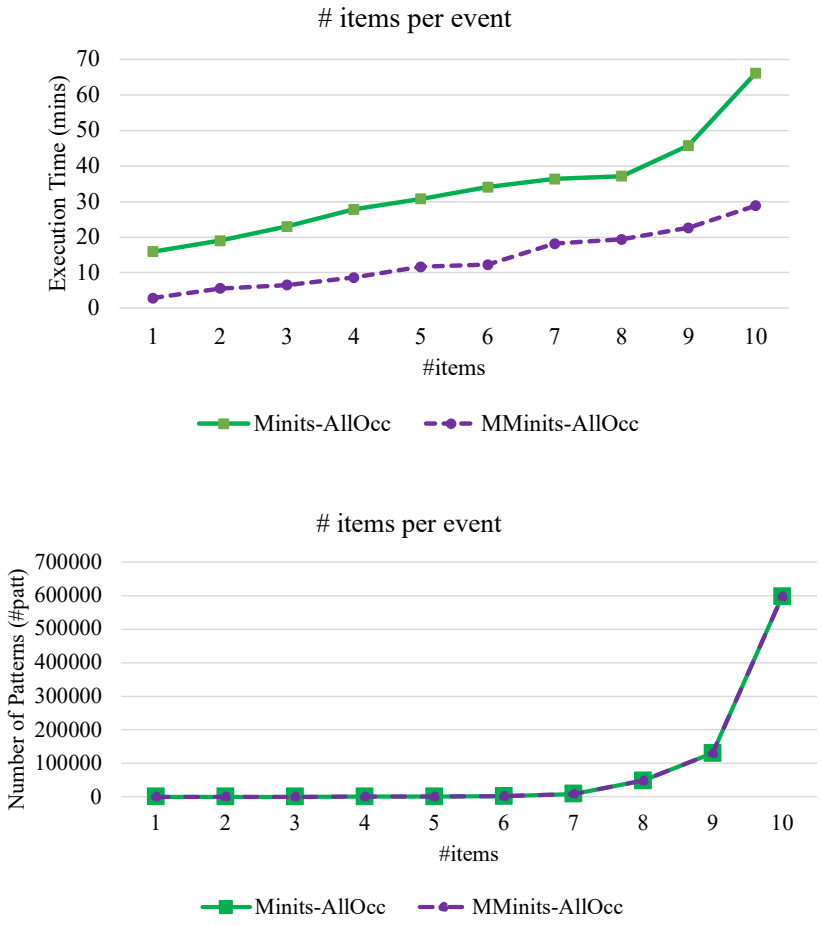


Figure 4.25: Impact of the number of items on execution time (ET) and number of patterns (#patt) using Synthetic dataset

Expanding the length of events further contributed to the growth in the number of candidates. This meant that the algorithms required more time, as illustrated in Figure 4.25, to evaluate these events, generate candidates, determine if they were timed sequential patterns, and report their temporal relationships. MMinits-AllOcc outperformed Minits-AllOcc in terms of execution time, achieving a 53% improvement.

4.2.4 Experimental Analysis of MinitsDays

In this section, we describe the experimental environment and present the evaluation results of testing the algorithms implemented on single-core CPUs (MinitsDays) and multi-core CPUs (MMinitsDays). The experiments were conducted on both real and synthetic datasets, considering various parameters. After extensive testing, we observed the following: First, MMinitsDays on a multi-core CPU consistently outperformed MinitsDays on a single-core CPU. Second, MinitsDays outperformed Minits-AllOcc, and, as expected, MMinitsDays outperformed MMinits-AllOcc.

4.2.4.1 Experimental Setup

All experiments were performed on a computer with a 3.20 GHz Intel(R) Core (TM) i7-8700 CPU processor running Windows 64-bit. The MinitsDays and MMinits-Days algorithms were implemented in Java 1.8.

4.2.4.1.1 Datasets and Experimental Parameters

We use Oklahoma Mesonet [21, 22], and synthetic datasets. 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 (OU) 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.

The dataset contains the following attributes: county ID, timestamp, air temperature, rainfall, wind, and moisture-humidity, as shown in Figure 4.18. We discretized the data first by using well-known scales in meteorology.

For air temperature, the index heat [43] 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 [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 [25]: 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 4.19.

The synthetic dataset was generated by using a tool provided by the SPMF Library [17]. Also, we set several parameters to conduct the experiments on the dataset. There are two types of parameters: static and dynamic. The values of the static parameters were not changed in experiments. In contrast, the values of the dynamic parameters were changed from one experiment to another. In this experiment, we had four dynamic parameters. The first is the minimum support threshold (min_sup). It is a user-defined threshold that applies to finding all timed sequential patterns in a Discretized Timed Sequence Database DTSDB. The second parameter is the number

of timed sequences TS in DTSDB (#seq), which refers to the number of tuples in the database. The third parameter is the length of TS in DTSDB, 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 was 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 studied the effects of all four parameters shown in Table 4.7 on the synthetic dataset. However, for the Oklahoma Mesonet dataset, the only valid dynamic parameter that is shown in Table 4.7 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 4.7. 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 had a range of 20% to 80% with a default value of 50%, which was the median of the interval. The range of the number of timed sequences parameters was from 1 to 100,000, and its median value of 50,000, was the default value. For the number of events per sequence, the default value was 25 because the range was from 5 to 50. The number of items in the last parameter range was set at 1 to 10 items per event; thus, the default value is 5, which was the median.

4.2.4.1.2 Evaluation Metrics

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

4.2.4.2 Experimental Results

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

4.2.4.2.1 Accuracy

To validate that MinitsDays consistently produces the same sequential patterns in terms of both their count and content (excluding the temporal relation), PrefixSpan was used [26]. PrefixSpan was selected as it is one of the most well-known algorithms for discovering sequential patterns and has been proven to generate complete and correct sequential patterns. First, all temporal relations were removed from the patterns generated by MinitsDays for comparison.

Next, these patterns were compared to the patterns generated by PrefixSpan to make sure that each sequential pattern generated by PrefixSpan had a corresponding 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 MinitsDays and MMinitsDays. We took away the temporal relations from Y and compared them with pattern X. If the order of at least one item set was different, pattern X did not match pattern Y. For instance, Z = < {b} [2, 5] {a} [3, 7] {b, a} > did not match pattern X because

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 4.9: Average execution time (ET) and #patt

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, MinitsDays and MMinitsDays, discovered the exact patterns that were produced by PrefixSpan. All algorithms produced the complete and correct set of sequential patterns.

4.2.4.2.2 Execution Time

The execution time was measured from the moment a dataset was read to the moment an algorithm produced the timed sequential patterns. Table 4.9 presents the average performance of the two algorithms, MinitsDays and MMinitsDays. The execution time (ET) of MMinitsDays decreased by 50% for the Oklahoma Mesonet dataset and by 90% for the synthetic datasets, compared to the execution time of MinitsDays.

4.2.4.2.3 Impact of Miniumum Support

In this set of experiments, we compared the execution time (ET) and the number of patterns (#patt) for different values of the minimum support threshold (min_sup) using the Oklahoma Mesonet and synthetic datasets. From Figure 4.26 and Figure 4.27, we observe that as the minimum support increased, the execution time of all algorithms decreased. This occurs because higher min_sup values result in fewer timed sequential patterns being generated, as there are fewer candidate sequences that satisfy the min_sup condition.

With a large amount of data and a significant number of discovered timed sequential patterns, MMinitsDays outperformed MMinits-AllOcc. Consequently, multi-core CPUs are recommended when working with large Discretized Timed Sequence Databases to improve performance.

The multi-core CPU version was also efficient when we had low min_sup. As shown in Figure 4.26 and Figure 4.27, the ETs of both MinitsDays and MMinitsDays were very close when the min_sup was greater than 60%. This is because the number of candidate sequences, and thus the number of timed sequential patterns, was decreasing, so most of the threads were idle. Therefore, MMinitsDays did not need to use all the available threads and behaved almost like a single-core version MinitsDays.

In Figure 4.26, Minits-Days outperformed Minits-AllOcc in terms of execution time, achieving a 52% improvement. Additionally, MMinitsDays outperformed MinitsDays, demonstrating a 28% improvement in execution time. In Figure 4.27, Minits-Days outperformed Minits-AllOcc in terms of execution time, achieving a 51% improvement. Additionally, MMinitsDays outperformed MinitsDays, demonstrating a 21% improvement in execution time.

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 Figure 4.26 and Figure 4.27. 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 Discretized Timed Sequence Database that was supposed to contain a candidate sequence decreased, as shown in Figure 4.26 and Figure 4.27.

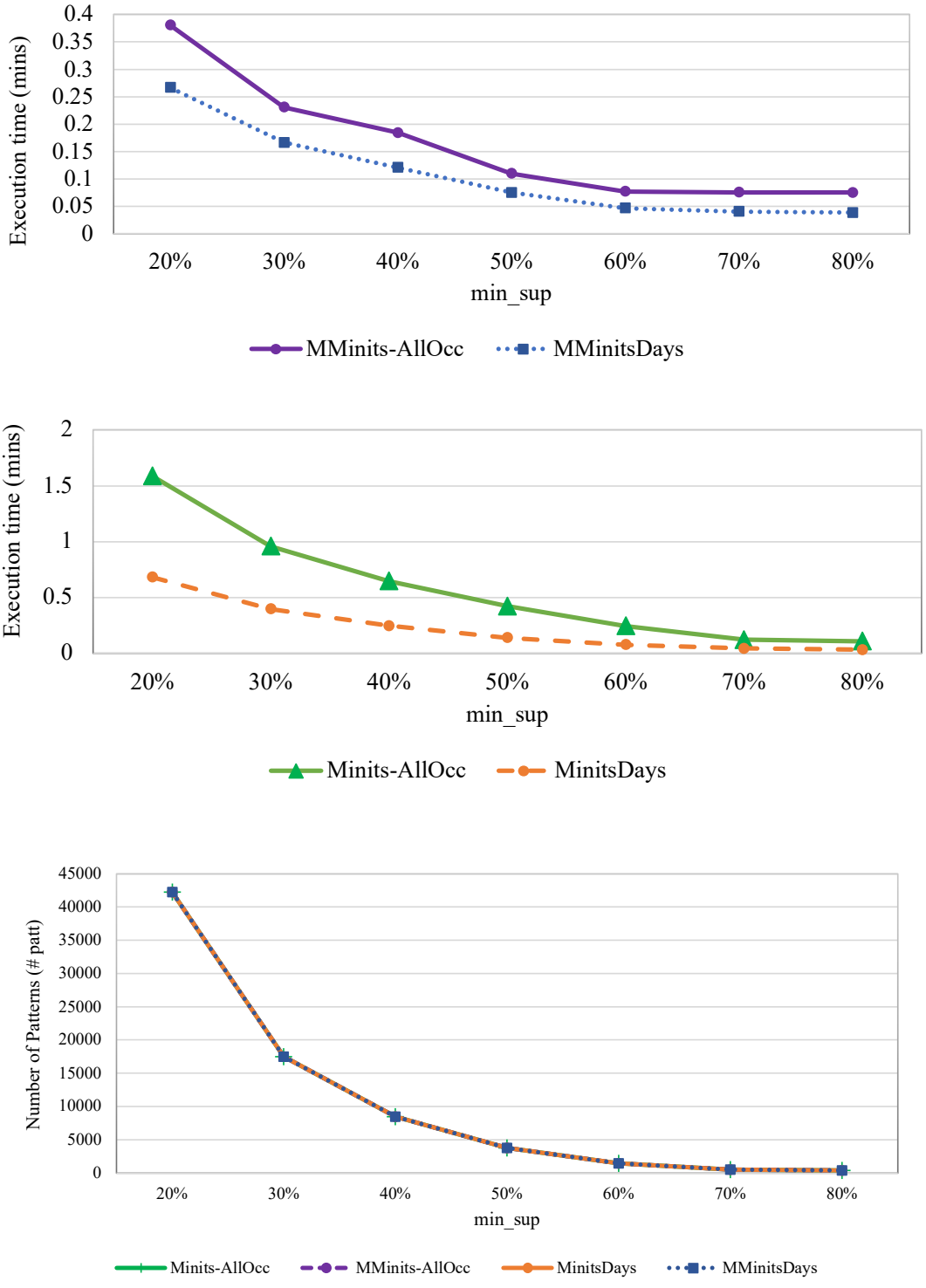


Figure 4.26: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Oklahoma Mesonet dataset

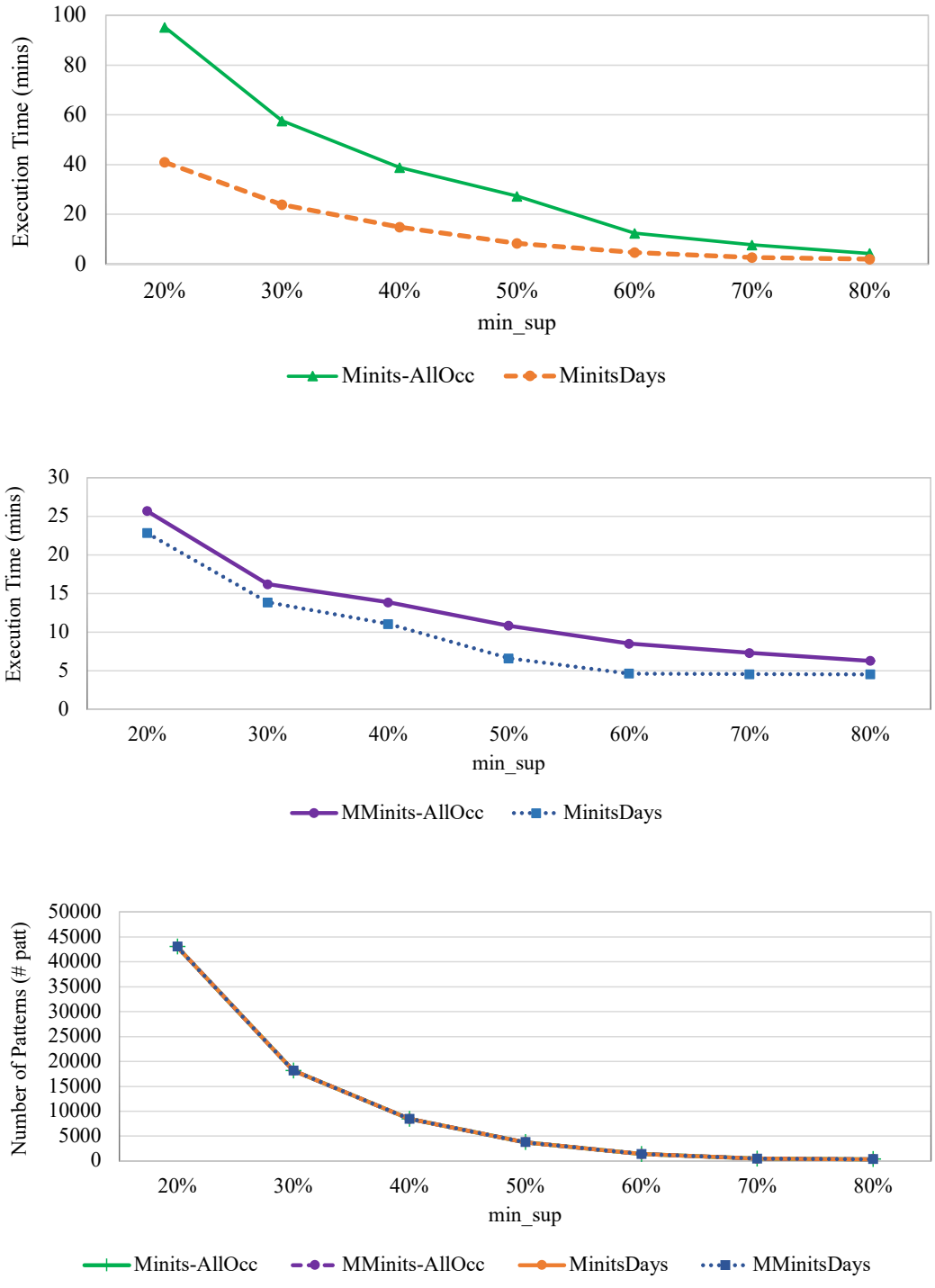


Figure 4.27: Impact of min_sup on execution time (ET) and number of patterns (#patt) using Synthetic dataset

4.2.4.2.4 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 (#patt) according to the number of the timed sequences (#seq). From Figure 4.28, 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 Discretized Timed Sequence Database to determine whether they contained a timed sequential pattern or not. We observed that the number of timed sequential patterns generated by these algorithms increased when the number of timed sequences increased, as shown in Figure 4.28. MinitsDays outperformed Minits-AllOcc in terms of execution time, achieving a 42% improvement. Additionally, MMinitsDays outperformed MMinits-AllOcc, demonstrating a 86% improvement in execution time.

The number of timed sequential patterns discovered by the algorithms also increased because the possibility of finding more patterns in the new timed sequences that satisfied 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 whether 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 were added did not satisfy the min_sup condition because they were not supported by enough timed sequences; however, with a new Discretized 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 Discretized Timed Sequence Database had 10,000 timed sequences.

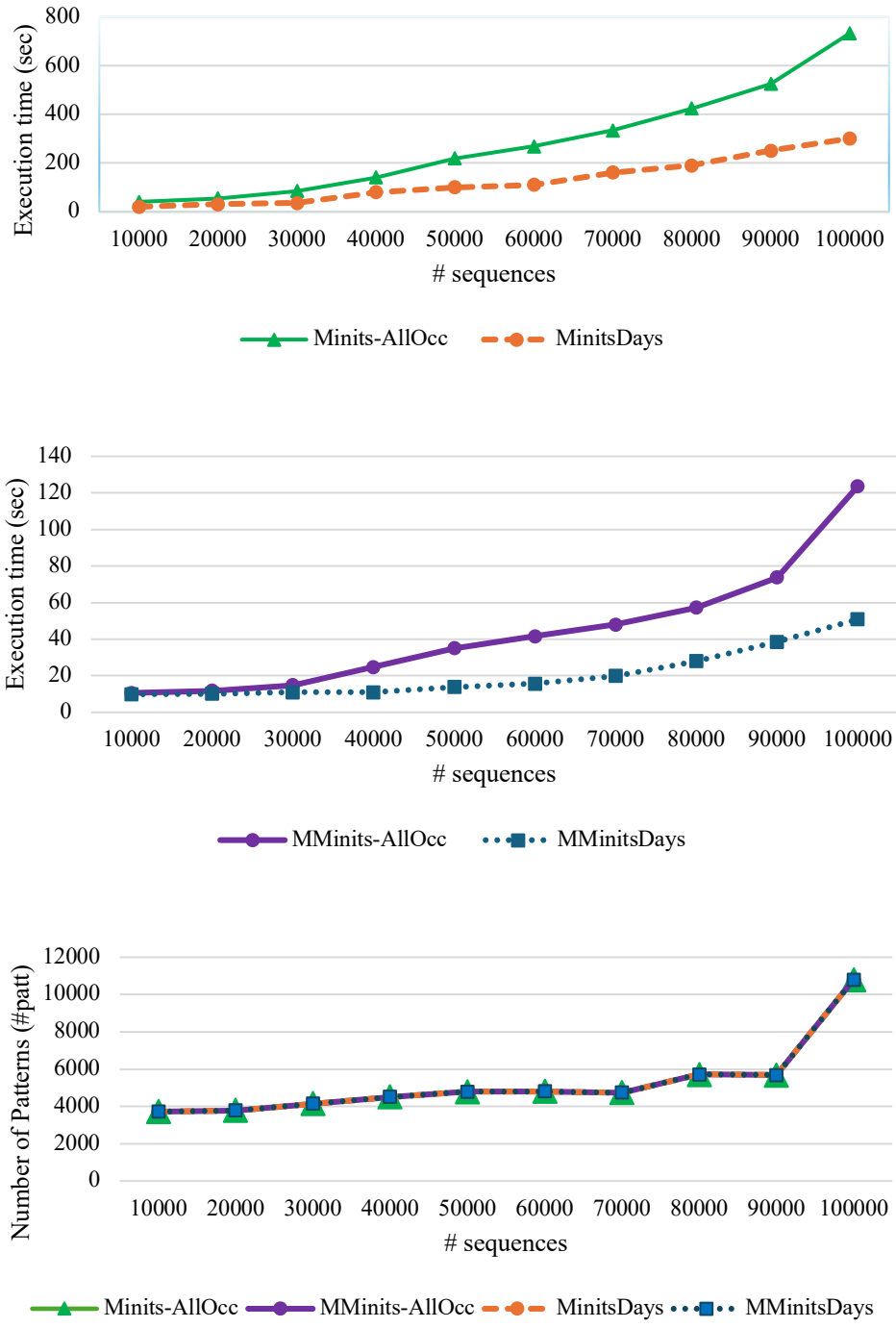


Figure 4.28: Impact of number of sequences on execution time (ET) and number of patterns (#patt) using Synthetic dataset

4.2.4.2.5 Impact of the Number of Events per Sequence

Figure 4.29 shows the impact of the number of events (#events) per timed sequence on the execution time (ET) and the number of discovered sequential patterns (#patt). MinitsDays outperformed Minits-AllOcc in terms of execution time, achieving a 19% improvement. Additionally, MMinitsDays outperformed MMinitsDays, demonstrating a 41% improvement in execution time. 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 with a length varying from 1 to n. Subsequently, the time required to discover those patterns will increase as shown in Figure 4.29.

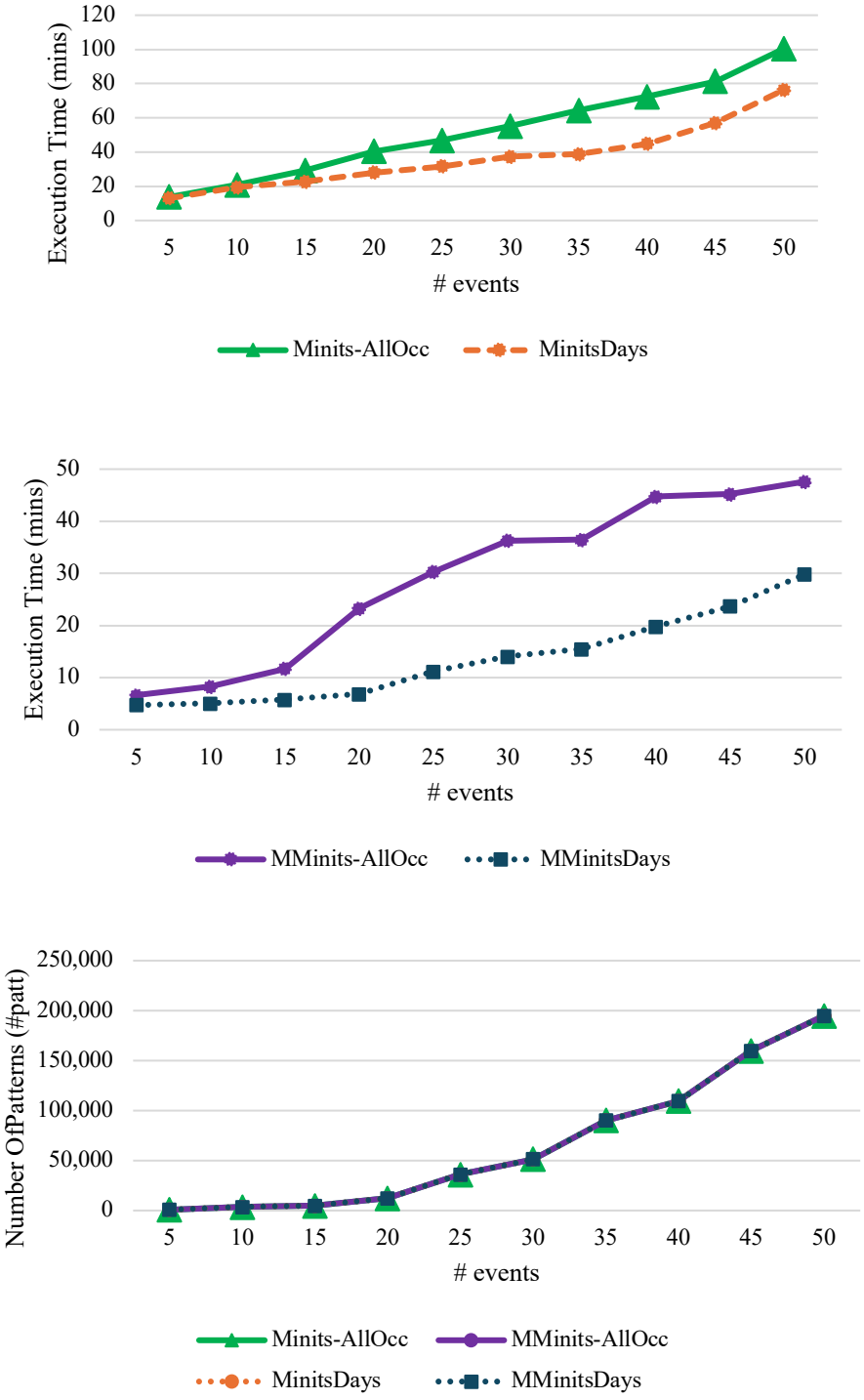


Figure 4.29: Impact of the number of events on execution time (ET) and number of patterns (#patt) using Synthetic dataset

4.2.4.2.6 Impact of the Modification Ratio

The modification ratio refers to the rate of modifications (changes) made to an existing database by updating, inserting, or deleting timed sequences. In the last experiment, we compared execution time (ET) and the number of patterns (#patt) for different modification ratios. Figure 4.30 shows the results of both the insertion and deletion ratios. When the Figures are read from left to right, the results of the insertion ratio can be seen; when read from right to left, the results of the deletion ratio are visible. From these Figures, we can see that when the percentage of the insertion ration is increased, the execution time is also increased. The algorithms required additional time to examine the additional timed sequences in the Discretized Timed Sequence Database to determine whether they had new sequential patterns. As shown in Figure 4.30, we found that as the number of timed sequences increased, so did the number of timed sequential patterns produced by these algorithms. The likelihood of discovering more timed sequential patterns that satisfied the min_sup (50% as the default value) when the insertion ratio is increased also rose. This led to an increase in the number of timed sequential patterns found by the algorithms.

The algorithms must determine whether new patterns not present in the previous timed sequences could emerge due to the growing quantity of timed sequences in the database. The algorithm then compared their support against the threshold (min_sup). The support of some old patterns in the database before new sequences were added did not satisfy the min_sup condition because they lacked the support of enough timed sequences. Still, with a new Discretized Timed Sequence Database, these patterns became timed sequential patterns. As a result, the quantity of recently found timed sequential patterns would rise.

For the deletion ratio case, ultimately, the opposite scenario to the insertion ratio case was seen. The execution time and the number of timed sequential patterns decreased because the possibility of finding more patterns in the new timed sequences that satisfied the min_sup (50% as the default value) condition was also reduced.

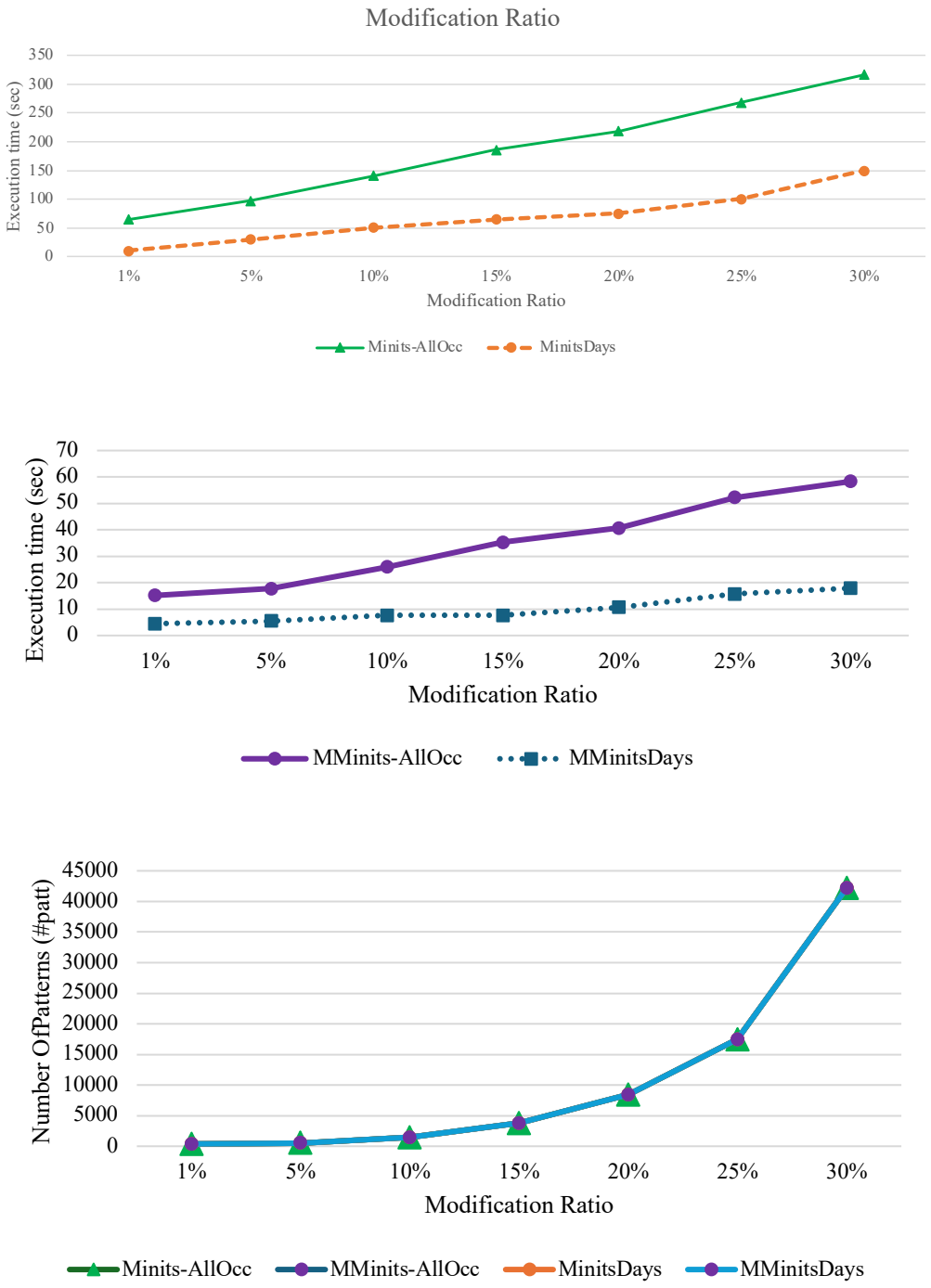


Figure 4.30: Impact of the modification ratio on execution time (ET) and number of patterns (#patt) using Synthetic dataset

4.3 Summary

This chapter presents an in-depth analysis and evaluation of three proposed algorithms—Minits+, Minits-AllOcc, and MinitsDays—for mining timed sequential patterns from static and dynamic discretized timed sequence databases. Minits+ is designed for static databases using a pseudo-projection technique to optimize performance, while Minits-AllOcc improves upon this by considering all occurrences of patterns using occurrence trees. MinitsDays extends the capability to dynamic databases, efficiently handling updates without re-mining the entire database. Experimental evaluations using real-world and synthetic datasets revealed that the multi-core versions of Minits+ (MMinits+), Minits-AllOcc (MMinits-AllOcc) and MinitsDays (MMinitsDays) significantly reduce execution time compared to single-core implementations, particularly for large datasets and low support thresholds. The chapter also highlights the impact of parameters like minimum support, number of sequences, and database updates on algorithm performance, demonstrating the scalability and efficiency of the proposed methods in diverse data scenarios.

CHAPTER 5
CONCLUSIONS AND FUTURE WORK

This dissertation proposes three algorithms for discovering timed sequential patterns: Minits+, Minits-AllOcc, and MinitsDays.

The first algorithm, Minits+, is designed to identify timed sequential patterns in a static sequence database. These patterns not only highlight the itemsets that frequently occur in sequence but also indicate the time required to transition from one itemset to another. Such patterns have applications in various domains, including recommendation systems for transportation, healthcare, and weather forecasting. Minits+ operates by first verifying whether a pattern appears in each timed sequence and then calculating its support to determine if it satisfies the minimum threshold condition. If the pattern is frequent, the temporal relationship between the itemsets is subsequently calculated.

The second algorithm, Minits-AllOcc, is designed to discover timed sequential patterns in a static discretized timed sequence database. Unlike the first algorithm, this method not only tracks whether a pattern occurs in a timed sequence but also records for the number of times the pattern occurs within a specific timed sequence. Once the pattern satisfies the minimum threshold condition, the temporal relationship between the itemsets is calculated.

The third algorithm, MinitsDays, is designed to discover timed sequential patterns in a dynamic discretized timed sequence database. In real-world scenarios, the content of a sequence database frequently changes due to data insertions, deletions, or modifications. This algorithm efficiently updates the set of timed sequential patterns in response to these changes, without the need to rerun the algorithm from scratch.

We conducted a complexity analysis to evaluate the worst-case execution time for all our proposed algorithms: Minits+, Minits-AllOcc, and MinitsDays. Experimental evaluations are performed using both real and synthetic datasets to compare the performance of these algorithms in terms of execution time and the number of discovered patterns.

5.1 Summary of the Performance Evaluation Results

In this section, we present the summaries of the experimental performance results of our three proposed algorithms: Minits+, Minits-AllOcc, and MinitsDays.

5.1.1 Summary of the Results of Minits+

Sequential pattern mining aims to find a set of frequently repeating sequential patterns that occur in databases of ordered sequences. This problem was first introduced in [1]. With the sequential patterns, we can answer questions such as in which order the symptoms of a heart attack frequently occur. For example, patients tend to have high blood pressure, which is followed by low temperature. Timestamps are used to order events within a sequential pattern, but the temporal relation between itemsets is not defined. In many applications, it is important to know the time-range, as in the real-world, for moving from one itemset to another. With timed sequential patterns, we can determine the order in which heart attack symptoms typically occur and their temporal progression.

In this dissertation, we have proposed an algorithm called Minits+ for mining timed sequential patterns from static sequence databases. This algorithm is based on the idea that after the discovery of a sequential pattern, the temporal relation between successive itemsets is calculated. Minits+ is summarized as follows:

- Minits+ is designed to determine the temporal relationship between itemsets in a frequent sequential pattern.
- To find the complete set of the timed sequential patterns, the following steps are performed in our algorithm:

1. Read the following from a user: min_sup, Discretized Timed Sequence Database, User's required statistics in the timed sequential patterns.
2. Scan timed sequence DB to determine distinct items.
3. For each distinct Item I_j, compute its support. If I_j is frequent, add it to the set of Frequent Items, denoted as FI and the set of timed sequential patterns TSP.
4. For each frequent item I_j, build the pseudo-projected database, which is a projection of the original database. This projection collects all pointers referring to the sequences in the main memory as a pseudo-projection. Building projected databases effectively reduces their size, significantly minimizing the effort required for generating candidate subsequences.
5. Compute the occurrences of event-relation and sequence-relation between I_i and each frequent item in FI.
6. Append f_K, an element from FI, to I_j considering the type of relation (event or sequence relation), and add the pattern into TSP after compute the statistics. If it is sequence relation and the pattern is frequent, then the user required statistic is computed.
7. Update FI to include the new frequent items in pseudo-projected database.
8. Repeat steps 2, 3, 4, and 5 until no new timed sequential patterns (TSP) can be discovered.

- Minits+ the uses a pseudo-projection technique inspired by PrefixSpan. The pseudo-projection database contains a pointer to a sequence in the original database, an offset

indicating the starting position of the postfix within the sequence, and the timestamp of the event that contains the prefix.

- A parallel version of Minits+ has been implemented by using multi-core CPUs called MMinits+ to deal with volume characteristic of large discretized timed sequence databases.
- Minits+ has a time complexity of $O(M^L*N*L)$, where M is the number of distinct items, N represents the number of timed sequences in the discretized timed sequence database (DTSDB), and L represents the maximum number of events per sequence in the database.
- Minits+ is 100% accurate, producing the same patterns (excluding temporal relations) in terms of quantity and content as PrefixSpan.
- In terms of execution time, the MMinits+ version reduces by about 74% compared to Minits+. This result was observed using real and synthetic datasets.
- Our experiments show that when the minimum support threshold increased, the execution time of both algorithms (Minits+ and MMinits+) decreased. Additionally, the number of discovered patterns decreased because fewer patterns satisfied the minimum support condition.
- Our experiments show that when the number of sequences increased, the execution times of both algorithms increased and the number of patterns that were discovered by both algorithms increased because the possibility of finding more patterns in new timed sequences that satisfy the min_sup condition increased.

5.1.2 Summary of the Results of Minits-AllOcc

Incorporating temporal information into sequential patterns presents additional challenges compared to ordinary sequential pattern mining. First, while both sequential pattern mining and timed sequential pattern mining are used to determine whether a pattern occurs in a sequence

database tuple, timed sequential pattern mining also determines how many times the pattern occurs in that tuple to compute the temporal relationship between the itemsets in the pattern. Therefore, to determine all possible occurrences of the pattern, a naïve approach must search each tuple in the database from the beginning to the end. Unlike sequential pattern mining, an algorithm will stop scanning the rest of the tuple in the database as soon as the pattern is identified. Once all positions of a pattern have been determined, the temporal data associated with it must be updated using the timestamps of all tuples that contain that pattern. To ensure the accuracy of the temporal relations, it is necessary to repeatedly scan the entire database. Naturally, this takes more time and space, which affects how well any algorithm performs. In this work, we propose a timed sequential pattern mining algorithm, Minits-AllOcc (MINIng Timed Sequential Pattern for All-time Occurrences), that addresses all these challenges. Minits-AllOcc is summarized as follows:

- Minits-AllOcc is an algorithm for finding the complete set of all occurrences of the timed sequential patterns that satisfy the min_sup threshold condition from a given static discretized timed sequence database.
- The following steps are performed by Minits-AllOcc:

1. Scan DTSDB to build an I_j-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 two relations between itemsets are considered while merging the steps *event-relation* and *sequence-relation.*
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 DTSDB 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.

- Minits-AllOcc uses efficient mechanisms to improve performance. First, the algorithm prunes forests by removing any branch in an O-tree that does not have a new appended node after the merging step is executed. Second, by avoiding the generation of unnecessary candidates and reducing the number of forests by using frequency matrix.
- A parallel version of Minits-AllOcc, called MMinits-AllOcc, was implemented using multi-core CPUs to handle the large-scale nature of discretized timed sequence databases.
- Minits-AllOcc has a time complexity of $O(M^L*N^2*L^2)$, where M represents the number of distinct items, N represents the number of timed sequences in the discretized timed sequence database (DTSDB), and L represents the maximum number of events per sequence in the database.
- Minits-AllOcc is 100% accurate, producing the same patterns, excluding temporal relations, as PrefixSpan.
- In terms of execution time, the MMinits-AllOcc version decreases by 50% to 60% compared to Minits-AllOcc. This result was observed using real and synthetic datasets.
- Our experiments show that when the minimum support threshold increased, the execution time of the algorithms, Minits-AllOcc and MMinits-AllOcc, decreased. This is because the algorithms generate fewer timed-sequential patterns when the min_sup is high. Also, the

multi-core CPUs version is efficient when the min_sup is low. In In addition, as min_sup increased, the number of timed sequential patterns decreased because fewer patterns met the threshold.

- Our experiments show that when the number of sequences increased, the execution times of all algorithms increased because the algorithms required more time to process additional sequences added to the database to determine whether they contained a timed sequential pattern. The likelihood of identifying additional patterns in new sequences increases, raising the number of discovered timed sequential patterns.
- Our experiments studied the impact of the number of events per timed sequence. We observed a positive correlation between the number of patterns found and execution time. More patterns were found when timed sequences were longer because the algorithm extended a pattern up to the length of the longest timed sequence. Thus, the amount of time needed to find those patterns also increased.
- Our experiments studied the impact of the number of distinct items per event. As the number of items increased, so did the possible combinations among them, leading to more candidate patterns. As event length grew, more candidate patterns were generated, increasing both the number of patterns and the required execution time.

5.1.3 Summary of the Results of MinitsDays

Most sequential pattern mining algorithms, as well as the two earlier algorithms, Minits+ and Minits-AllOcc, operate under the assumption that databases are static and do not undergo changes over time. In practice, real-world databases frequently change in response to a variety of circumstances, including user interactions, changing needs, and data updates. Therefore, to avoid necessitating a re-mining of the entire discretized timed sequence database, we need to design an

algorithm that can respond to these changes to identify the full updated set of timed sequential patterns. In this dissertation, we propose an algorithm for MINIng Timed Sequential patterns from DYnamic timed Sequence database, called MinitsDays. MinitsDays is summarized as follows:

- MinitsDays is an algorithm for MINIng all possible Occurrences of Timed Sequential patterns from a DYnamic discretized timed sequence database.
- The following steps are performed by MinitsDays:

1. Determine the type of changes: Identify whether the Update_Type is an insertion or deletion operation.
2. Check the existence of data structures: If the required data structures have not been created, this indicates the first execution of the algorithm, and they must be initialized.
3. Read the list of updated timed sequences (Updated_TS_List):
    a. If the operation is an insertion, the *handle_insertion()* function is called.
    b. If the operation is a deletion, the *handle_deletion()* function is called.
4. Update the content of the data structures based on the type of changes:
    a. If a new distinct item appears due to the addition of new timed sequences, it is added to the 1-Sequence data structure.
    b. If an existing distinct item becomes infrequent due to the deletion of some timed sequences, it is removed from the TSS-tree along with all its derived patterns.
5. Traverse the updated TSS-tree: The updated tree is traversed to provide the complete and updated set of timed sequential patterns.

- A parallel version of MinitsDays, called MMinitsDays, was implemented by using multi-core CPUs to deal with volume characteristic of large discretized timed sequence databases.

- The MinitsDays has a time complexity of O (M^L * N * L), where M represents the number of distinct items, N represents the number of timed sequences in the discretized timed sequence database (DTSDB), and L represents the maximum number of events per sequence in the database.
- MinitsDays is 100% accurate, producing the same number and content of patterns as those produced by PrefixSpan, excluding the temporal relation.
- The parallel version, MMinitsDays, reduces execution time by 50% to 90% compared to MinitsDays. This result was observed using real and synthetic datasets.
- Our experiments show a negative correlation between the rise in the minimum support threshold and the decrease in the execution time of both algorithms, MinitsDays and MMinitsDays. This occurs because fewer timed-sequential patterns meet the high minimum support threshold. The multi-core version is highly efficient when min_sup is low.
- Our experiments show a positive correlation between the number of timed sequences and the execution times of all algorithms. This is because the algorithms require additional time to evaluate the newly added timed sequences in the database to determine whether they include a timed sequential pattern. The probability of discovering further patterns in the new timed sequences also rose, increasing the algorithms' detection of timed sequential patterns.
- Our experiments studied the effects of varying the number of events in a timed sequence. There was a positive correlation between the execution duration and the number of patterns detected. Because the method may extend a pattern up to the length of the largest timed

sequence, more patterns were discovered when the timed sequences were longer. As a result, it took longer to identify those patterns.

- Our experiments showed that when the percentage of insertion ratio increased, the execution times of both algorithms increased, and the number of patterns discovered by both algorithms increased because the possibility of finding more patterns in new timed sequences that satisfied the min_sup condition increased.
- In contrast, when the percentage of deletion ratio decreased, the execution times of both algorithms decreased, and the number of patterns discovered by both algorithms decreased because the possibility of finding more patterns in new timed sequences that satisfied the min_sup condition decreased.

5.2 Future Research

In this section, we discuss potential future research directions related to our proposed algorithms: Minits+, Minits-AllOcc, and MinitsDays. While this work addresses the problem of mining timed sequential patterns and discovering them from a static or dynamic database, there remains significant scope for further exploration and improvement in future studies.

The first potential research direction involves developing methods to determine the optimal time for re-mining the database to identify the updated set of timed sequential patterns. In some cases, changes in the database content may have minimal impact on the final set of patterns, or the user may only be interested in approximate timed sequential patterns rather than the complete set.

A second potential research direction involves enhancing the proposed algorithms to address other complex practical applications, such as mining from data streams.

A third potential research direction involves designing a more scalable and parallelized version of MinitsDays that can leverage modern parallel computing environments, such as GPU and Apache Spark, to handle Big Data. This approach would address scenarios where not only the number of sequences and events but also the number of items is significantly large.

REFERENCES

[1] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the Eleventh International Conference on Data Engineering*, Taipei, Taiwan, Mar. 1995, pp. 3–14.

[2] N. Jay. G. Herengt, E. Albuisson, F. Kohler, and A. Napoli, "Sequential pattern mining and classification of patient path," *World Congress on Medical Informatics*, pp. 73–80, Jan. 2004.

[3] Y. W. T. Pramono, 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 *Proceedings of the 2014 International Conference of Advanced Informatics: Concept, Theory and Application*, Bandung, Indonesia, Aug. 2014, pp. 203–208.

[4] O. Dermy and A. Brun, "Can we take advantage of time-interval pattern mining to model students' activity?" in *Proceedings of the 13th International Conference on Educational Data Mining*, Online, Jul. 2020, pp. 69–80.

[5] R. Srikant and R. Agrawal, "Mining sequential patterns: Generalizations and performance improvements," in *Proceedings of the International Conference on Extending Database Technology*, Avignon, France, Mar. 1996, vol. 1057, pp. 1–17.

[6] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal and M.-C. Hsu, "PrefixSpan: Mining sequential patterns efficiently by Prefix-projected pattern growth," in *Proceedings of the 17th International Conference on Data Engineering*, Heidelberg, Germany, Apr. 2001, pp. 215–224.

[7] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal and M.-C. Hsu, "FreeSpan: Frequent pattern-projected sequential pattern mining," in *Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Boston, MA, USA, Aug. 2000, pp. 355–359.

[8] M. J. Zaki, "SPADE: An efficient algorithm for mining frequent sequences," *Machine Learning*, vol. 42, pp. 31–60, Jan. 2001.

[9] F. Giannotti, M. Nanni, F. Pinelli and D. Pedreschi, "Trajectory pattern mining," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Jose, CA, USA, Aug. 2007, pp. 330–339.

[10] Y.-L. Chen, M.-C. Chiang, M.-T. Ko, "Discovering time-interval sequential patterns in sequence databases," *Expert Systems with Applications*, vol. 25, no. 3, pp. 343–354, Oct. 2003.

[11] H. H. Le, T. Yamada, Y. Honda, M. Kayahara, M. Kushima, K. Araki and H. Yokota, "Effects of mining parameters on the performance of the sequence pattern variants analyzing method applied to electronic medical record systems," in *Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services*, Munich, Germany, Dec. 2019, pp. 127–135.

[12] H. H. Le, T. Yamada, Y. Honda, T. Sakamoto, R. Matsuo, T. Yamazaki, K. Araki and H. Yokota, "Methods for analyzing medical-order sequence variants in sequential pattern mining for electronic medical record systems," *ACM Transactions on Computing for Healthcare*, vol. 3, no. 1, pp. 1–28, Sep. 2023.

[13] D. D. Gajski and J.-K. Peir, "Essential issues in multiprocessor systems," *IEEE Computer*, vol. 18, no. 6, pp. 9–27, Jun. 1985.

[14] American Heart Association. "Understanding Blood Pressure Readings." Accessed: April 2, 2020. [Online.] Available: https://www.heart.org/en/health-topics/high-blood-pressure/understanding-blood-pressure-readings,https://www.heart.org/en/health-topics/high-blood-pressure/understanding-blood-pressure-readings

[15] National Hurricane Center. "Saffir-Simpson Hurricane Wind Scale." Accessed: April 1, 2024. [Online.] Available: https://www.nhc.noaa.gov/aboutsshws.php.

[16] M. A. Rossetti. "Analysis of weather events on US railroads," in *National Transportation Systems Center (U.S.)*, Online, Jan. 2007.

[17] T. Simes, "A blow to train operations: Can strong winds cause derailment?" in *Proceedings of the International Railway Safety Conference, Melbourne*, Australia, Online, Oct. 2011.

[18] P.-N. Tan, M. Steinback, and V. Kumar, *Introduction to Data Mining*, 1st ed., Uttar Pradesh, India: Pearson Education India, 2016.

[19] C. C. Aggarwal. *Data mining: the textbook*, 1st ed. Cham, New York: Springer, 2015.

[20] N. R. Mabroukeh and C. I. Ezeife, "A taxonomy of sequential pattern mining algorithms," *ACM Computing Surveys*, vol. 43, no. 1, pp.1–41, Dec. 2010.

[21] P. Fournier-Viger, A. Gomariz, M. Campos and R. Thomas, “Fast vertical mining of sequential patterns using co-occurrence information,” in *Proceedings of the 18th Pacific-Asia Conference, Advances in Knowledge Discovery and Data Mining*, Tainan, Taiwan, May. 2014, vol. 8443, pp. 40–52.

[22] J. Ayres, J. Flannick, J. Gehrke and T. Yiu, “Sequential PAttern mining using a Bitmap Representation,” in *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, Edmonton, AB, Canada, Jul. 2002, pp. 429–435.

[23] F. V. Brock, K. C. Crawford, R. L. Elliott, G. W. Cuperus, S. J. Stadler, H. L. Johnson and M. D. Eilts, “The Oklahoma Mesonet: A technical overview,” *American Meteorological Society*, vol. 12, no.1, pp.12: 5–19, Feb. 1995.

[24] R. A. McPherson et al., “Statewide monitoring of the mesoscale environment: A technical update on the Oklahoma Mesonet,” *American Meteorological Society*, vol. 24, no. 3, pp. 301–321, Mar. 2007.

[25] Y.-H. Hu, T. C.-K. Huang, H.-R. Yang and Y.-L. Chen, “On mining multi-time-interval sequential patterns,” *Data Knowledge Engineering*, vol. 68, no. 10, pp. 1112–1127, Oct. 2009.

[26] H. Yang, L. Gruenwald and M. Boulanger, “A novel real-time framework for extracting patterns from trajectory data streams,” in *Proceedings of the 4th ACM SIGSPATIAL International Workshop on GeoStreaming*, Orlando, FL, USA, Nov. 2012, pp. 26–32.

[27] T-P Hong, K-Y Lin and S-L Wang, “Mining fuzzy sequential patterns from multiple-item transactions,” in *Proceedings of the Joint 9th IFSA World Congress and 20th NAFIPS International Conference*, Vancouver, BC, Canada, Jul. 2001, pp. 457–462.

[28] C. F. Ahmed, S. K. Tanbeer and B.-Y. Jeong, “A novel approach for mining high-utility sequential patterns in sequence databases,” *Electronics and Telecommunications Research Institute Journal*, vol. 32, no. 5, pp. 676–686, Oct. 2010.

[29] S. K. Tanbeer, C. F. Ahmed, B. S. Jeong and Y. K. Lee, “Discovering periodic-frequent patterns in transactional databases,” in *Proceedings of the 13th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Bangkok, Thailand, Apr. 2009, vol. 5476, pp. 242–253.

[30] H. Cao, N. Mamoulis and D. W. Cheung, “Mining frequent spatio-temporal sequential patterns,” in *Proceedings of the Fifth IEEE International Conference on Data Mining*, Houston, TX, USA, Nov. 2005, pp. 1550–4786.

[31] F. Giannotti, M. Nanni and D. Pedreschi, “Efficient mining of temporally annotated sequences,” in *Proceedings of the 2006 SIAM International Conference on Data Mining*, Bethesda, MD, USA, Apr. 2006, pp. 348–359.

[32] C. Jou, H.-J. Shyur and C.-Y. Yen, “Timed sequential pattern mining based on confidence in accumulated intervals,” in *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration*, Redwood City, CA, USA, Aug. 2014, pp. 771–778.

[33] M. Y. Alzahrani and F. A. Mazarbhuiya, “Discovering constraint-based sequential patterns from medical datasets,” *International Journal of Recent Technology and Engineering*, vol. 8, no. 4, pp. 724–728, Nov. 2019.

[34] K. Uragaki, T. Hosaka, Y. Arahori, M. Kushima, T. Yamazaki and K. Araki, “Sequential pattern mining on electronic medical records with handling time intervals and the efficacy of medicines,” in *Proceedings of the IEEE Symposium on Computers and Communication*, Messina, Italy, Jun. 2016, pp. 20–25.

[35] H. H. Le, H. Edman, Y. Honda, M. Kushima, T. Yamazaki and K. Araki, “Fast generation of clinical pathways including time intervals in sequential pattern mining on electronic medical record systems,” in *Proceedings of the 2017 International Conference on Computational Science and Computational Intelligence,* Bandung, Indonesia, Dec. 2017, pp. 1726–1731.

[36] V. Purushothama Raju and G. P. Saradhi Varma, “Mining closed sequential patterns in large sequence databases,” *International Journal of Database Management Systems*, vol. 7, no.1, pp. 29–42, Feb. 2015.

[37] J. Pei et al, “Mining sequential patterns by pattern-growth: The PrefixSpan approach,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 11, pp.1424–1440, Nov. 2004.

[38] M.-Y. Lin and S.-Y. Lee, “Incremental update on sequential patterns in large databases by implicit merging and efficient counting,” *Information Systems*, vol. 29, no. 5, pp. 385–404, Jul. 2004.

[39] S. Parthasarathy, M. J. Zaki, M. Ogihara and S. Dwarkadas, “Incremental and interactive sequence mining,” in *Proceedings of the Eighth International Conference on Information and Knowledge Management*, Kansas City, MO, USA, Nov. 1999, pp. 251–258.

[40] Q. Zheng, K. Xu, S. Ma and W. Lu, “The algorithms of updating sequential patterns,” *arXiv preprint cs/0203027*, pp.1–12, Mar. 2002.

[41] F. Masseglia, P. Poncelet and M. Teisseire, “Incremental mining of sequential patterns in large databases,” *Data & Knowledge Engineering*, vol. 46, no. 1, pp. 97–121, Jul. 2003.

[42] M. Zhang, B. Kao B, D. Cheung and C.-L. Yip, “Efficient algorithms for incremental update of frequent sequences,” in *Proceedings of the 6th Pacific-Asia Conference on Knowledge Discovery and Data Mining,* Taipei, Taiwan, May 2002, pp. 185–194.

[43] M. Zhang, B. Kao, C. L. Yip and D. Cheung, “A GSP-based efficient algorithm for mining frequent sequences,” in *Proceedings of ic-ai*, Hong Kong, China, June, 2001, pp. 497–503.

[44] M.-Y. Lin and S.-Y. Lee, “Incremental update on sequential patterns in large databases by implicit merging and efficient counting,” *Information Systems*, vol. 29, no. 5, pp. 385–404, Jul. 2004.

[45] M. El-Sayed, C. Ruiz and E. A. Rundensteiner, “FS-Miner: Efficient and incremental mining of frequent sequence patterns in web logs,” in *Proceedings of the 6th Annual ACM International Workshop on Web Information and Data Management*, Washington, D.C., USA, Nov. 2004, pp. 128–135.

[46] M.-Y. Lin, S.-C. Hsueh and C.-C. Chan, “Incremental discovery of sequential patterns using a backward mining approach,” in *Proceedings of the 12th IEEE International Conference on Computational Science and Engineering*, Vancouver, BC, Canada, Aug. 2009, pp. 64–70.

[47] X. Lyu and H. Ma, “An efficient incremental mining algorithm for discovering sequential pattern in wireless sensor network environments,” *Sensors Journal*, vol. 19, no. 1, pp. 1-29, Dec. 2018.

[48] S. Saleti and Subramanyam R. B. V., “A MapReduce solution for incremental mining of sequential patterns from big data,” *Expert Systems with Applications*, vol. 133, pp. 109–125.

[49] L. Wang, L. Gui and P. Xu, “Incremental sequential patterns for multivariate temporal association rules mining,” *Expert Systems with Applications*, vol. 207, pp. 118020 – 118034, Sep. 2022.

[50] T. Guyet, W. Zhang and A. Bifet, “Incremental mining of frequent serial episodes considering multiple occurrences,” in *Proceedings of the International Conference on Computational Science*, London, UK, Jun. 2022, pp. 460–472.

[51] J.-W. Huang, C.-Y. Tseng, J.-C. Ou and M.-S. Chen, “A general model for sequential pattern mining with a progressive database,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 9, pp. 1153–1167, Sep. 2008.

[52] A. Mhatre, M. Verma and D. Toshniwal, “Extracting sequential patterns from progressive databases: A weighted approach,” in *Proceedings of the 2009 International Conference on Signal Processing Systems*, Patiala, India, Mar. 2009, pp. 788–792.

[53] J.-W. Huang, S.-C. Lin and M.-S. Chen, “DPSP: Distributed progressive sequential pattern mining on the Cloud,” *Advances in Knowledge Discovery and Data Mining (PAKDD 2010),* vol. 6119, pp. 27–34, Jun. 2010.

[54] S. Chakrabarti and H. N. Saha, “Parallel progressive sequential pattern (PPSP) mining,” in *Proceedings of the 2022 IEEE 12th Annual Computing and Communication Workshop and Conference*, Las Vegas, NV, USA, Jan. 2018, pp. 294–300.

[55] A. Jamshed, S. Singh and R. K. Bharti, “An analysis of sequential pattern mining approach for progressive database by deep learning technique,” in *Proceedings of the 6th International Conference on Intelligent Computing and Control Systems*, Erode, India, Mar. 2022, pp. 1409–1415.

[56] S. Karsoum, C. Barrus, L. Gruenwald and E. Leal, “Minits-AllOcc: An efficient algorithm for mining timed sequential patterns,” in *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Online, May 2021, pp. 667–679.

[57] Royal Meteorological Society. “The Beaufort Wind Scale.” Accessed: April 2, 2024. [Online.] Available: https://www.rmets.org/metmatters/beaufort-wind-scale

[58] J. Yuan, Y. Zheng, C. Zhang, W. Xie, X. Xie, G. Sun and Y. Huang, “T-Drive: Driving directions based on taxi trajectories,” in *Proceedings of the 18th SIGSPATIAL*

International Conference on Advances in Geographic Information Systems, San Jose, CA, USA, Nov. 2010, pp. 99–108.

[59] J. Yuan, Y. Zheng, X. Xie and G. Sun, "T-drive: Enhancing driving directions with taxi drivers' intelligence," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 1, pp. 220–232, Jan. 2013.

[60] Y. Zheng, "Trajectory data mining: An overview," *ACM Transactions on Intelligent Systems and Technology*, vol. 6, no. 3, pp. 1–41, May 2015.

[61] S. Qiao, C. Tang, S. Dai, M. Zhu, J. Peng, H. Li and Y, Ku, "PartSpan: Parallel sequence mining of trajectory patterns," in *Proceedings of the 5th International Conference on Fuzzy Systems and Knowledge Discovery*, Jinan, China, Oct. 2008, pp. 363–367.

[62] Y. Zheng and X. Xie, "Learning travel recommendations from user-generated GPS traces," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 1, pp. 1–29, Jan. 2011.

[63] C.-C. Chen and M.-F. Chiang, "Trajectory pattern mining: Exploring semantic and time information," in *Proceedings of the 2016 Conference on Technologies and Applications of Artificial Intelligence*, Hsinchu, Taiwan, Nov. 2016, pp. 130–137.

[64] M. Ester, H.-P. Kriegel, J. Sander and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, OR, Aug. 1996, pp. 226–231.

[65] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, Sep. 1994, pp. 487–499.

[66] C. Buchta, M. Hahsler, and D. Diaz, arulesSequences: Mining Frequent Sequences, R package version 0.2-31, Aug. 22, 2024. Online, Available: https://CRAN.R-project.org/package=arulesSequences

[67] Y. Zheng, Q. Li, Y. Chen, X. Xie, and W.-Y. Ma, "Understanding mobility based on GPS data," in *Proceedings of the 10th International Conference on Ubiquitous Computing*, Seoul, South Korea, Sep. 2008, pp. 312–321.

[68] Y. Zheng, X. Xie and W.-Y. Ma, “GeoLife: A collaborative social networking service among user, location and trajectory,” *EEE Data Engineering Bulletin*, vol. 33, pp. 32–39, Jun. 2010.

[69] Y. Zheng, L. Zhang, X. Xie, W.-Y. Ma, “Mining interesting locations and travel sequences from GPS trajectories,” in *Proceedings of the 18th International World Wide Web Conference*, Madrid, Spain, Apr. 2009, pp. 791–800.

[70] J.-G. Lee, J. Han and K.-Y. Whang, “Trajectory clustering: A partition-and- group framework,” in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, Beijing, China, Jun. 2007, pp. 593–604.

[71] S.S. Ho and S. Ruan, “Preserving privacy for interesting location pattern mining from trajectory data,” *Transactions on Data Privacy*, vol. 6, no. 1, pp. 87–106, Apr. 2013.

[72] E. Schubert, J. Sander, M. Ester, H. P. Kriegel and X. Xu, “DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN,” *ACM Transactions on Database Systems*, vol. 42, no. 3, pp.1–21, Jul. 2017.

[73] J. Han, M. Kamber and J. Pei. *Data Mining: Concepts and Techniques*, 3rd ed. San Francisco, CA, USA: Morgan Kaufman, 2023.

[74] J. Yuan, Y. Zheng, X. Xie and G. Sun, “Driving with knowledge from the physical world,” in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, CA, USA, Aug. 2011, pp. 316–324.

[75] P. Fournier-Viger. “SPMF: A Java open-source data mining library.” Accessed April 1, 2019. [Online.] Available: https://www.philippe-fournier-viger.com/spmf/

[76] S. Karsoum, L. Gruenwald and E. Leal, “Impact of trajectory segmentation on discovering trajectory sequential patterns,” in *Proceedings of the 2018 IEEE International Conference on Big Data*, Seattle, WA, Dec. 2018, pp. 3432–3441.

[77] E. Salvemini, F. Fumarola, D. Malerba and J. Han, “FAST sequence mining based on sparse Id-lists,” in *Proceedings of the International Symposium on Methodologies for Intelligent Systems*, Warsaw, Poland, Jun. 2011, pp. 316–325.

[78] US Department of Commerce NNWS. “What is the heat index?” Accessed: April 2, 2024. [Online.] Available:

https://www.weather.gov/ama/heatindex#:~:text=The%20heat%20index%2C%20also%20known,for%20the%20human%20body's%20comfort

[79] U.S. Geological Survey. "The 100-year flood." Accessed April 2, 2024. [Online.] Available: https://www.usgs.gov/special-topics/water-science-school/science/100-year-flood?qt-sci%ADence_center_objects=0#qt-science_center_objects.

[80] U.S. Department of Commerce NNWS. "Dew point vs humidity." Accessed: April 2, 2024. [Online.] Available: https://www.weather.gov/arx/why_dewpoint_vs_humidity#:~:text=The%20dew%20point%20is%20the,water%20in%20the%20gas%20form