國立交通大學

資訊科學與工程研究所

博士論文

探勘時間間隔循序特徵樣式之相關研究

A Study on Time Interval-based Sequential Patterns Mining

研 究 生：陳以錚

指導教授：李素瑛　教授

彭文志　副教授

中 華 民 國 一〇一 年 六 月

探勘時間間隔循序特徵樣式之相關研究

A Study on Time Interval-based Sequential Patterns Mining

研 究 生：陳以錚　　　　　　Student：Yi-Cheng Chen

指導教授：李素瑛、彭文志　　Advisor：Suh-Yin Lee, Wen-Chih Peng

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

博 士 論 文

A Dissertation

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

June 2012

Hsinchu, Taiwan, Republic of China

探勘時間間隔循序特徵樣式之相關研究

研究生：陳以錚　　指導教授：李素瑛 博士、彭文志 博士

國立交通大學資訊工程博士班

摘　要

循序特徵樣式因其廣泛的實用性，在資料探勘的領域中一直扮演著非常重要的角色，藉著探勘出存在的時間順序對許多相關實務有很大的幫助。但現今的演算法大都只考慮針對時間點的循序樣式探勘，並無時間的持續概念，處理時間間隔的相關演算法與應用領域一直為學者專家所忽略。本論文專注於探討時間間隔循序樣式之相關問題，研究如何設計正確的表示方法與有效率的探勘演算法，並討論所產生之相關技術。表示方法(representation)是在處理時間間隔序列時，最基礎的問題。對以時間間隔為基礎的循序樣式而言，單純依照發生時間的所排序出的前後關係，並無法表示出一個完整的時間間隔循序樣式。本論文改進目前幾種表示法的缺點，在有效利用儲存空間的前提之下，提出了兩種表示法：同時片段表示法(coincidence representation)與端點表示法(endpoint representation)，能有效表達樣式中間隔彼此的關係，並且避免產生混淆問題(ambiguous problem)。以片段表示法為基礎，我們設計了一個能在大型資料庫中，有效率探勘時間間隔特徵樣式的演算法(CTMiner)。本論文也考慮了相關變化型樣式，設計出了以端點表示法為基礎，探勘封閉式時間間隔特徵樣式的演算法(CEMiner)與漸增式探勘時間間隔特徵樣式的演算法(Inc_CTMiner)。從合成與真實測資的實驗結果可得知，我們所提出的三種演算法，都能有效率且正確地找出所有的相關時間間隔特徵樣式，並且只需少量

的記憶體使用量。本論文也將三種演算法實際應用於現實生活的資料上，找出有用的時間間隔特徵樣式，以證明演算法的實用性。

關鍵詞：循序特徵樣式、封閉循序特徵樣式、漸增式樣式探勘、時間間隔特徵樣式、表示方法。

A Study on Time Interval-based Sequential Patterns Mining

Student: *Yi-Cheng Chen* Advisor: Dr. *Suh-Yin Lee*, Dr. *Wen-chih Peng*

Department of Computer Science and Information Engineering
National Chiao Tung University

ABSTRACT

Sequential pattern mining is a key issue in data mining. However, most of the previous studies are mainly focused on time point-based event data. Little attention has been paid to mining patterns from time interval-based event data, where each event persists for a time interval. Since intervals may overlap, the relation between any two intervals is intrinsically complex. In this dissertation, we propose two new representations, **coincidence representation** and **endpoint representation** to simplify the processing of complex relations among event intervals. Then, three efficient algorithms, **CTMiner**, **CEMiner** and **Inc_CTMiner**, are developed to discover several types of temporal patterns from interval-based data. Based on coincidence representation, an efficient algorithm, **CTMiner** is developed to discover frequent temporal patterns from interval-based data. The algorithm employs two pruning techniques to reduce the search space effectively. The mining of closed sequential patterns has attracted researchers for its capability of using compact results to preserve the same expressive power as conventional mining. In this dissertation, a novel algorithm, CEMiner, is developed to discover closed temporal patterns based on endpoint representation. Algorithm CEMiner also utilizes some optimization technique to reduce the search space in processing. In several real-life applications, sequence databases generally update incrementally with time. A number of discovered sequential patterns may be invalidated, and a number of new patterns may be introduced by the evolution on the database. We proposed an efficient algorithm, Inc_CTMiner to incrementally mine temporal patterns in interval database. Moreover, the algorithm employs some optimization techniques to reduce the search space effectively. The experimental studies indicate that all proposed algorithms are efficient and scalable and outperforms the state-of-the-art algorithms. The improvement of proposed pruning strategies also has been discussed. Furthermore, we also apply our algorithms on real data to show the efficiency and validate the practicability of interval-base temporal mining.

Keywords: sequential pattern, closed sequential pattern, incremental pattern mining, temporal pattern mining, representation.

誌謝

在博士班期間，首先要感謝的就是指導教授李素瑛老師。李老師在研究上的諄諄善誘與耐心教誨，才讓我得以完成博士論文。李老師嚴謹的治學態度，於研究與投稿的每個階段屢次再現，總見老師百忙中抽空，費心審查。除了在學術方面的指導外，李老師在生活態度、待人處世，以及各方面給予我的教導，都遠遠超出一名指導教授的職責，不僅讓我終身受益，也讓我深刻感受到李老師對於學生的衷心關愛與呵護。

我也要感謝我的共同指導教授彭文志老師，平日研究上的指導，總能指出我尚需改進之處。在一些人生未來的方向上，彭老師也給予我寶貴的意見與幫助，讓我獲益良多。

在此我以真摯的心感謝所有口試委員，不吝於提供多年的寶貴研究經驗，充實了本論文的深度與廣度。謝謝陳銘憲老師、陳良弼老師、曾新穆老師、沈錳坤老師、與黃俊龍老師，為豐富本論文內容提供絕佳的意見，在方法適用範圍、方法評比、研究結果的適切論述、方法的差異性等等見解，使本論文更臻完善。諸位口試委員是我學術研究的最佳典範，也是我最值得學習的對象。

此外，資訊系統實驗室的學長姊及學弟妹們，非常感謝你們在這段期間的協助與支持，不僅同窗，更為好友；是你們使我的研究之路如此精彩，如此溫馨。

最後要感謝我最親愛的家人。先父陳哲浩先生、母親葉忱女士和姊姊陳以涵女士。有你們分享我的一切，這才使我的努力有了意義。無論是在日常生活上的悉心照顧與關懷，還是在博士班期間給予我永無止盡的精神支持，都是我源源不絕的動力來源。雖然先父尚未看見我完成學業即匆促辭世，但我仍要將此論文獻給我的父親，我依然是追隨著您的腳步。

再次感謝一路走來教誨我、陪伴我、支持我的大家，謝謝。

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently, sequential pattern mining is an active research topic in data mining domain, due to its widespread applicability. This kind of applications always considers order relation and time issue in our daily lives. Sequential pattern mining mainly deals with extracting the positive behavior of a sequential pattern that can help in predicting the next event after a sequence of events. However, finding sequential patterns is a difficult problem since the mining may have to generate or examine a large number of intermediate subsequence combinations. Many sequential pattern mining algorithms have focused on exploring an approach to discover frequent time-point based correlations or patterns in large sets of temporal data.

However, in many real-world scenarios, events usually tend to persist for periods of time instead of instantaneous occurrences, cannot be treated as "time points". In such cases, the data is usually a sequence of events with both start and finish times. Much existing research mainly focused on discovering patterns from time point-based event data. These approaches is hampered by the fact that they can only handle instantaneous events efficiently, not event intervals. By comprehensive observation, we can perceive that time point-based issue is just a special case of the time interval-based issue (where start time is identical to finish time), but not vise versa. Mining time interval-based patterns (also called **temporal patterns**) from such data is undoubtedly more complex and arduous, and requires a different approach from mining time point-based data, such as mining traditional sequential patterns or episode.

To the best of our knowledge, all the related research in this domain is based on Allen's temporal logics [2], which are categorized into 13 temporal relations between any two event intervals as: "*before*," "*after*," "*overlap*," "*overlapped by*," "*contain*," "*during*," "*start*," "*started by*," "*finish*," "*finished by*," "*meet*," "*met by*," and "*equal*." These 13 relationships can describe any relative position of two intervals based on the arrangements of the start and the finish end time points. However, all the Allen's logics are binary relation and may suffer several problems when describing relationships among more than three events. An appropriate representation is very crucial for facing this circumstance. Various representations have been proposed but most of them have restriction on either ambiguity or scalability.

In this dissertation, we discuss three issues related to temporal pattern, mining temoral pattern, mining closed temporal pattern and incremental mining of temoral pattern. For each issue, we discuss its challenge and the major bottleneck, and propose proper representation for processing intervals among event sequence. Based on proposed representations, some algorithms are given to address each issue.

For temporal pattern mining, we develop the concept of slice-coincidence to trim the processing of complicated relationship among event intervals effectively, and facilitate the temporal pattern mining. Allen's 13 temporal logics can be reduced to simply three relationships, i.e. "*before*," "*equal*" and "*after*." All event intervals are incised to event slices and grouped into coincidence regarding to the global information of end time arrangements in the sequence. Utilizing the incision strategy, a new algorithm, **CTMiner (Coincidence Temporal Miner)** is proposed to address the crucial problem and discover the frequent temporal patterns efficiently and effectively. Experimental studies on both synthetic and real datasets indicate that proposed strategy and algorithm are both efficient and scalable and outperforms state-of-the-art algorithms. Furthermore, our experiments also show that the proposed approach consumes a much smaller memory space.

For closed temporal pattern mining, we simplify the processing of complex relations among intervals by capturing the global information of all endpoints in a sequence. Various existing representations may lead to different kinds of problem. We develop a compact representation, endpoint representation, to express a pattern or sequence nonambiguously. Endpoint representation can facilitate the process and improve the performance of algorithm. A novel algorithm, **CEMiner**, which stands for Closed Endpoint Temporal Miner, is proposed to discover closed temporal patterns efficiently and effectively. Furthermore, CEMiner employs some optimization strategies to reduce the search space and avoids nonpromising closure checking and database projection.

For incremental mining of temporal pattern, this dissertation proposes an efficient algorithm, **Inc_CTMiner** which stands for *Inc*remental *C*oincidence *T*emporal *Miner*, to address the crucial problem and incrementally discover temporal patterns based on the coincidence representation. Furthermore, Inc_CTMiner employs some pruning strategies to reduce the search space. Experimental studies on both synthetic and real datasets indicated that, in the incremental environment, Inc_CTMiner is efficient and outperforms the state-of-the-art algorithms, which are based on static database. Our experiments also revealed that the proposed approach is scalable and consumes a smaller memory space. We also applied Inc_CTMiner on real world datasets to demonstrate the practicability of maintaining the temporal patterns.

The rest of the dissertation is organized as follows. Chapter 2 gives a novel algorithm for mining temporal patterns from interval-based data. Chapter 3 addresses the problem of closed temporal pattern mining and develops a novel algorithm for finding closed patterns from interval-based data. Chapter 4 provides the detailed discussion for an incremental mining algorithm for temporal patterns from interval-based database. Finally, we conclude in Chapter 5.

Chapter 2

An Efficient Algorithm for Mining Temporal Patterns from Interval-based Data

2.1 Introduction

Recently, sequential pattern mining is an active research topic in data mining domain, due to its widespread applicability. This kind of applications always considers order relation and time issue in our daily lives. Sequential pattern mining mainly deals with extracting the positive behavior of a sequential pattern that can help in predicting the next event after a sequence of events. However, finding sequential patterns is a difficult problem since the mining may have to generate or examine a large number of intermediate subsequence combinations. Most of the previous sequential pattern mining algorithms, such as GSP [32], MEMISP [20], PrefixSpan [30], PSP [22] and SPADE [39] to name a few, focus on exploring an approach to discover frequent time-point based correlations or patterns in large sets of temporal data.

In many real world scenarios, some events, which intrinsically tend to persist for periods of time instead of instantaneous occurrences, cannot be treated as "time points". In such cases, the data is usually a sequence of events with both start and finish times. For example, in the medical field, some relationships can be mined from clinical records of patients to study the correlations between the symptoms and the diseases, or the influences between the diseases and other diseases. One may find that during Kawasaki disease infections, the patients often begin with a high-grade and persistent fever. Another discovery might be that during the presence of Kawasaki diseases, the affected patients develop red eyes, red mucous membranes in the mouth, and cracked red lips.

Much existing research mainly focused on discovering patterns from time point-based event data. These approaches is hampered by the fact that they can only handle instantaneous events efficiently, not event intervals. By comprehensive observation, we can perceive that time point-based issue is just a special case of the time interval-based issue (where start time is identical to finish time), but not vise versa. Mining time interval-based patterns (also called **temporal patterns**) from such data is undoubtedly more complex and arduous, and requires a different approach from mining time point-based data, such as mining traditional sequential patterns or episode.

To the best of our knowledge, all the related research in this domain is based on Allen's temporal logics [2], which are categorized into 13 temporal relations between any two event intervals as: "*before*," "*after*," "*overlap*," "*overlapped by*," "*contain*," "*during*," "*start*," "*started by*," "*finish*," "*finished by*," "*meet*," "*met by*," and "*equal*." These 13 relationships can describe any relative position of two intervals based on the arrangements of the start and the finish end time points, as shown in Table 2.1. However, all the Allen's logics are binary relation and may suffer several problems when describing relationships among more than three events. An appropriate representation is very crucial for facing this circumstance. Various representations have been proposed but most of them have restriction on either ambiguity or scalability.

Table 2.1: Allen's 13 relations between two intervals

Temporal Relation	**Inversed Relation**	**Pictorial Example**	**Endpoints Constraint (*s*: starting time, *f*: finishing time)**
A before *B*	*B* after *A*	A B	$A.f < B.s$
A overlaps *B*	*B* overlapped-by *A*	A B	$(A.s < B.s) \wedge (A.f > B.s) \wedge (A.f < B.f)$
A contains *B*	*B* during *A*	A B	$(A.s < B.s) \wedge (A.f > B.f)$
A starts *B*	*B* started-by *A*	A B	$(A.s = B.s) \wedge (A.f < B.f)$
A finished-by *B*	*B* finishes *A*	A B	$(A.s > B.s) \wedge (A.f = B.f)$
A meets *B*	*B* met-by *A*	A B	$A.f = B.s$
A equal *B*	*B* equal *A*	A B	$(A.s = B.s) \wedge (A.f = B.f)$
A after *B*	*B* before *A*	B A	$B.f < A.s$
A overlapped-by *B*	*B* overlaps *A*	B A	$(B.s < A.s) \wedge (B.f > A.s) \wedge (B.f < A.f)$
A during *B*	*B* contains *A*	B A	$(B.s < A.s) \wedge (B.f > A.f)$
A started-by *B*	*B* starts *A*	B A	$(B.s = A.s) \wedge (B.f < A.f)$
A finishes *B*	*B* finished-by *A*	B A	$(B.s > A.s) \wedge (B.f = A.f)$
A met-by *B*	*B* meets *A*	B A	$B.f = A.s$

In this chapter, a fundamentally different technique from previous work is proposed to discover temporal patterns. Without any doubt, the major bottleneck of temporal mining task is the complex relationship among event intervals. We develop the concept of slice-coincidence to trim the processing of complicated relationship among event intervals effectively, and facilitate the temporal pattern mining. Allen's 13 temporal logics can be reduced to simply three relationships, i.e. "*before*," "*equal*" and "*after*." All event intervals are incised to event slices and grouped into coincidence regarding to the global information of end time arrangements in the sequence. Utilizing the incision strategy, a new algorithm, **CTMiner (Coincidence Temporal Miner)** is proposed to address the crucial problem and discover the frequent temporal patterns efficiently and effectively. Experimental studies on both synthetic and real datasets indicate that proposed strategy and algorithm are both efficient and scalable and outperforms state-of-the-art algorithms. Furthermore, our experiments also show that the proposed approach consumes a much smaller memory space.

The rest of this chapter is organized as follows. Section 2.2 gives the related work. Section 2.3 provides the detailed definitions. Section 2.4 introduces the incision strategy and the coincidence representation. Section 2.5 describes the CTMiner algorithm. Section 2.6 presents the experiments and performance study, and we summarize in Section 2.7.

2.2 Related Work

Sequential pattern mining is one of the most important research themes in data mining. Recently, there has been a stream of research on it [1, 3, 6, 10, 11, 18, 20, 21, 22, 30, 32, 39] and its extensions, including closed patterns [4, 5, 15, 34, 38, 40], incremental pattern mining [4, 5, 7, 9, 12, 14, 19, 23, 26, 28, 42] to name a few. Almost all of these related studies mentioned above are focused on time point-based event data which has no duration concept. Some recent works have investigated the mining of interval-based events [2, 13, 16, 17, 24, 25, 27, 29, 31, 33, 35, 36, 37, 41].

Villafane et al. [33] proposed a graph mining technique to discover time interval-based sequential pattern by transforming data sequences to containment graphes. However, the containment rules discussed are constrained only to "*contains*" and "*during*." Kam et al. [16] proposed a compact encoding method, hierarchical representation and designed an algorithm to discover frequent temporal patterns. Although hierarchical representation only use $k + (k - 1) = 2k - 1$ memory space for describing a k-intervals pattern (k event indices, $k - 1$ describers), it may suffer from two ambiguous problems. First, the same relationships among event intervals can be mapped to different temporal patterns. As shown in Fig. 2.1(a), the pattern can be expressed as "((*A overlaps B*) *before C*) *overlaps D* " or "(*A overlaps B*) *before* (*C during D*)." Second, the same temporal pattern can represent different relationships among event intervals. For example, Fig. 2.1(b) shows that pattern "(*A overlaps B*) *overlaps C* " can represent two different relations among intervals.

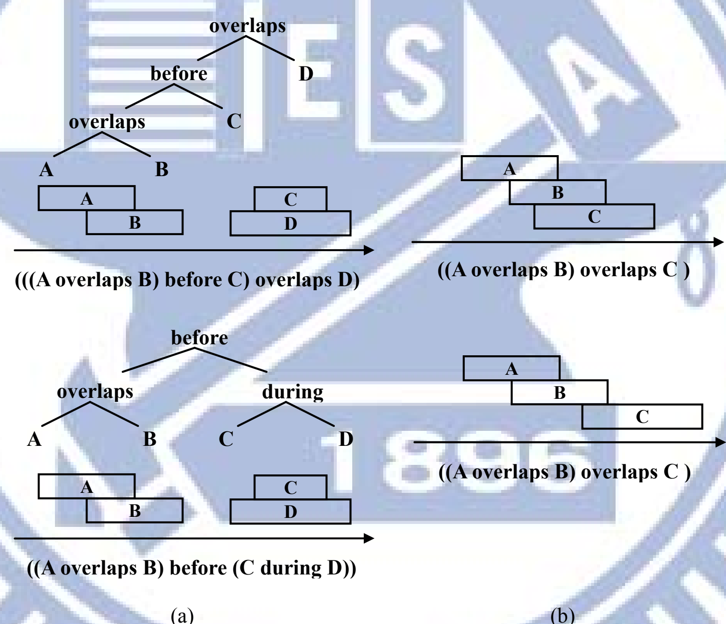
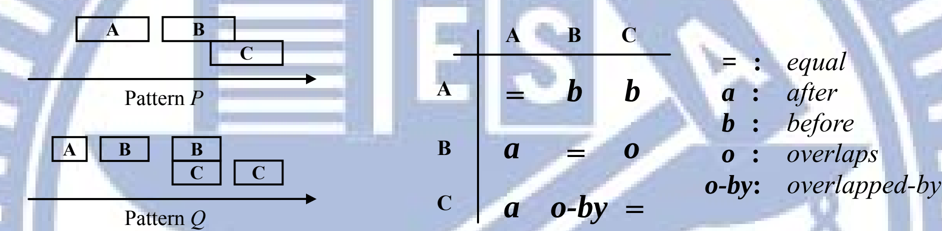


Fig. 2.1: Example of two ambiguous problems of hierarchical representation

Rainsford et al. [31] presented an approach that combine temporal semantics with association rules. The algorithm firstly generates the traditional association rules, and then finds all the possible pairings of temporal items in each rule. Hoppner [13] proposed a nonambiguous representation, relation matrix which exhaustively lists all binary relationships between event intervals in a pattern. For example, pattern P in Fig. 2.2(a) can be represented as a matrix in Fig. 2.2(b). The relation matrix does not scale well if plenty of intervals appear in a pattern since it needs $2k + (k \times (k-1)) = k^2 + k$ memory space to describe a k-intervals pattern ($2k$ event indices, $k^2 - k$ describers).

H-DFS [27] was proposed to discovery frequent arrangements of temporal intervals. This approach transforms an event sequence into a vertical representation using id-lists. The id-list of one event is merged with the id-list of other events to generate temporal patterns. TSKR [24] expressed the temporal concepts of coincidence and partial order for interval patterns. The pattern represented in TSKR format is easily understandable but may reveal the relationship between pairwise event intervals in a pattern ambiguously. For example, in Fig. 2.2(a), pattern P and Q are represented as the identical TSKR expression "*AB*(*BC*)*C*."



(a) Two example temporal patterns (b) Relation matrix for *P*

Fig. 2.2: Example of relation matrix representation

Laxman et al. [17] extended the original framework of frequent episode discovery in event sequences by incorporating event duration constraints. The authors also presented some algorithms based on finite-state automaton. Based on the efficient algorithm MEMISP [20], the algorithm ARMADA [35] is proposed to find frequent temporal patterns from large database. DTP [41] partitions database into some disjoint datasets, so that scanning the whole database could be avoided when calculating the support of each pattern. However, DTP only discusses two of the Allen relationships: "*contains*" and "*during*.

Temporal representation [36] utilizes endpoint arrangements to represent the temporal pattern nonambiguously. For example, in Fig. 2.2(a), pattern P can be represented as the expression "$A^+ < A^- < B^+ < C^+ < B^- < C^-$", where "+" and "−" represent the start and finish endpoints of an event interval, respectively. It requires $2k + (2k - 1) = 4k - 1$ space to describe a k-intervals pattern ($2k$

event indices, $2k - 1$ describers). TPrefixSpan [36] used temporal representation to discover frequent temporal patterns. TPrefixSpan first generates all the possible candidates and then discovers frequent events and scans the projected databases for support counting.

Patel et al. [29] utilized additional counting information to achieve a lossless hierarchical representation, named augmented representation. Every Allen describer must take a space to store five counters, i.e., *contain*, *finish-by*, *meet*, *overlap* and *start* counters for accumulating the occurrences of corresponding relations. For example, in Fig. 2.2(a), pattern P can be represented as expression "(A *before*[0,0,0,0,0] B) *overlaps*[0,0,0,1,0] C." The counter of overlap describer is [0,0,0,1,0] since C only overlaps B. Augmented hierarchical representation is not easily comprehensible and needs $k + (k - 1) \times 6 = 7k - 6$ memory space in a k-intervals pattern (k event indices, $6\times(k-1)$ describers). IEMiner [29] was designed to discover frequent temporal patterns from interval-based events based on the augmented representation.

HTPM [37] was developed to mine hybrid temporal pattern from event sequences, which contain both point-based and interval-based events. Authors modify temporal representation [36] to also express event points. Moerchen et al. developed a new kind of pattern, SIPO [25], to express Allen relationship. Authors utilize the boundaries of interval and further consider the noise tolerance. However, SIPO may suffer the ambiguous problem and the mining algorithm requires discovering both closed sequential pattern and closed itemset, and therefore is time consuming.

There are three contributions from our work reported in this chapter. The first contribution is that we propose an incision strategy, to simplify processing complex relations when mining temporal patterns. The incision strategy segments all intervals to disjoint slices based on the global information in a pattern. The second contribution is that we develop a new representation, coincidence representation, to express a pattern or sequence nonambiguously, based on the incision strategy. As mentioned above, various existing representations may lead to different kinds of problem. An appropriate representation can facilitate processing and improve performance of algorithm. Coincidence representation has several advantages and we will discuss in details in section 2.4.2.

The final contribution is that we design a new algorithm, **CTMiner,** which can effectively avoid the effort on candidate generation and test for mining temporal patterns. We first transform interval sequences in database to coincidence format and then borrow the idea from PrefixSpan [21] (Prefix-projected Sequential pattern mining), an efficient pattern growth-based algorithm in finding sequential patterns from transactional database, to mine frequent temporal patterns. Furthermore, CTMiner employs the proposed optimization strategies to reduce the search space and avoids non-promising projection. The performance in both synthetic datasets and real datasets shows that CTMiner outperforms state-of-the-art algorithms. Our experimental results also show that the proposed approach consumes a much smaller memory space.

2.3 Preliminary

Definition 2.1 (Event interval)

Let $E = \{e_1, e_2, \ldots, e_k\}$ be the set of event symbols. Without loss of generality, we define a set of uniformly spaced time points based on the natural number N. We say the triplet $(e_i, s_i, f_i) \in E \times N \times N$ is an event interval, where $e_i \in E$, $s_i, f_i \in N$ and $s_i < f_i$. The two time points s_i, f_i are called event times, where s_i is the starting time and f_i is the finishing time. The set of all event intervals over E is denoted by I.

Definition 2.2 (Event sequence and maximal property)

An event sequence is a series of event interval triplets $\langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_n, s_n, f_n)\rangle$, where $s_i \leq s_{i+1}$, and $s_i < f_i$. Every interval (e_i, s_i, f_i) must be maximal in a sequence, that there is no (e_j, s_j, f_j) in the sequence such that $e_i = e_j$ and $[s_i, f_i)$, $[s_j, f_j)$ overlap or meet each other. We call this assumption, maximal property, defined as follows:

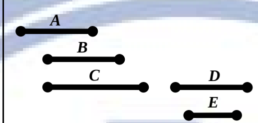
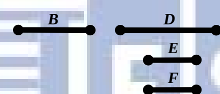
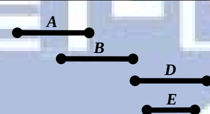
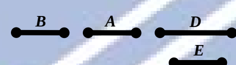
$$\forall\ (\boldsymbol{e_i}, \boldsymbol{s_i}, \boldsymbol{f_i}), (\boldsymbol{e_j}, \boldsymbol{s_j}, \boldsymbol{f_j}) \in \boldsymbol{I}, \boldsymbol{i} \neq \boldsymbol{j} : (\boldsymbol{s_i} \leq \boldsymbol{s_j}) \wedge (\boldsymbol{f_i} \geq \boldsymbol{s_j}) \rightarrow \boldsymbol{e_i} \neq \boldsymbol{e_j} \tag{1}$$

(1) is also called the maximality assumption [9]. The maximal property guarantees that each event interval is maximal in the series. If maximal property is violated, we can merge both event intervals and replace them by their union $(e_i, min(s_i, s_j), max(f_i, f_j))$.

Definition 2.3 (Temporal database)

Considering a database $DB = \{r_1, r_2, \ldots, r_m\}$, each record r_i, where $1 \le i \le m$, consists of a sequence-id and an event interval (i.e. an event symbol, a starting time, and a finishing time, where starting time < finishing time). *DB* is called a temporal database.

Table 2.2: Database example

SID	event symbol	start time	finish time	event sequence	coincidence sequence
1	*A*	2	7		$\boldsymbol{A^+(A^-B^+C^+)\,B^-C^-D^+E\,D^-}$
1	*B*	5	10		
1	*C*	5	12		
1	*D*	16	22		
1	*E*	18	20		
2	*B*	1	5		$\boldsymbol{B\,D^+(EF)\,D^-}$
2	*D*	8	14		
2	*E*	10	13		
2	*F*	10	13		
3	*A*	6	12		$\boldsymbol{A^+(A^-B^+)\,B^-@\,D^+E\,D^-}$
3	*B*	7	14		
3	*D*	14	20		
3	*E*	17	19		
4	*B*	8	16		$\boldsymbol{B\,A\,D^+E\,D^-}$
4	*A*	18	21		
4	*D*	24	28		
4	*E*	25	27		

Actually, if all records in the database *DB* with the same client-id are grouped together and ordered by nondecreasing start time, the database can be transformed into a collection of event sequences. As a result, the database *DB* can be viewed as a collection of event sequences. For example, in Table 2.2, the temporal database consists of 17 event intervals, and 4 event sequences.

2.4 Incision Strategy and Coincidence Representation

We focus on the discussions of temporal pattern mining due to the widespread applicability of this technique and the lack of research on this topic. However, the time interval-based mining problem is much more arduous than time point-based mining problem. Since the time period of the two intervals may overlap, the relation among event intervals is more complex than that of the event points, as shown in Table 2.1. In this chapter, an efficient strategy is derived to simplify the processing of temporal pattern mining. We also propose a new format to express temporal patterns effectively.

2.4.1 Incision Strategy

By our observation, the complex relations between event intervals are the major bottleneck for mining temporal patterns. We propose an incision strategy to address this critical issue. Before introducing the incision strategy, we give some definitions first.

Definition 2.4 (Time set and time sequence)

Given an event sequence $q = \langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_n, s_n, f_n)\rangle$, A set $T_q = \{s_1, f_1, s_2, f_2, \ldots, s_i, f_i, \ldots, s_n, f_n\}$ is called a time set corresponding to q. If we order all the elements in T_q and eliminate redundant element, we can derive a sequence $TS_q = \langle t_1, t_2, \ldots, t_k\rangle$ where $t_i \in T_q$, $t_i < t_{i+1}$. TS_q is called a time sequence corresponding to q.

Definition 2.5 (Incising function and event slice)

Given an event sequences $q = \langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots(e_i, s_i, f_i), \ldots, (e_n, s_n, f_n)\rangle$ where $(e_i, s_i, f_i) \in I$ and corresponding time sequence TS_q. Let $t_j, t_{j+1} \in TS_q$, an incising function Ψ is defined as,

$$\boldsymbol{\Psi(t_j, t_{j+1}, (e_i, s_i, f_i))} = \begin{cases} e_i & \text{if } (s_i = t_j) \wedge (f_i = t_{j+1}) \\ e_i^+ & \text{if } (s_i = t_j) \wedge (f_i > t_{j+1}) \\ e_i^- & \text{if } (s_i < t_j) \wedge (f_i = t_{j+1}) \\ e_i^* & \text{if } (s_i < t_j) \wedge (f_i > t_{j+1}) \\ \varnothing & \text{otherwise.} \end{cases} \tag{2}$$

An event slice $S = \Psi(t_j, t_{j+1}, (e_i, s_i, f_i))$ and is called,

- ***intact slice*** of event e_i, if $s_i = t_j$ and $f_i = t_{j+1}$, and denoted as e_i;
- ***starting slice*** of event e_i, if $s_i = t_j$ and $f_i > t_{j+1}$, and denoted as e_i^+;
- ***finishing slice*** of event e_i, if $s_i < t_j$ *and* $f_i = t_{j+1}$, *and denoted as* e_i^-;
- ***intermediate slice*** of event e_i, if $s_i < t_j$ and $f_i > t_{j+1}$, and denoted as e_i^*.

Obviously, an event interval can only have one starting slice and one finishing slice but can have many intermediate slices. The corresponding slice of a starting (finishing) slice is defined as the finishing (starting) slice of the same interval.

For example, in Table 2.2, sequence 2 has 4 event intervals, (B, 1, 5), (D, 8, 14), (E, 10, 13), and (F, 10, 13) and its corresponding time set = {1, 5, 8, 14, 10, 13, 10, 13} and time sequence = $\langle 1, 5, 8, 10, 13, 14\rangle$. An event interval D can be incised into three event slices, start slice D^+ = Ψ(8, 10, (D, 8, 14)), intermediate slice D^* = Ψ(10, 13, (D, 8, 14)), and finish slice D^- = Ψ(13, 14, (D, 8, 14)). The event interval B has only one intact slice B = Ψ(1, 5, (B, 1, 5)).

Definition 2.6 (Grouping function and coincidence)

Given an event sequences $q = \langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots (e_i, s_i, f_i), \ldots (e_n, s_n, f_n)\rangle$, where $(e_i, s_i, f_i) \in I$, and $t_j, t_{j+1} \in TS_q = \langle t_1, t_2, \ldots, t_k\rangle$, $1 \le j \le k$-1, a grouping function is defined as,

$$\boldsymbol{\Phi\ (t_j, t_{j+1}, q) = \{\Psi(t_j, t_{j+1}, (e_1, s_1, f_1)), \Psi(t_j, t_{j+1}, (e_2, s_2, f_2)), \ldots, \Psi(t_j, t_{j+1}, (e_n, s_n, f_n))\}.} \quad (3)$$

A coincidence C_j is defined as $\Phi\ (t_j, t_{j+1}, q) = (S_{j1}, S_{j2}, \ldots, S_{j\ell}, \ldots)$ and sorting $S_{j\ell}$ in lexicographic order. For brevity, the brackets are omitted if a coincidence has only one slice, i.e., coincidence (S) is written as S.

With the incising function and grouping function, we can transform an event sequence into slice-and-coincidence expression. However, here come two problems. First, two adjacent intervals and two separate intervals can not be discriminated by merely collecting all coincidences. Accordingly, we use a meet slice, @, to distinguish two adjacent intervals. The slice @ indicates that the finishing slices and/or intact slices in the previous coincidence are adjacent to the starting slices and/or intact slices in the next coincidence. We take sequence 3 in Table 2.2 as example, we can not distinguish the *meet* relation between interval B and D by just collecting all coincidences to form a sequence, i.e., $\langle\ A^+(A^-B^+)\ B^-D^+E\ D^-\ \rangle$. Meet slice @ is inserted between event slice B^- and D^+ to express the *meet* relation. Second, the information of intermediate slice, actually, need not be considered. Without intermediate slice, we still can express an event sequence nonambiguously. For example, as the sequence 2 in Table 2.2, without D^*, sequence $\langle\ B\ D^+(EF)\ D^-\ \rangle$ still can represent sequence 2 correctly.

Definition 2.7 (Coincidence sequence)

Given an event sequences $q = \langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_i, s_i, f_i), \ldots, (e_n, s_n, f_n)\rangle$, by definition 5 and 6, we can derive a coincidence sequence $q_c = \langle C_1, C_2, \ldots, C_{k-1}\rangle$ with meet slice addition and intermediate slice pruning. q_c is also called the coincidence representation of q. Additionally, to deal with multiple occurrences of events, we attach occurrence number to event slices to distinguish multiple occurrences of the same event type in a coincidence sequence. For example, $\langle A_1^+ (B A_1^-) A_2 D\rangle$ is a coincidence sequence with occurrence number, where event A occurs twice.

For a temporal database DB, we can transform it into a set of tuples $\langle sid, q_c\rangle$, where sid is the sequence-id of each event sequence q in DB and q_c is the coincidence representation of q. For example, in Table 2.2, we can transform four event sequences in DB into corresponding coincidence sequences. For better readability, later in this chapter, we suppose that the temporal database has been transformed into coincidence representation.

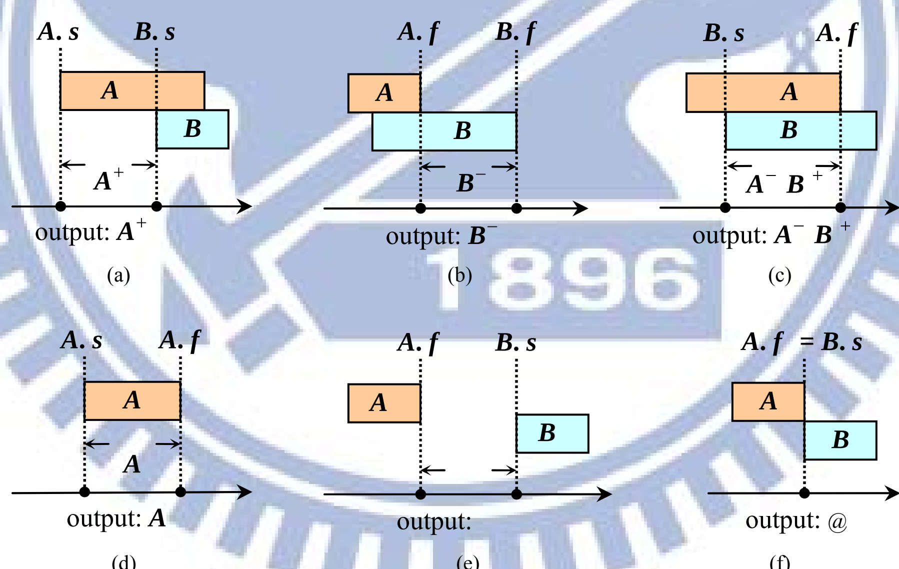


Fig. 2.3: Six possible segmentations between two consecutive end time points

Actually, there are six possible segmentations between any two end time points in a time sequence. Considering two consecutive end time points, we use a "< " or " = " to describe the smaller or equal order relation respectively. Without loss of generality, we use "A " and "B " to represent two different event intervals. All possible segmentations are listed as follows,

1) $\boldsymbol{A.s} < \boldsymbol{B.s}$ **:** The event interval A is segmented to starting slice A^{+} and output, as in Fig. 2.3(a);

2) $\boldsymbol{A.f} < \boldsymbol{B.f}$ **:** The event interval B is segmented to finishing slice B^{-} and output, as in Fig. 2.3(b);

3) $\boldsymbol{B.s} < \boldsymbol{A.f}$ **:** The event interval A and event interval B are segmented to finishing slice A^{-} and starting slice B^{+} respectively, and $A^{-}B^{+}$ is output, as in Fig. 2.3(c);

4) 4) $\boldsymbol{A.s} < \boldsymbol{A.f}$ **:** The event interval A does not require any segmentation. We can directly output the intact slice A, as in Fig. 2.3(d);

5) $\boldsymbol{A.f} < \boldsymbol{B.s}$ **:** we only consider the period between two consecutive end time points. There is no interval nor slice in this time period, so we do nothing in this case, as the interval A and B in Fig. 2.3(e);

6) $\boldsymbol{A.f} = \boldsymbol{B.s}$ **:** instead of segmenting any event interval, we only output the meet slice "@" to assist the distinction of two adjacent event intervals, as the A and B in Fig. 2.3(f).

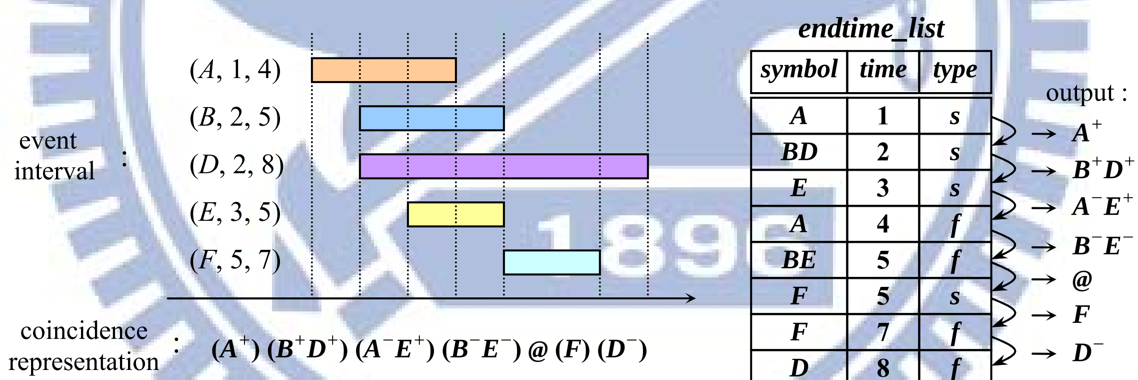
```
algorithm 2.1: incision_strategy ( q )

Input: q: An event sequence
Output: q': A coincidence sequence
Variable: endtime_list, last_endtime, coincidence

endtime_list ← ∅ , last_endtime ← ∅ , coincidence ← ∅ , q' ← ∅ ;
add all the end time points of every event interval in q into endtime_list;
sort every endtime in endtime_list by endtime. time in nondecreasing order;
merge all endtime. symbols together with identical endtime. time and endtime. type;
for each endtime T in endtime_list do
    coincidence ← ∅ ;
    if last_endtime. time = T . time then    // segmentation 6
        coincidence ← coincidence ∪ "@";
    else    // last_endtime. time ≠ T . time
        if last_endtime. type = "s" then    // segmentation 1, 3, and 4
            coincidence ← coincidence ∪ every symbol in last_endtime. symbol add "+";
           // starting slice
        if T.type = "f" then    // segmentation 2, 3, and 4
            coincidence ← coincidence ∪ every symbol in T. symbol add "–";
           // finishing slice
    combine start slice and finish slice with same symbol in coincidence;    // intact slice
    q' ← q' ◇ ⟨ coincidence ⟩ ;    //append coincidence to coincidence sequence
    last_endtime ← T ;
output q' ;
```

Fig. 2.4: The pseudocode of Incision Strategy

In this chapter, we propose an efficient method, incision strategy, to transform an event sequence into coincidence sequence effectively. The pseudo code of incision strategy is elaborated in Fig. 2.4. We use an example to explain the algorithm. Incision strategy first puts all the end time points of every event interval in a sequence into a data structure *endtime_list* which has three attributes: *symbol*, *time* and *type*, as shown in Fig. 2.5(b). Then it sorts the record in *endtime_list* in nondecreasing order based on their times and types (starting or finishing). If the times of two end time points are the same but the types are different, the order is based on the type, i.e., finishing type is smaller than starting type. Then we merge the event symbol of end time points together if both time and type of end time points are identical. For example, considering an event sequence with 5 intervals shown in Fig. 2.5(a), we put all 10 end time points into *endtime_list* and sort them in nondecreasing order as in Fig. 2.5(b). Since the *B.s* is identical to the *D.s*, we can merge them together. But we can not combine *F.s* with *B.f* and *E.f*, since the type of end time points are not the same. Then we traverse all the sorted end time points in *endtime_list* one-by-one to incise the event slices.



(a) an event sequence with 5 intervals (b) *endtime_list* data structure

Fig. 2.5: An example of incision strategy

Reducing memory usage and computation time are two important issues for algorithm design. Since we have utilized meet slice to effectively distinguish two adjacent intervals, intermediate slices need not be incised. Given an example as in Fig. 2.5(a), the event interval D can be segmented into five event slices, one starting slice D^+, three intermediate slices D^*, and one finishing slice D^-. By trimming the intermediate slices, we can still express the relationship between any two intervals correctly. As shown in the graph, this tatic can reduce one-third of storage space, and thereby improves the performance of our incision strategy.

Notice that if the starting slice and is corrsponding finishing slice are in the same coincidence, we have to combine them to form an intact slice since the interval is not incised (Line 14, algorithm 2.1). By the merge operation of incision strategy, the event slices occurring simultaneously in the same time period can be grouped together to form a coincidence easily.

2.4.2 Coincidence Representation

We know that the Allen's 13 relationships are binary relation and may suffer several problems when describing relationships among more than three events. An appropriate representation is very crucial for facing this circumstance. As mentioned above, various representations have been proposed but most of them have restriction on either ambiguity or scalability.

In this chapter, a new representation, coincidence representation is proposed to address the ambiguity and scalability problems. The coincidence representation utilizes the concept of slice and coincidence, and considers the information of entire event sequence instead of individual event interval. By incision strategy, all event intervals in a sequence are segmented into event slices and simultaneously occurring slices are grouped together to form the coincidences. Concatenation of all coincidences can describe an event sequence effectively and simplify the processing of complex pairwise relationships among all intervals efficiently. This is also the primary motivation of coincidence representation.

The coincidence representation of Allen's 13 relations between two event intervals is categorized as in Fig. 2.6. Given two different event intervals "*A*" and "*B*", we discuss Allen's 13 relationships with coincidence representation in details as follows,

(1) (*A* before *B*) or (*B* after *A*) : *A* and *B* are totally disjoint. According to whether the intervals are incised or not, there are four kinds of coincidence representation: $(A)(B)$, $(A)(B^+)(B^-)$, $(A^+)(A^-)(B)$, and $(A^+)(A^-)(B^+)(B^-)$. There may exist some other interleaved event intervals or slices, but the order will not change.

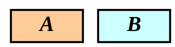
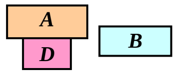
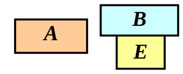
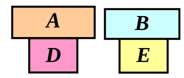
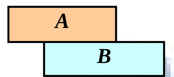
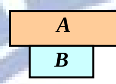
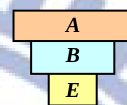
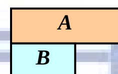
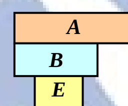
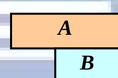
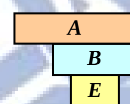
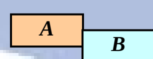
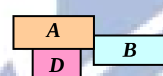
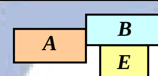
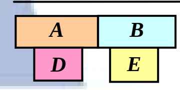
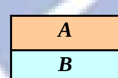
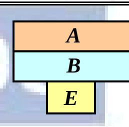
Temporal Relation	Inversed Relation	Pictorial Example	Coincidence representation	Pictorial Example	Coincidence representation
A *before* *B*	*B* *after* *A*	 	**(*A*) (*B*)** **(A^+) (A^-) (*B*)**	 	**(*A*) (B^+) (B^-)** **(A^+) (A^-) (B^+) (B^-)**
A *overlaps* *B*	*B* *overlapped-by* *A*		**(A^+) (A^-B^+) (B^-)**		
A *contains* *B*	*B* *during* *A*		**(A^+) (*B*) (A^-)**		**(A^+) (B^+) (B^-) (A^-)**
A *starts* *B*	*B* *started-by* *A*		**(A^+B) (A^-)**		**(A^+B^+) (B^-) (A^-)**
A *finished-by* *B*	*B* *finishes* *A*		**(A^+) (A^-B)**		**(A^+) (B^+) (A^-B^-)**
A *meets* *B*	*B* *met-by* *A*	 	**(*A*) @ (*B*)** **(A^+) (A^-) @ (*B*)**	 	**(*A*) @ (B^+) (B^-)** **(A^+) (A^-) @ (B^+) (B^-)**
A *equal* *B*	*B* *equal* *A*		**(*AB*)**		**(A^+B^+) (A^-B^-)**

Fig. 2.6: The coincidence representation of Allen's 13 relations between two intervals

(2) **(*A* overlaps *B*) or (*B* overlapped-by *A*) :** A part of *A* intersects a part of *B*, therefore *A* and *B* must both have been incised into event slices. The corresponding coincidence representation is $(A^+)(A^-B^+)(B^-)$. There may exist some other interleaved event intervals or slices, but the order will not change and the finish slice A^- and the start slice B^+ occur simultaneously.

(3) **(*A* contains *B*) or (*B* during *A*) :** A part of *A* intersects the whole of *B*, therefore *A* must have been incised into start and finish slices. If *B* is also incised, the coincidence representation will be $(A^+)(B^+)\ (B^-)(A^-)$. If *B* is not incised, the coincidence representation will be $(A^+)(B)(A^-)$. There may be some other interleaved event intervals or slices, but the order will not change.

(4) (*A* starts *B*) or (*B* started-by *A*) : The whole of *A* intersects a part of *B*, therefore *B* must have been incised into start and finish slices. If *A* is also incised, the coincidence representation is $(A^{+}B^{+})(A^{-})(B^{-})$. If *A* is not incised, the coincidence representation is $(AB^{+})(B^{-})$. There may be some other interleaved event intervals or slices, but the order will not change. The main characteristic is that the start slice or the intact slice of interval *A* occurs with the start slice of *B* simultaneously.

(5) (*A* finished-by *B*) or (*B* finish *A*) :A part of *A* intersects the whole of *B*, therefore *A* must have been incised into start and finish slices. If *B* is also incised, the coincidence representation is (A^{+}) (B^{+}) $(A^{-}B^{-})$. That means the finish slices of *A* and *B* occur simultaneously. If *B* is not incised, the coincidence representation is (A^{+}) $(A^{-}B)$. That means the finish slice of *A* and the intact slice of *B* occur simultaneously. There may exist some other interleaved event intervals or slices, but the order will not change. The main characteristic is that the finish slice or the intact slice of *B* occurs with the finish slice of *A* simultaneously.

(6) (*A* meets *B*) or (*B* met-by *A*) : *A* and *B* are adjacent. Just like the *before* and *after* relations, there are four kinds of coincidence representation. We only utilize a meet slice "@" to discriminate "*meets*" and "*met-by*" relations effectively. According to whether the intervals are incised or not, the corresponding coincidence representation may be represent as $(A)@(B)$, $(A)@(B^{+})(B^{-})$, $(A^{+})(A^{-})@(B)$, and $(A^{+})(A^{-})@(B^{+})$ (B^{-}).

(7) (*A* equal *B*) or (*B* equal *A*): *A* and *B* are entirely overlapped. If *A* and *B* are both incised, the corresponding coincidence representation is $(A^{+}B^{+})$ $(A^{-}B^{-})$. On the contrary, if both *A* and *B* are not incised, the corresponding coincidence representation is (AB). There may exist some other interleaved event intervals or slices, but the order will not change. The main characteristic is that *A* occurs with *B* simultaneously, whether both of them are incised or not.

We utilize coincidence representation to express both event sequences and temporal patterns since it have several advantages, as follows:

- **Nonambiguity:** A representation is ambiguous [13] if 1) the same relationships between intervals may be mapped to different temporal patterns and 2) the patterns cannot reveal the temporal relations among all pairs of intervals. Accordingly, the following observations indicate that the ambiguity no longer exists in our coincidence representation. First, by definition 2.5 and 2.6, we can build a unique coincidence sequence by transforming every event sequence into coincidence representation. In other words, the temporal relations among intervals can be mapped one-to-one to a coincidence sequence. Second, in a coincidence sequence, the order relation of the start and finish slices of *A* and *B* can be categorized as shown in Fig. 2.6. We can infer the original temporal relationships between intervals *A* and *B* nonambiguously.

- **Good scalability:** In the best case, all *k* intervals in a pattern are equal, thus memory space for describing a *k*-intervals pattern is *k*. In the worst case, all *k* intervals overlap one-by-one, thus we require 2*k* memory space to express a *k*-intervals pattern. The coincidence representation scales well even if plenty of intervals appear in a pattern.
- **Simple is good:** Obviously, the complex relations between intervals are the major bottleneck of temporal pattern mining since the mining may need to generate or examine explosive number of intermediate subsequences. By incision strategy, we can transform event intervals into non-overlapped fragments, event slices. The relations between event slices are simple, just "*before*," "*after*" and "*equal*." The simpler the relations, the less number of intermediate candidate sequences are generated and processed. Therefore, with coincidence representation, we can discover frequent temporal patterns more efficiently.
- **Compact space usage:** Since the utilization of meet token, we can omit the intermediate slices within the coincidence sequences or patterns. This tactic can reduce the computation time and memory space efficiently, as shown in Fig. 2.5(c).

2.5 Proposed algorithm

In this section, we propose a new algorithm, called **CTMiner** (**Coincidence Temporal Miner**), to mine frequent temporal patterns efficiently. CTMiner utilizes the concepts of slice-and-coincidence to accomplish the temporal pattern discovering. Section 2.5.1 details the algorithm. We mine temporal patterns based on coincidence representation and propose two pruning mechanisms for reducing the search space. In section 2.5.2, we discuss the difference between traditional sequential projection and temporal projection, and propose a new projection technique, **multi-projection** taking into account of interval-based event sequence. Finally, section 2.5.3 proves the correctness and completeness of CTMiner algorithm.

2.5.1 CTMiner

Definition 2.8 (Projected database)

Let α be a coincidence sequence in a database *DB*. The α-projected database, denoted as $DB|_{\alpha}$, is the collection of suffixes of sequences in *DB* with regards to prefix α.

Definition 2.9 (temporal pattern)

Considering two coincidence sequence $\alpha = \langle a_1, a_2, \ldots, a_n \rangle$ and $\beta = \langle b_1, b_2, \ldots, b_m \rangle$, α is called a subsequence of β, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \le i_1 \le i_2 \le \ldots \le i_n \le m$ such that $a_1 \subseteq b_{i1}, a_2 \subseteq b_{i2}, \ldots, a_n \subseteq b_{in}$, and β is also called a supersequence of α. Given a temporal database *DB*, a tuple $\langle sid, q_c \rangle$ is said to contain a coincidence sequence α, if α is a subsequence of q_c. The support of α in *DB* is the number of tuples in the database containing α, i.e.,

$$\textbf{support}(\alpha) = |\{\langle \textbf{sid}, \textbf{q}_c \rangle \mid (\langle \textbf{sid}, \textbf{q}_c \rangle \in \textbf{DB}) \wedge (\alpha \sqsubseteq \textbf{q}_c)\}|. \tag{4}$$

Given a positive integer *min_sup* as the support threshold, a coincidence sequence α is called frequent if support (α) ≥ *min_sup*. A frequent coincidence sequence is called temporal pattern if all event slices in sequence appear in pair, i.e., every starting (finishing) slice has corresponding finishing (starting) slice.

Let database in Table 2.2 with *min_sup* =2 be an example. The coincidence sequence $\langle (A^+)\ (A^- B^+)\ (B^-) \rangle$ is a temporal pattern since it occurs in sequence 1 and 3, and its *support* = 2 ≥ *min_sup*. A coincidence sequence $\langle (A^+)\ (A^- C^+)\ (C^-) \rangle$ is not frequent since it occurs only in sequence 1, and its *support* = 1 ≤ *min_sup*. Although $\langle (A^+)\ (A^- B^+) \rangle$ is also a frequent coincidence sequence, it is not a temporal pattern due to B^+ has no corresponding finishing slice in sequence.

Fig. 2.7 illustrates the main framework which includes the necessary processing steps of CTMiner. Given a temporal database, the event intervals associated with the same sequence ID are grouped into an event sequence. CTMiner first transforms the temporal database into coincidence respresentation (Line 2, algorithm 2.2), and then calls sub-procedure **CPrefixSpan** to discover and output all temporal patterns (Lines 3-4, algorithm 2.2). By borrowing the idea of the PrefixSpan [30], CPrefixSpan is developed based on the concepts of slice-and-coincidence and with two search space pruning method. CPrefixSpan first scans projected database once to collect all local frequent slices and remove infrequent slices (Lines 1-3, algorithm 2.2). For each frequent slice, we can append it to original prefix to generate a new coincidence sequence with the length increased by 1. This way, the prefixes are extended (Lines 7-12, algorithm 2.2). Finally, we can discover all frequent temporal patterns by constructing the projected database with the frequently extended prefixes and recursively running until the prefixes cannot be extended (Lines 13-18, algorithm 2.2).

Algorithm 2.2: *CTMiner* (*DB*, *min_sup*)

```
Input: DB: a temporal database, min_sup: the minimum support threshold
Output: TP: set of all frequent patterns in DB

01:  TP ← ∅;
02:  use incision_strategy transforming DB into coincidence representation;
03:  call CPrefixSpan (DB , ⟨⟩, min_sup, TP );
04:  output TP;

Procedure CPrefixSpan (DB|α , α , min_sup, TP )
05:  scan DB|α once, remove infrequent slices and find every frequent slice s such that:
06:  (i)  s can be assembled to the last coincidence of α;
      or (ii) ⟨s⟩ can be appended to α;
07:  for each frequent slice s do
08:        if s is a "finishing slice" then
09:              if exist corresponding starting slice in α then    // pre-pruning
10:                    append s to α to form α';
11:        if s is a "starting slice" or "intact slice" then
12:              append s to α to form α';
13:  for each α' do
14:        construct DB|α' with insignificant postfix elimination; // post-pruning
15:        if |DB|α' | ≥ min_sup then
16:              if α' is a temporal pattern then // all slices in α' appearing in pair
17:                    TP ← TP ∪ {α' };
18:        call CPrefixSpan (DB|α', α', min_sup, TP );
```

Fig. 2.7: CTMiner algorithm

Taking into account the property of event slice and coincidence, we propose two pruning strategies, pre-pruning and post-pruning to reduce the searching space efficiently and effectively. Firstly, the starting slices and finishing slices definitely occur in pairs in a coincidence sequence. We only require projecting the frequent finishing slices which have the corresponding starting slices in their prefixes (Lines 8-10, algorithm 2.2). It is called pre-pruning strategy which can prune off non-qualified patterns before constructing projected database.

Secondly, when we construct a projected database, some slices in postfix sequences need not be considered. With respect to a prefix sequence $\langle \alpha \rangle$, a finishing slice in a projected postfix sequence is called significant, if it has corresponding starting slices in $\langle \alpha \rangle$. When constructing the projected database $DB_{|\langle \alpha \rangle}$, only the significant slices in postfix sequences are collected. All insignificant slices are eliminated since they can be ignored in the discovery of frequent temporal patterns. The second pruning method is called post-pruning strategy which eliminates insignificant sequences when constructing projected database (Lines 13-14, algorithm 2.2).

Because of the post-pruning strategy, CPrefixSpan can not guarantee that the new coincidence sequences formed from appending previously discovered frequent sequences with locally frequent slices are always frequent. We require an additional computation to insure that the support count of the coincidence sequences in a projected database is no less than *min_sup* (Line 15, algorithm 2.2). Since $|DB_{|a}|$ (number of sequences in $DB_{|a}$) can be produced by using a simple counter when we project the database, the computation cost is nearly negligible. Finally, if all slices in a frequent coincidence sequence appear in pairs, i.e., every starting (finishing) slice has corresponding finishing (starting) slice, we can out this frequent coincidence sequence as a temporal pattern (Lines 16-17, algorithm 2.2). The experimental studies indicate that pre-pruning and post-pruning strategies can improve the performance in both computation time and memory usage efficiently.

Notice that, when scanning projected database to calculate the support count of an intact slice s, both s and starting slice s^{+} occurring in coincidence sequences need to be accumulated. Since the only difference between intact slice and starting slice is whether the event interval have been incised or not, both of them in the coincidence sequence imply the existence of an event interval.

But when counting the support of starting slice s^{+} or finishing slice s^{-}, only the occurrence of s^{+} or s^{-} in a database need to be accumulated. Same as database projection, when we construct the projection with respect to intact slice $\langle s \rangle$, we collect not only the sequence prefixed with $\langle s \rangle$, but also prefixed with $\langle s^{+} \rangle$ as the projected database.

Table 2.3: Example of projected databases and frequent temporal patterns

<table>
<tr><th>event sequences with corresponding coincidence representation</th><th>slice prefix</th><th>projected coincidence database
: insignificant</th><th>temporal patterns</th></tr>
<tr><td rowspan="7">S1: $\langle A^{+}(A^{-}B^{+}C^{+})B^{-}C^{-}D^{+}E\,D^{-} \rangle$
S2: $\langle B\,D^{+}(EF)\,D^{-} \rangle$
S3: $\langle A^{+}(A^{-}B^{+})B^{-}@\,D^{+}E\,D^{-} \rangle$
S4: $\langle B\,A\,D^{+}E\,D^{-} \rangle$

↓ infrequent slice elimination

S1: $\langle A^{+}(A^{-}B^{+})B^{-}\,D^{+}E\,D^{-} \rangle$
S2: $\langle B\,D^{+}\,E\,D^{-} \rangle$
S3: $\langle A^{+}(A^{-}B^{+})B^{-}@\,D^{+}E\,D^{-} \rangle$
S4: $\langle B\,A\,D^{+}E\,D^{-} \rangle$</td><td>$\langle A \rangle$</td><td>S1: $\langle (\_\,B^{+})\,B^{-}D^{+}E\,D^{-} \rangle$
S3: $\langle (\_\,B^{+})\,B^{-}@\,D^{+}E\,D^{-} \rangle$
S4: $\langle D^{+}E\,D^{-} \rangle$</td><td rowspan="2">$\langle A \rangle$
$\langle A\,D \rangle$
$\langle A\,E \rangle$
$\langle A\,D^{+}E\,D^{-} \rangle$
$\langle A^{+}(A^{-}B^{+})\,B^{-} \rangle$
$\langle A^{+}(A^{-}B^{+})\,B^{-}E \rangle$</td></tr>
<tr><td>$\langle A^{+} \rangle$</td><td>S1: $\langle (A^{-}B^{+})\,B^{-}D^{+}E\,D^{-} \rangle$
S3: $\langle (A^{-}B^{+})\,B^{-}@\,D^{+}E\,D^{-} \rangle$</td></tr>
<tr><td>$\langle B \rangle$</td><td>S1: $\langle D^{+}E\,D^{-} \rangle$
S2: $\langle D^{+}E\,D^{-} \rangle$
S3: $\langle @\,D^{+}E\,D^{-} \rangle$
S4: $\langle A\,D^{+}\,E\,D^{-} \rangle$</td><td rowspan="2">$\langle B \rangle$
$\langle B\,D \rangle$
$\langle B\,E \rangle$
$\langle B\,D^{+}E\,D^{-} \rangle$</td></tr>
<tr><td>$\langle B^{+} \rangle$</td><td>S1: $\langle B^{-}D^{+}E\,D^{-} \rangle$
S3: $\langle B^{-}@\,D^{+}E\,D^{-} \rangle$</td></tr>
<tr><td>$\langle D \rangle$</td><td>$\varnothing$</td><td rowspan="2">$\langle D \rangle$
$\langle D^{+}E\,D^{-} \rangle$</td></tr>
<tr><td>$\langle D^{+} \rangle$</td><td>S1: $\langle E\,D^{-} \rangle$
S2: $\langle E\,D^{-} \rangle$
S3: $\langle E\,D^{-} \rangle$
S4: $\langle E\,D^{-} \rangle$</td></tr>
<tr><td>$\langle E \rangle$</td><td>S1: $\langle D^{-} \rangle$
S2: $\langle D^{-} \rangle$
S3: $\langle D^{-} \rangle$
S4: $\langle D^{-} \rangle$</td><td>$\langle E \rangle$</td></tr>
</table>

We take the database in Table 2.2 with *min_sup* = 2 as an example. There are 17 event records which can be regarded as 4 event sequences in the database. After transforming, the event sequences with corresponding coincidence representation are shown as in first column in Table 2.3. We can find all the frequent slices with scanning database once. Since the pre-pruning strategy, we only require process the intact slices and starting slices as shown in second column in Table 2.3. We take the slice A^{+} and E as examples to further discuss in details. The projected database with respect to $\langle A^{+} \rangle$ has 2 sequences: $\langle (A^{-}B^{+})B^{-}D^{+}ED^{-} \rangle$ and $\langle (\mathrm{A}B^{+})B^{-}@\, D^{+}ED^{-} \rangle$. Continuing the recursive process with the $\langle A^{+} \rangle$ - projected database, we can discover all frequent temporal patterns prefixed with $\langle A^{+} \rangle$. In addition, when projecting intact slice $\langle E \rangle$, the generated postfix sequences will be eliminated by post-pruning strategy directly since $\langle D^{-} \rangle$ is insignificant.

Hence, we do not need to consider the $\langle E \rangle$ -projected database. The last column in Table 2.3 lists all generated temporal patterns.

2.5.2 Multi-Projection Technique

The projection approach partitions the data and the set of frequent patterns to be processed, and confines each process to the corresponding smaller projected database. This approach can reduce the search space effectively. For a frequent pattern, we only require searching its corresponding projected database for locally frequent items, and then append them to original pattern to form new frequent patterns.

However, the projection method is designed for traditional time point-based patterns mining. When mining the interval-based temporal patterns, the complex relationship between any two intervals will cause unanticipated result if we adopt projection approach directly without any modification. For example, as in Fig. 2.8(a), when projecting a time point-based sequence q_1 = $\langle (A)(B)(C)(A)(BD) \rangle$ with respect to a prefix $\langle (A)(B) \rangle$, a projected sequence q_1' = $\langle (C)(A)(BD) \rangle$ will be generated. The projected result q_1' is accurate since the relationship between any two time point-based events is just "*before*" and "*after*." The pairwise relations of first $(A)(B)$ and second $(A)(B)$ in s_1 are both (A before B). But the feature of time interval is quite different from that of time point; the pairwise relationships among intervals are more complex. For example, as in Fig. 2.8 (b), when projecting a coincidence sequence $q_2 = \langle (A^+)(B^+C)(A^-B^-)(A^+)(B^+D)(A^-)(B^-) \rangle$ with respect to a prefix $\langle (A^+)(B^+) \rangle$, only a projected sequence q_2' = $\langle (C)(A^-B^-)(A^+B^+D)(A^-)(B^-) \rangle$ is generated if we adopt projection approach without modification. Although the projected result looks promising, actually the revealed information is not sufficient. The first occurrence of $(A^+)(B^+)$ in q_2 implies the temporal relation between interval A and B is (A *finished-by* B), but the second occurrence of $(A^+)(B^+)$ in q_2 implies the temporal relation between interval A and B is (A *overlaps* B). Obviously, only q_2' does not present the projected result sufficiently. In this chapter, a new projection strategy, multi-projection, is proposed for time interval-based patterns mining to address this problem.

From conventional projection, the major difference of multi-projection lies in the postfixes generation and collection. For a given sequence x as prefix, the traditional projection method forms projected database from collection of postfixes of sequences in database with regards to x. The generation of postfixes only considers the first occurring position of x in sequences, as shown in Fig. 2.8(a). However, given a coincidence sequence y as prefix, the multi-projection method generates postfixes with regards to every occurring position of y in every sequence in database, and then collects all the generated postfixes to construct projected database. For example, in Fig. 2.8(b), multi-projecting a coincidence sequence q_2 with regard to a prefix $\langle(A^+)(B^+)\rangle$ will generate two postfix es q_2' and q_2''. Usually, large size of projected databases will be generated by multi-projection technique. With regards to a prefix, the more occurrences in a sequence, the more postfixes will be generated. The size of projected database is the crucial bottleneck in CTMiner since the major cost of algorithm is recursive database projection. If the number of generated postfixes can be reduced, the performance of temporal mining can be further improved.

prefix $x = \langle(\boldsymbol{A})(\boldsymbol{B})\rangle$

↓ **project**

time point sequence $q_1 = \langle\underline{(\boldsymbol{A})(\boldsymbol{B})}(\boldsymbol{C})(\boldsymbol{A})(\boldsymbol{BD})\rangle$

↓ **generate**

postfix sequence $q_1' = \langle(\boldsymbol{C})(\boldsymbol{A})(\boldsymbol{BD})\rangle$

(a) example of traditional projection

prefix $y = \langle(\boldsymbol{A}^+)(\boldsymbol{B}^+)\rangle$

↓ **project** ⇣ **project**

coincidence sequence $q_2 = \langle\underline{(\boldsymbol{A}^+)(\boldsymbol{B}^+}\boldsymbol{C})(\boldsymbol{A}^-\boldsymbol{B}^-)(\boldsymbol{A}^+)(\boldsymbol{B}^+\boldsymbol{D})(\boldsymbol{A}^-)(\boldsymbol{B}^-)\rangle$

↓ **generate**

postfix sequence $q_2' = \langle(\_\boldsymbol{C})(\boldsymbol{A}^-\boldsymbol{B}^-)(\boldsymbol{A}^+)(\boldsymbol{B}^+\boldsymbol{D})(\boldsymbol{A}^-)(\boldsymbol{B}^-)\rangle$

⇣ **generate**

postfix sequence $q_2'' = \langle(\_\boldsymbol{D})(\boldsymbol{A}^-)(\boldsymbol{B}^-)\rangle$

(b) example of multi-projection

Fig. 2.8: Example of projection and multi-projection technique

The pseudoprojection technique proposed by Pei et al. [30] is a good solution for reducing the size of projected database. Instead of performing physical projection, pseudoprojection registers the sequence-ID and the starting position of the projected postfix in the sequence. Then, a physical projection of a sequence is replaced by registering a sequence identifier and the projected position index point. With this technique, the usage of main memory can be reduced intrinsically. The implementation of multi-projection also utilizes pseudoprojection technique to avoid physically copying postfixes. Thus, we can promote both computation time and memory space efficiently. Our experimental result shows that the performance of multi-projection in both synthetic data and real data still scales well when processing considerable event sequences.

2.5.3 Correctness of Algorithm

The correctness of the CTMiner is proven as below.

Lemma 2.1 (Support property of projected database) *Let α and β be two temporal patterns in temporal database DB such that α is a prefix of β. The support of β in DB equals to the one in $DB_{|\alpha}$.*

Proof: As discussing in [30], we know that to collect support count of sequence β in *DB*, only the sequences in the *DB* sharing the same prefix α should be considered. Furthermore, only those suffixes with the prefix α being a supersequence of β should be counted. Hence, the support of β in *DB* equals to the one in $DB_{|\alpha}$.

Theorem 2.1 (Correctness of CTMiner) *The temporal patterns discovered from CTMiner are correct.*

Proof: By lemma 2.1, we realize that the CTMiner can enumerate the support count correctly. Therefore, if CTMiner says that the support of α is frequent and all event slices in α appearing in pairs, α is a temporal pattern.

2.6 Experimental Results and Performance Study

To evaluate the performance of CTMiner, four temporal pattern mining algorithms, ARMADA [35], H-DFS [27], IEMiner [29] and TPrefixSpan [36], were also implemented for comparison. All algorithms were implemented in C++ language and tested on a Pentium D 3.0 GHz with 2 GB of main memory running Windows XP system. The comprehensive performance study has been conducted on both synthetic and real world datasets. To show the efficiency of CTMiner, we perform four kinds of experiments. First, we compare the running time of CTMiner and other temporal pattern mining algorithms using synthetic datasets. We also show the distribution of pattern length with different thresholds. Second, we investigate the scalability and memory usage of CTMiner. Third, we discuss the improvement of runtime performance with proposed pruning strategies. Finally, we apply CTMiner in some real datasets to compare the performance and also discuss the practicability of temporal pattern mining.

2.6.1 Data Generation

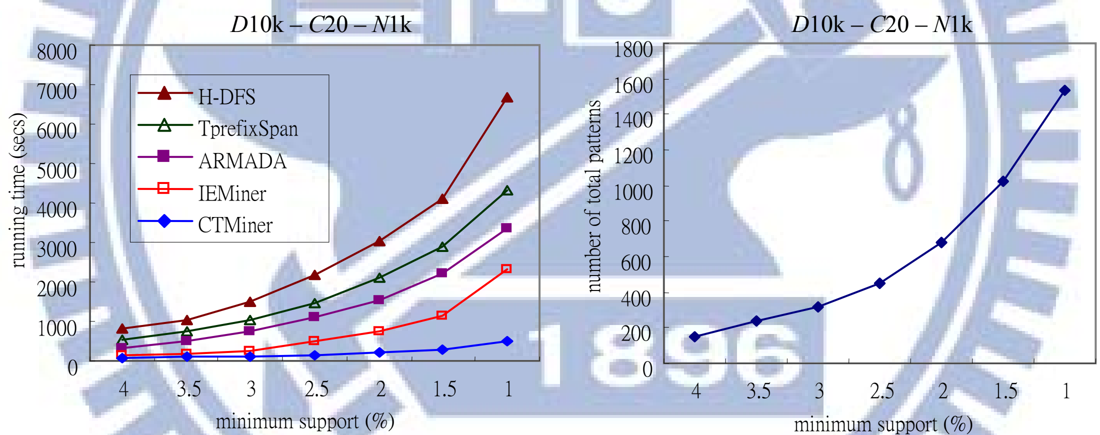
The synthetic data sets in the experiments are generated using synthetic generation program proposed by Agrawal et al. [1]. Since the original data generation program was designed to generate time point-based data, the generator for the temporal pattern mining algorithms requires modifications accordingly. The parameter setting of temporal data generator is shown in Table 2.4.

Table 2.4: Parameters of synthetic data generator

Parameters	Description
$\lvert D \rvert$	Number of event sequences
$\lvert C \rvert$	Average size of event sequences
$\lvert S \rvert$	Average size of potentially frequent sequences
N_S	Number of potentially frequent sequences
N	Number of event symbols

We first create a set of maximal potentially large sequences used in the generation of event sequences. The number of maximal potentially large sequence is N_S. A maximal potentially large sequence is generated by first picking the size from a Poisson distribution with mean equal to $|S|$. Then, we chose the event interval symbols in maximal potentially large sequence from N events randomly. Since the time interval in a sequence has duration, the data generator for temporal pattern mining algorithms requires an additional tuning for experimental data generation. We adopt the modification proposed by Wu et al. [36]. All the duration times of event intervals are classified into three categories: long, medium and short. The long, medium and short interval events are with an average length of 12, 8 and 4 respectively. For each event interval, we first randomly decide its category and then determine its length by drawing a value from a normal distribution.

Finally, we select the temporal relations between consecutive intervals randomly and form a maximal potentially large sequence. Since we adopt normalized temporal patterns [13], the temporal relationships can be chosen from the set {*before*, *meets*, *overlaps*, *is-finished-by*, *contains*, *starts*, *equal*}. After all maximal potentially large sequences are determined, we generate $|D|$ event sequences. Each event sequence is generated by first deciding its size, which was picked from a Poisson distribution with mean equal to $|C|$. Then, each event sequence is generated by assigning a series of maximal potentially large sequences.



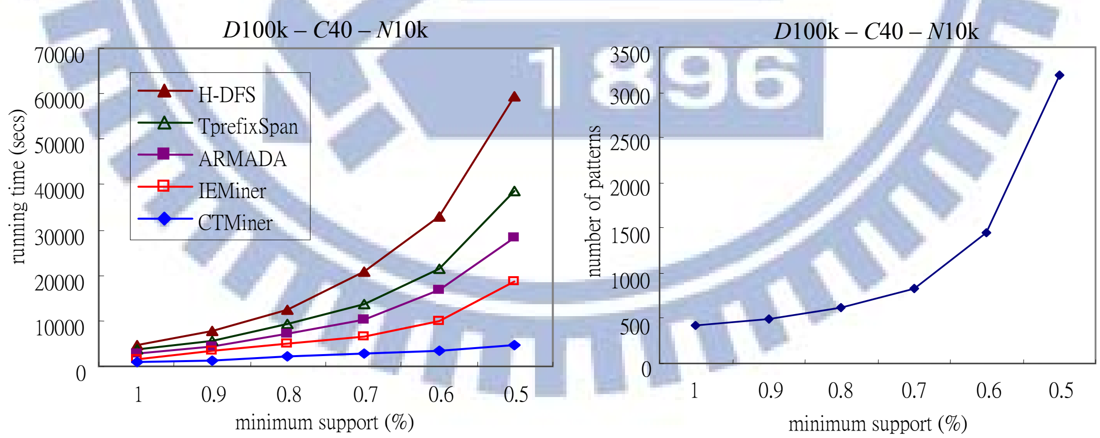
(a) Performance of the five algorithms (b) The number of temporal patterns

Fig. 2.9: Experimental results on dataset $D10k - C20 - N1k$

2.6.2 Runtime Performance on Synthetic Datasets

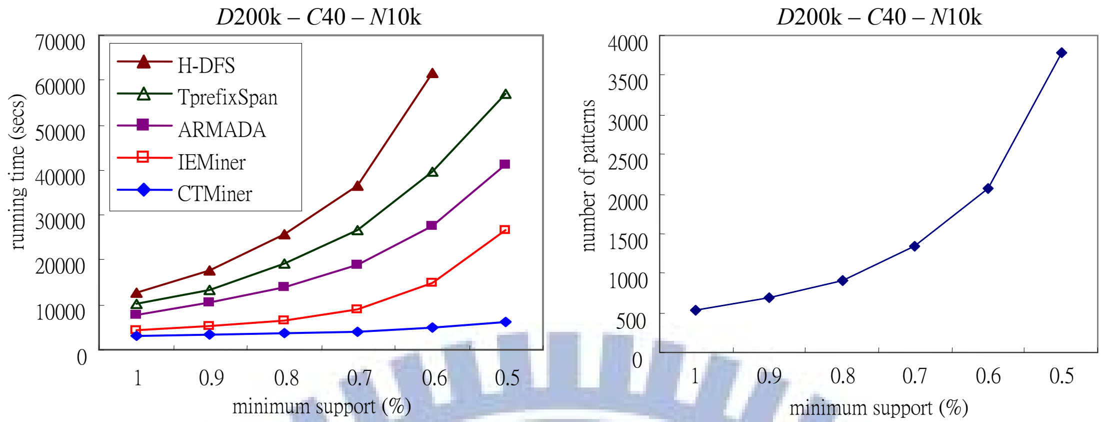
In all the following experiments, some parameters are fixed, i.e., $|S| = 4$ and $N_S = 5{,}000$. The other parameters are configured for comparing the temporal pattern mining algorithms. The first experiment of the five algorithms is on the dataset $D10k–C20–N1k$, which contains 10,000 event sequences, the average length of sequence is 20 and the number of events is 1,000. Fig. 2.9(a) and 2.9(b) show the processing time of the five algorithms and the number of generated temporal patterns at different support thresholds respectively. The minimum support thresholds vary from 1 % to 4 %. Obviously, when the minimum support value decreases, the processing time required for all algorithms increases. However, the runtime for ARMADA, H-DFS, IEMiner and TPrefixSpan increase drastically compared to CTMiner. When minimum support is 1 %, the data set contains a large number of temporal patterns. From the graph, we can observe that CTMiner is about 4.5 times faster than IEMiner, more than 6.6 times faster than ARMADA, about 8.5 times faster than TPrefixSpan and more than 13.1 times faster than H-DFS.

The second experiment is performed on dataset *D*100k–*C*40–*N*10k, which is much larger since it contains 100,000 event sequences, average length 40 and 10,000 event intervals. Fig. 2.10(a) and 2.10(b) show the running time and the number of generated temporal patterns at different support thresholds respectively. However, we vary the minimum support thresholds from 0.5 percent to 1 percent to generate larger number of frequent patterns from large data set. The data set contains a large number of temporal patterns when minimum support is reduced to 0.5 %. We can see that CTMiner is about 4 times faster than IEMiner, about 6 times faster than ARMADA, more than 8.2 times faster than TPrefixSpan and more than 12.6 times faster than H-DFS.



(a) Performance of the five algorithms (b) The number of temporal patterns

Fig. 2.10: Experimental results on dataset *D*100k – *C*40 – *N*10k



(a) Performance of the five algorithms (b) The number of temporal patterns

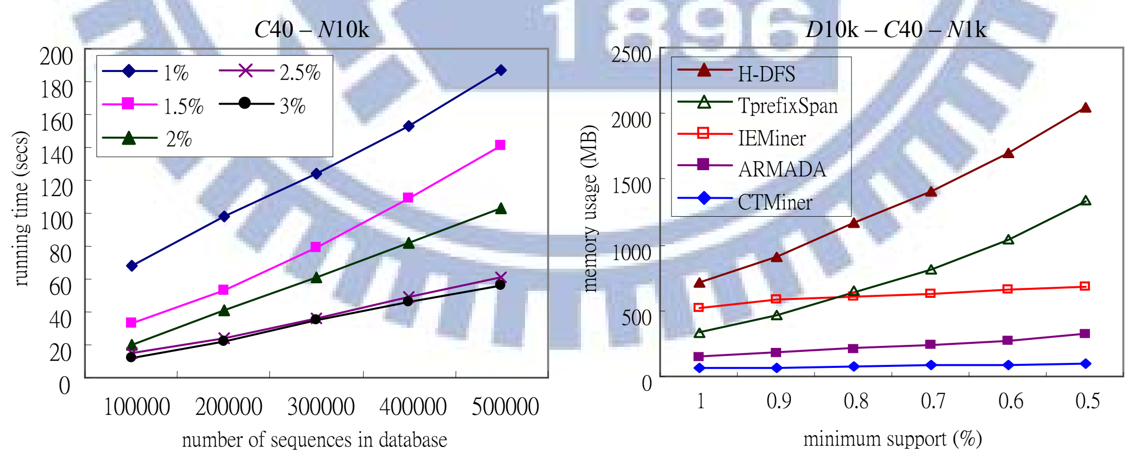
Fig. 2.11: Experimental results on dataset *D*200k – *C*40 – *N*10k

The third experiment is performed on dataset *D*200k–*C*40–*N*10k, which contains 200,000 event sequences, average length 40 and 10,000 event intervals. Fig. 2.11(a) and 2.11(b) show the running time and the number of generated temporal patterns at different support thresholds respectively. Same as second experiment, the minimum support thresholds vary from 0.5 percent to 1 percent. When minimum support is reduced to 0.5 %, CTMiner is more than 4.2 times faster than IEMiner, more than 6.5 times faster than ARMADA, about 9.1 times faster than TPrefixSpan, while H-DFS never terminates on our machine. The total experiments indicate that even with extremely low support and a large number of temporal patterns, CTMiner algorithm is still efficient and outperforms state-of-the-art algorithms.

2.6.3 Scalability and Memory Usage Studies

In this section, we study the scalability and memory usage of the CTMiner algorithm. Fig. 2.12(a) shows the results of scalability tests of the CTMiner algorithm, with the database size growing from 100K to 500K sequences, and with different minimum support threshold settings. Here, we use the data set *C*40–*N*10k which the average length of the sequence is 40 and the number of events in the database is 10,000 with varying different database size. As the size of database increases and minimum support decreases, the processing time of CTMiner increases, since the number of frequent patterns also increases. As can be seen, CTMiner is linearly scalable with different minimum support threshold. When the number of generated temporal patterns is large, the runtime of CTMiner still increases linearly with different database size.

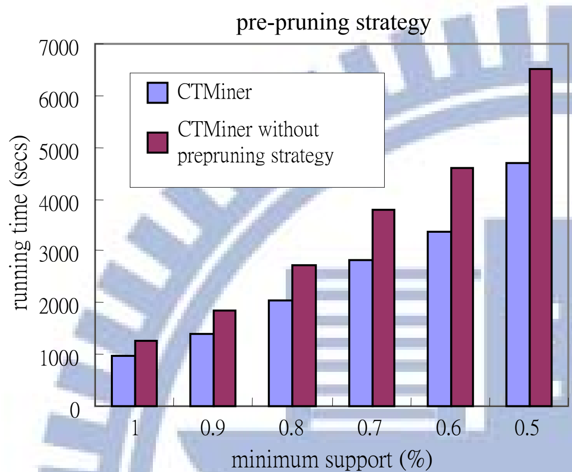
Then, we compare the memory usage among the five algorithms, ARMADA, CTMiner, H-DFS, IEMiner and TPrefixSpan, using synthetic data set *D*100k – *C*40 – *N*10k. Fig. 2.12(b) shows the results, from which we can observe that CTMiner is not only more efficient, but also more stable in memory usage than the other four algorithms. For example, when minimum support threshold is reduced to 1%, CTMiner consumes is about 3.4 times smaller than ARMADA, more than 7.1 times smaller than IEMiner, more than 13 times smaller than TPrefixSpan and about 21 times smaller than H-DFS. This also explains why in our previous performance tests when the support threshold becomes extremely low, why CTMiner is still efficient and outperforms state-of-the-art algorithms. Based on our analysis, CTMiner only needs memory space to hold the sequence data sets plus a set of header tables and pseudoprojection tables to construct projected databases. Although TPrefixSpan is also designed based on PrefixSpan, it still consumes memory space to hold the generated candidate sequences because of the complex relation among intervals. Both IEMiner and H-DFS need memory space to hold candidate sequences in each level. When the minimal support threshold drops, the set of candidate sequences grows up quickly, which results in memory consumption upsurging.



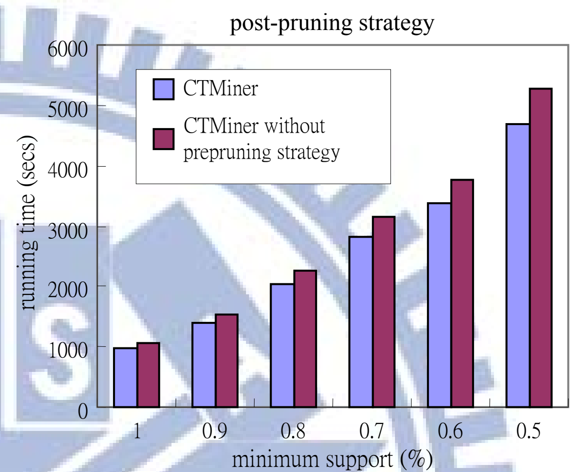
(a) CTMiner with different database size (b) Memory usage of five algorithms

Fig. 2.12: Experiments of scalability and memory usage

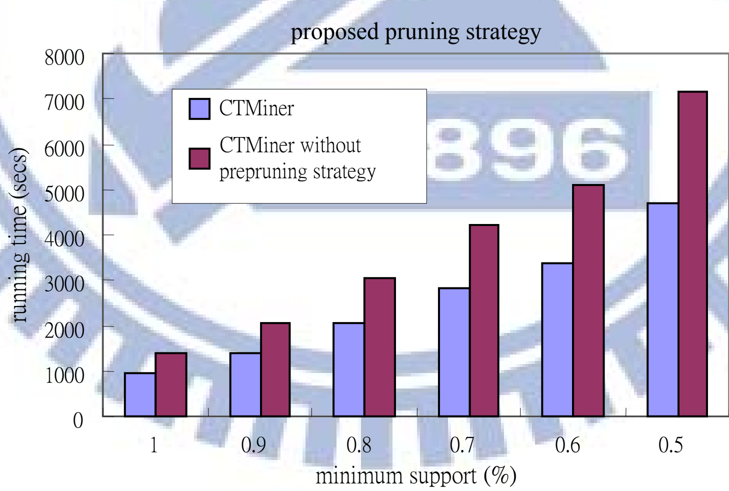
In summary, our performance study shows that CTMiner has the best overall performance among the four algorithms tested. The scalability study also shows that CTMiner scales well even with large databases and low thresholds. The memory usage analysis shows the efficient memory consumption of CTMiner and part of the reason why other algorithms become slow since the candidate sequences may consume a huge amount of memory.



(a) The performance testing of influence on pre-pruning strategy



(b) The performance testing of influence on post-pruning strategy



(c) The performance testing of influence on proposed pruning strategies

Fig. 2.13: The performance testing of influence on proposed pruning strategies

2.6.4 Influence of Proposed Pruning Strategies

In this section, to reflect the speedup of proposed pruning methods, we measure the CTMiner with two pruning strategies and without pruning strategy on time performance. The experiment is performed on the data set *D*100k–*C*40–*N*10k, which contains 100,000 event sequences, the average length of sequence is 40 and the number of events is 10,000. Fig. 2.13 is the results of varying minimum support thresholds from 0.5 percent to 1 percent. As shown in Fig. 2.13(a), pre-pruning strategy can improve 23.4% to 27.9% of the performance of CTMiner. Because of removing non-qualified slices before database projection, pre-pruning strategy can efficiently speedup the execution time. The impact of the post-pruning strategy is presented in Fig.2.13(b).

As can be seen from the graph, when CTMiner is without post-pruning strategy, the execution time is about 9.5% slower than CTMiner in average. We can find that post-pruning strategy can improve the performance of CTMiner by effectively eliminating all useless slices for temporal pattern construction. Fig. 2.13(c) depicts the influence on two proposed pruning strategies. We can see that CTMiner is constantly about 33% faster than the one without any pruning strategy. Nevertheless, the proposed pruning strategies not only effectively reduce the searching space but also efficiently improve the performance of CTMiner.

2.6.5 Real World Dataset Analysis

In addition to using synthetic data sets, we have also performed an experiment on real world datasets [18] to compare the performance and indicate the applicability of temporal pattern mining. We use five datasets for evaluation, as shown in Table 2.5. The origin and preprocessing steps of each dataset are briefly described as follows. For more details, please refer to [18].

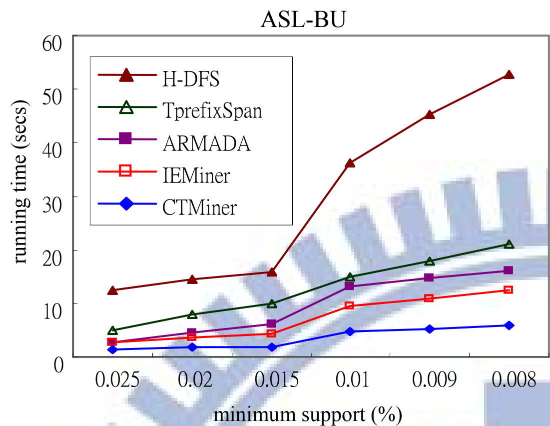
- *ASL-BU*: The intervals are transcriptions from videos of American Sign Language expressions provided by Boston University. It consists of observation interval sequences with labels such as head mvmt: nod rapid or shoulders forward.
- *ASL-GT*: The intervals are derived from numerical time series with features derived from videos of American Sign Language expressions. The numerical time series were discretized into 2-4 states. Each sequence represents one of 40 word like brown or fish.

Table 2.5: Five real-life databases

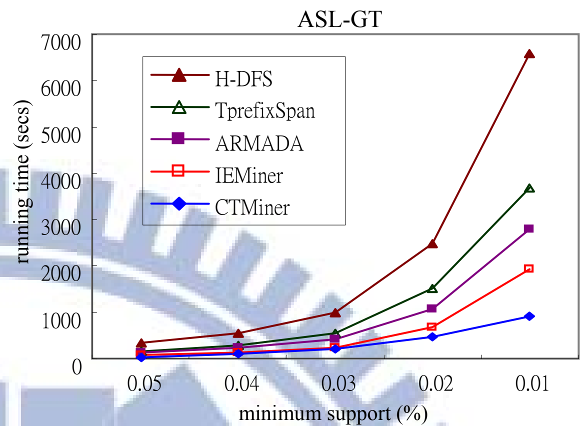
Database	Intervals	Labels	Sequences
ASL-BU	18,250	154	441
ASL-GT	89,247	47	3493
Pioneer	4,883	92	160
Auslan2	900	12	200
Library	549,071	206,844	28,339

- *Pioneer*: The intervals were derived from the Pioneer-1 datasets in the UCI repository. The numerical time series were discretized by choosing thresholds manually based on exploratory data analysis. Each sequence describes one of three scenarios: gripper, move, turn.
- *Auslan2*: The intervals were derived from the high quality Australian Sign Language dataset in the UCI repository. The dimensions were discretized using Persist and the median as the divider. Each sequence represents a word like girl or right.
- *Library*: We collect 1,098,142 library records (lending and returning) for three years from the National Chiao Tung University Library [6]. The database includes 206,844 books and 28,339 readers. An event interval is constructed by a book ID and corresponding lending and returning time. The size of database is the number of sequences in database (same as the number of readers, 28,339). The maximal and the average length of sequences are 262 and 38 respectively.

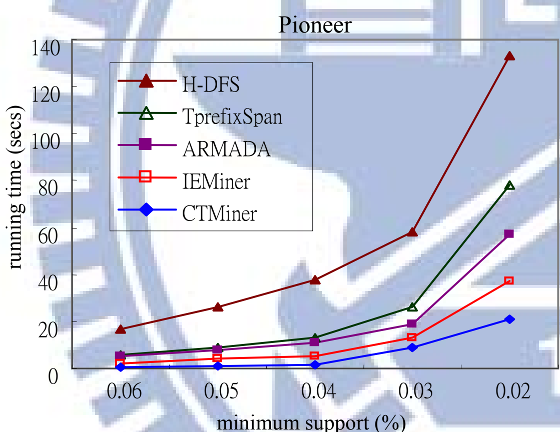
In Fig. 2.14 and Fig. 2.15, we show the execution time of five algorithms on all real datasets with varying minimum support thresholds. Obviously, all experiments indicate that even with extremely low support, CTMiner is still efficient and outperforms all other mining algorithms, especially, with large datasets, such as Library. As can be seen from Fig 2.15(a) and 2.15(b), when the minimum support is greater than 0.1 %, most of the generated temporal patterns are with length one or two. As the minimum support drops down to 0.05 %, there are 14,549 temporal patterns and the execution time of CTMiner is about 1.7 times faster than IEMiner, more than 3 times faster than ARMADA, about 4.2 times faster than TPrefixSpan and H-DFS has never terminated.



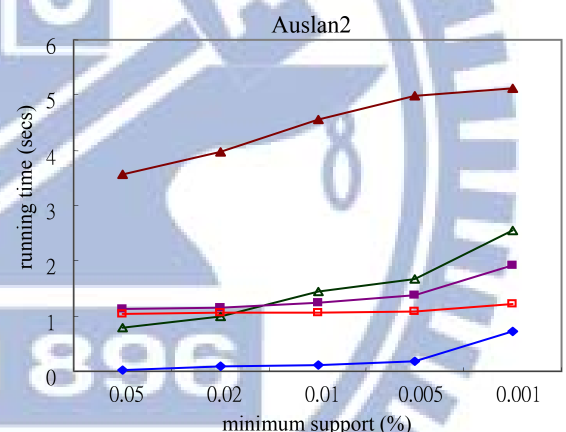
(a) Performance of five algorithms on ASL-BU dataset



(b) Performance of five algorithms on ASL-GT dataset

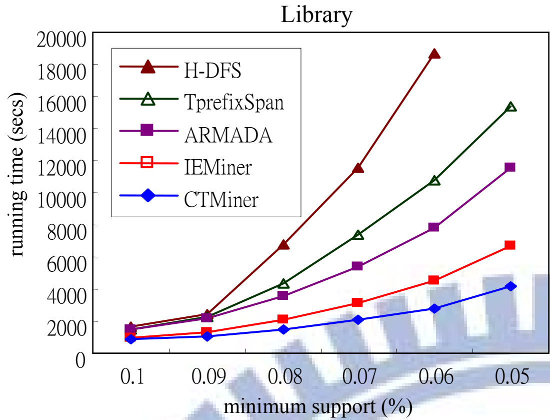


(c) Performance of five algorithms on Pioneer dataset

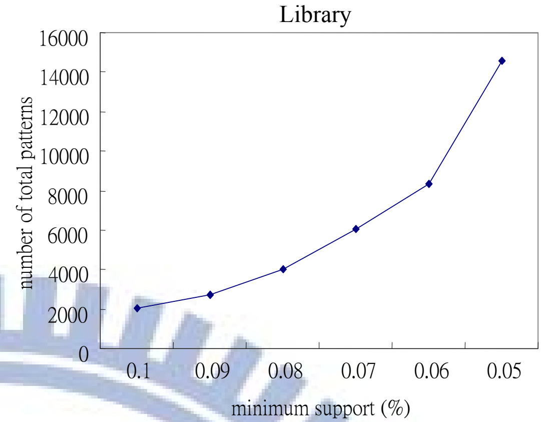


(d) Performance of five algorithms on Auslan2 dataset

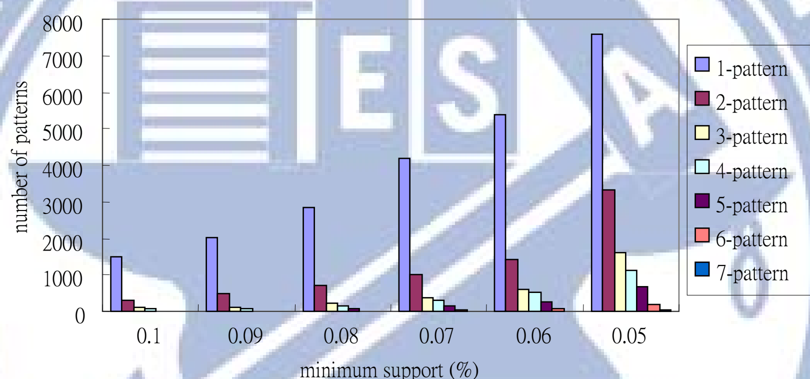
Fig. 2.14: Experimental results on ASL-BU, ASL-GT, Pioneer, and Auslan2



(a) Performance of five algorithms on Library dataset



(b) The number of generated temporal patterns on Library



(c) Distribution of temporal patterns

Fig. 2.15: Experimental results on Library dataset

Finally, to show the practicability of temporal patterns, we applied the CTMiner algorithm in book lending dataset to extract the compact reader's behaviors. This kind of information would be more helpful than conventional sequential pattern for reader recommendation. Table 2.6 illustrates some temporal patterns (part of mining results) discovered from the NCTU library. We take pattern 1 and 2 as examples. Suppose two readers, Mary and Sue, both check out the books "*The Know-It-All*" and "*The Curious Incident of the Dog in the Night-time*", if Mary check out two books at the same time, the library can send her an e-mail to notify that the book "*The Hitchhiker's guide to the galaxy*" is still on shelf or the book "*The Restaurant at the End of the Universe*" will be returned by 23rd of June, 2011. But if Sue checks out two books at different times, the library may send an e-mail to her to notify the information of the books "*Le Cosmicomiche*" or "*The One Hundred Years of Solitude*".

Table 2.6: Some temporal patterns discovered from of NCTU library

PID	temporal patterns	support
1	"The Know-It-All" "The Curious Incident of the Dog in the Night-time" "The Hitchhiker's guide to the galaxy" "The Restaurant at the End of the Universe"	163 (0.57%)
2	"The Know-It-All" "The Curious Incident of the Dog in the Night-time" "Le Cosmicomiche" "The One Hundred Years of Solitude"	43 (0.15%)
3	"The Homed Man" "Corazón tan blanco" "Ut og stjaele hester"	109 (0.38%)
4	"Magic Toyshop" "Wise Children" "Nights at the Cirus" "Burning Your Boats"	97 (0.34%)
5	"Palaverers" "Closely Watched Trains" "Vita nuova"	88 (0.31%)
6	"The End of the Affair" "I Served the King of England" "The Pearl in the Deep"	92 (0.32%)
7	"The End of the Affair" "I Served the King of England" "The Inheritance of Loss"	35 (0.12%)

To show the phenomena of pattern 1 and 2 are not just an anecdote, we discuss the case why readers, lending the same two books at different time, may have totally different interest. We find that the books "*The Know-It-All*" and "*The Curious Incident of the Dog in the Night-time*" are placed side by side on the shelf in NCTU Library. The author of "*The Curious Incident of the Dog in the Night-time*" has mentioned the books "*The Hitchhiker's guide to the galaxy*" and "*The Restaurant at the End of the Universe*" several times in the article. Hence, this can explain the expression from pattern 1 in Table 9, i.e., 0.57% readers who check out two books together will lend other two books later.

Moreover, we analyze the readers with behavior as pattern 2 in Table 9 and observe that all of them have taken an optional course, *Discussion of Human Relationship in Modern Society from Literature*. In this class, the first and second reading assignments are "*The Know-It-All*" and "*The Curious Incident of the Dog in the Night-time*", respectively. The final report is the discussion of alienation and antagonism between people from "*The One Hundred Years of Solitude*." This is the reason why these 43 students have the lending behavior as pattern 2.

From this example, we show the practicability of temporal pattern mining. We also can perceive that temporal patterns can promise a more expressive result to extract correlations among event data than conventional sequential patterns.

2.7 Summary

Mining temporal patterns from time interval-based data is a difficult problem since the processing for complex relations among intervals may require generating and examining large amount of intermediate subsequences. In this chapter, a novel technique, **incision strategy** and a new representation, **coincidence representation** are proposed to remedy this critical issue. We simplify the processing of complex relations among event intervals effectively. Coincidence representation is nonambiguous and has several advantages over existing representations.

Based on coincidence representation, we develop an efficient algorithm, **CTMiner** to discover frequent temporal patterns without candidate generation. The algorithm further employs two proposed pruning techniques to reduce the search space effectively. By analyzing the differences between mining sequential patterns and temporal patterns, we also propose a new projection technique, **multi-projection** to correctly project a database into a set of smaller projected databases. The experimental studies indicate that CTMiner is efficient and scalable. Both running time and memory usage of CTMiner outperform state-of-the-art algorithms.

To the best of our knowledge, most previous extensions of mining sequential pattern only focus on time point-based data. Little attention has been paid to the related extension studies of mining temporal patterns from time interval-based data. The major reason is the complex relation among intervals. In this chapter, we utilize proposed coincidence representation to overcome this problem and facilitate the processing. Hence, based on coincidence representation, there are many interesting extensions that may be studied further, such as mining closed and maximal temporal patterns, incremental temporal patterns maintaining, and method toward data stream.

Chapter 3

An Efficient Algorithm for Mining Closed Temporal Patterns from Interval Database

3.1 Introduction

Recently, sequential pattern mining is an active research topic in data mining for its wide applications such as customer analysis, network intrusion detection, discovery of tandem repeats in DNA sequences, study of scientific and medical processes, to name a few. Many efficient algorithms [1, 3, 6, 10, 11, 18, 20, 21, 30, 32, 39] proposed so far have good performance for discovering complete-set sequential patterns. But when mining long frequent sequences, or when using low support thresholds, the performance of such algorithms usually degrade dramatically. For example, assume a database contains only one long sequence $\{\langle(a_1)(a_2)(a_3)\ldots(a_{100})\rangle\}$. If the minimum support is 1, in the complete-set frequent pattern mining, there will be $(2^{100} - 1)$ frequent patterns: $\{\langle a_1\rangle{:}1, \langle a_2\rangle{:}1, \ldots, \langle(a_1)(a_2)(a_3)\ldots(a_{100})\rangle{:}1\}$. All of them except $\langle(a_1)(a_2)(a_3)\ldots(a_{100})\rangle{:}1$ are redundant, since all the other frequent patterns and their supports can be derived from this pattern.

Undoubtedly, a long sequential pattern usually contains an explosive number of subsequences and using low support threshold often bears huge number of computations. When a user or an application only needs the longest or more expressive sequential pattern, closed pattern mining algorithm may be a better alternative. We can avoid exhaustive enumeration of all frequent sequences and thus improve the performance. Hence, the mining of closed sequential patterns has attracted researchers for its capability of using compact results to preserve the same expressive power as complete-set frequent patterns mining.

Previous researches of closed sequential pattern mining [4, 5, 15, 34, 38] mainly focus on time point-based data. There has been no efficient method developed for mining closed sequential pattern from time interval-based data. However, in many real world scenarios, some events, which intrinsically tend to persist for periods of time instead of instantaneous occurrences, cannot be treated as "time points". In such cases, the data is usually a sequence of events with both start and finish times. Examples include library lending, stock fluctuations, patient diseases, and meteorology data. Actually, discovering closed sequential patterns from time interval-based data can reveal more interesting patterns. For example, in the medical field, the simple ordered sequence of events such as "fever → cough → headache," may be inadequate to express the complex relationships among symptoms. If we consider the duration time of events, some relationships can be mined from clinical records of patients to study the correlations between the symptoms and the diseases, or the influences between the diseases and other diseases. One may find that "in the case of myocardial infarction, chest pain usually contains the cardiac enzymes increasing." Another discovery might be that "in many tuberculosis patients, the presence of coughing up blood usually overlaps intermittent fever."

Existing time point-based approaches are hampered by the fact that they can only handle instantaneous events efficiently, not event intervals. We can perceive that time point-based issue is just a special case of the time interval-based issue (where start time is identical to finish time), but not vise versa. Mining closed time interval-based patterns (also referred to as **closed temporal patterns**) is more complex and arduous, and requires a different approach from mining time point-based data. So far, little effort has been paid to the issue of mining closed time interval-based sequential patterns. This is partly because of the complicated relationship among event intervals. Since the feature of time interval is quite different from time point, the pairwise relationships between any two time interval-based events are intrinsically complex. This complex relation is really a crucial bottleneck when we endeavor to design an efficient and effective algorithm for mining closed temporal patterns, since the complex relations may lead to generate larger number of candidate sequences and workload for counting the support of a candidate sequence.

Allen's 13 temporal logics [2] are comprehensively used to describe the relations among intervals, as shown in Fig. 2.1. Considering the arrangements of the start and the finish endpoints, there are 13 temporal relations between any two event intervals as: "*before*," "*after*," "*overlap*," "*overlapped by*," "*contain*," "*during*," "*start*," "*started by*," "*finish*," "*finished by*," "*meet*," "*met by*," and "*equal*." However, all the Allen's logics are binary relation and may suffer several problems when describing relationships among more than three event intervals. An appropriate representation is very crucial for facing this circumstance. Various representations [8, 13, 16, 24, 25, 29, 36] have been proposed but most of them have restriction on either ambiguity or scalability.

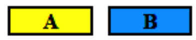
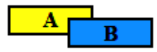
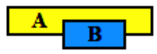
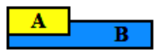
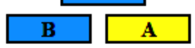
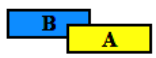
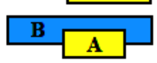
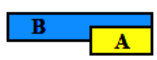
Temporal Relation	Pictorial Example	Endpoints Constraint (*s*: start time, *f*: finish time)
A before *B*		$A.f < B.s$
A overlaps *B*		$(A.s < B.s) \wedge (A.f > B.s) \wedge (A.f < B.f)$
A contains *B*		$(A.s < B.s) \wedge (A.f > B.f)$
A starts *B*		$(A.s = B.s) \wedge (A.f < B.f)$
A finished-by *B*		$(A.s > B.s) \wedge (A.f = B.f)$
A meets *B*		$A.f = B.s$
A equal *B*		$(A.s = B.s) \wedge (A.f = B.f)$
A after *B*		$B.f < A.s$
A overlapped-by *B*		$(B.s < A.s) \wedge (B.f > A.s) \wedge (B.f < A.f)$
A during *B*		$(B.s < A.s) \wedge (B.f > A.f)$
A started-by *B*		$(B.s = A.s) \wedge (B.f < A.f)$
A finishes *B*		$(B.s > A.s) \wedge (B.f = A.f)$
A met-by *B*		$B.f = A.s$

Fig. 3.1: Allen's 13 relations between two intervals

The contributions of this chapter are as follow:

- We simplify the processing of complex relations among intervals by capturing the global information of all endpoints in a sequence. Comparing with the complex relations between intervals, the relations among endpoints are simple, i.e., only "*before*," "*after*" and "*equal*."
- Various existing representations may lead to different kinds of problem. We develop a compact representation, **endpoint representation**, to express a pattern or sequence nonambiguously. Endpoint representation can facilitate the process and improve the performance of algorithm.

- A novel algorithm, **CEMiner**, which stands for **Closed Endpoint Temporal Miner**, is proposed to discover closed temporal patterns efficiently and effectively. Furthermore, CEMiner employs some optimization strategies to reduce the search space and avoids nonpromising closure checking and database projection.

Experimental studies on both synthetic and real datasets indicate that proposed strategy and algorithm are both efficient and scalable and outperforms the state-of-the-art algorithms. Our experiments also show that the proposed approach consumes a much smaller memory space. The remainder of this chapter is organized as follows. Section 3.2 and 3.3 provide the related work and some preliminaries, respectively. Section 3.4 introduces the endpoint representation. Section 3.5 describes the CEMiner algorithm. Section 3.6 gives the experiments and performance study, and we summarize in Section 3.7.

3.2 Related Works

CloSpan [38] is the first algorithm for mining closed sequential patterns from time point data. It generates a set of closed sequence candidates and then do post-pruning to discover closed sequential patterns. Although it performs two pruning methods to reduce search space, it still consumes much memory to maintain the set of historical closed sequence candidates.BIDE [34] is a fast algorithm for mining closed sequential patterns. Different from CloSpan, it uses a sequence closure checking scheme to avoid the maintenance of closed candidate sequence. The Proposed BackScan pruning method can prune the search space more aggressively than the methods used in CloSpan. COBRA [15] is a two-phased mining algorithm. It first finds all closed frequent itemsets [40], and then extends search space with only these frequent closed itemsets. Because COBRA uses both vertical and horizontal database formats to reduce the searching time in mining process, the memory usage is a major problem.

Some recent works have investigated the mining of complete-set temporal patterns. Kam et al. [16] designed an algorithm that uses the hierarchical representation to discover temporal patterns. However, the hierarchical representation is ambiguous and many spurious patterns are found. Hoppner [13] defined the supporting level of a pattern as the total time in which the pattern can be observed within a sliding window. But the algorithm needs to scan the database repeatedly, which would significantly lower its efficiency.

H-DFS [27] was proposed to discovery frequent arrangements of event intervals. This approach transforms an event sequence into a vertical representation using id-lists. Hence, H-DFS does not scale well when the temporal pattern length increases. TSKR [24] expressed the temporal concepts of coincidence and partial order for interval patterns. The pattern represented in this format is easily understandable but may reveal the relationship between pairwise event intervals in a pattern ambiguously. Based on MEMISP [20], an algorithm ARMADA [35] is proposed to find frequent temporal patterns from large database. This approach only needs two database scans and does not require candidate generation or database projection. Wu et al. [36] devised a nonambiguous expression, temporal representation, and TPrefixSpan algorithm to discover frequent temporal patterns. Although this algorithm only need two scans of the database, it does not employ any pruning strategy to reduce the search space.

Patel et al. [29] utilized additional counting information to achieve a lossless hierarchical representation, named Augmented Representation, and proposed an algorithm, IEMiner. Although IEMiner uses some optimization strategies to reduce the search space and remove nonpromising candidate sequences, it still has to scan database multiple times. HTPM [37] was developed for mining hybrid temporal patterns from event sequences, which contain both point-based and interval-based events. A new robust representation, SIPO [25], utilizes the boundaries of interval and further considers the noise tolerance to express relationships among intervals. The mining algorithm requires discovering both closed sequential pattern and closed itemset. Based on a compact representation, coincidence representation, CTMiner [8] is an efficient algorithm for mining temporal patterns. Algorithm also proposed some pruning strategies to significantly reduce the search space.

All of these algorithms only focus on mining closed sequential patterns from time point-based data or mining temporal patterns from time interval-based data. No effort has been put to closed temporal pattern. In this chapter, we discuss and design an efficient method to discover closed temporal patterns from interval-based data.

3.2 Preliminary

Definition 3.1 (Event interval and event sequence)

Let $E = \{e_1, e_2, \ldots, e_k\}$ be the set of event symbols. Without loss of generality, we define a set of uniformly spaced time points based on the natural number N. We say the triplet $(e_i, s_i, f_i) \in E \times N \times N$ is an event interval, where $e_i \in E$, $s_i, f_i \in N$ and $s_i < f_i$. The two time points s_i, f_i are called endpoints of an event interval, where s_i is the starting endpoint and f_i is the finishing endpoint. The set of all event intervals over E is denoted by I. An event sequence is a series of event interval triplets $\langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_n, s_n, f_n)\rangle$, where $s_i \leq s_{i+1}$, and $s_i < f_i$.

Definition 3.2 (Temporal database)

Considering a database $DB = \{r_1, r_2, \ldots, r_m\}$, each record r_i, where $1 \leq i \leq m$, consists of a sequence-id, *SID* and an event interval (i.e. an event symbol, a starting endpoint, and a finishing endpoint, where starting time < finishing time). *DB* is called a temporal database.

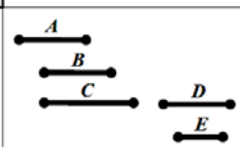
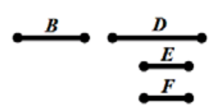
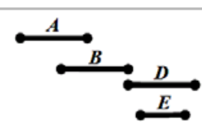
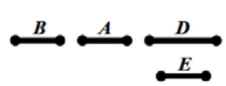
SID	event symbol	start time	finish time	event sequence	endpoint representation
1	*A*	2	7		$A^+(B^+C^+)A^-B^-C^-D^+E^+E^-D^-$
1	*B*	5	10		
1	*C*	5	12		
1	*D*	16	22		
1	*E*	18	20		
2	*B*	1	5		$B^+B^-D^+(E^+F^+)(E^-F^-)D^-$
2	*D*	8	14		
2	*E*	10	13		
2	*F*	10	13		
3	*A*	6	12		$A^+B^+A^-(B^-D^+)E^+E^-D^-$
3	*B*	7	15		
3	*D*	15	20		
3	*E*	17	19		
4	*B*	8	16		$B^+B^-A^+A^-D^+E^+E^-D^-$
4	*A*	18	21		
4	*D*	24	29		
4	*E*	25	27		

Fig. 3.2: An example database

Actually, if all records in *DB* with the same client-id are grouped together and ordered by nondecreasing start time, the database can be transformed into a collection of event sequences. As a result, the database *DB* can be viewed as a collection of event sequences. As in Fig. 3.2, example database consists of 17 event intervals, and 4 event sequences.

3.3 Endpoint Representation

The time interval-based mining problem is much more difficult than time point-based mining issue. Since the time period of the two intervals may overlap, the relation among event intervals is intrinsically more complicated than that of the event points. Allen's 13 temporal logics [2], in general, are adopted to describe the relations among intervals, as shown in Fig. 1. Unfortunately, when describing relationships among more than three events, Allen's temporal logics may suffer several problems.

A suitable representation is very important for describing a temporal pattern. As mentioned above, various representations have been proposed but most of them have restriction on either ambiguity or space usage. In this chapter, a new expression, endpoint representation is proposed to address the ambiguous and scalable problem.

Definition 3.3 (Endpoint sequence)

Given an event sequence $q = \langle (e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_i, s_i, f_i), \ldots, (e_n, s_n, f_n) \rangle$, $T_q = \{ s_1, f_1, s_2, f_2, \ldots, s_i, f_i, \ldots, s_n, f_n \}$ is a set of all endpoints in q. After sorting T in nondecreasing order, an endpoint sequence $q_e = \langle t_1, t_2, \ldots, t_{2n} \rangle$ can be derived by representing s_i and f_i as e_i^+ and e_i^-, respectively. Note that we use the parenthesis to indicate the times of endpoints are the same. To deal with multiple occurrences of events, we attach **occurrence number** to endpoint to distinguish multiple occurrences of the same event type in an endpoint sequence.

For example, in Fig. 3.2, an event sequence with *SID* 2 is $\langle (B, 1, 5), (D, 8, 14), (E, 10, 13), (F, 10, 13) \rangle$ and its corresponding endpoint sequence is $\langle\ B^+\ B^-\ D^+\ (E^+\ F^+)\ (E^-\ F^-)\ D^-\ \rangle$. An endpoint sequence q_e is also called the **endpoint representation** of q. $\langle A_1^+ B_1^+ (B_1^- D^+)\ D^- (A_1^- B_2^+)\ B_2^- A_2^+ A_2^- \rangle$ is an endpoint sequence with occurrence number where both event A and B occur twice. For a temporal database DB, by Definition 3.3, we can transform it into a set of tuples $\langle SID$,

$q_e\rangle$ where *SID* is the sequence-id of each event sequence q in *DB*, q_e is the endpoint representation of q. For example, in Fig. 3.2, we can transform four event sequences into corresponding endpoint sequences. For readability, in this chapter, we suppose that all temporal database mentioned later have been transformed into endpoint representation.

Temporal Relation	Temporal Relation (Inversed)	Pictorial Example	Endpoint representation
A before B	*B after A*	A B	$(A^+)(A^-)(B^+)\ (B^-)$
A overlaps B	*B overlapped-by A*	A B	$(A^+)(B^+)(A^-)(B^-)$
A contains B	*B during A*	A B	$(A^+)(B^+)(B^-)(A^-)$
A starts B	*B started-by A*	A B	$(A^+B^+)(B^-)(A^-)$
A finished-by B	*B finishes A*	A B	$(A^+)(B^+)(A^-B^-)$
A meets B	*B met-by A*	A B	$(A^+)(A^-B^+)(B^-)$
A equal B	*B equal A*	A B	$(A^+B^+)(A^-B^-)$

Fig. 3.3: The endpoint representation of Allen's 13 relations between two intervals

The endpoint representation has several benefits, and the most important one is that it can simplify the processing of complex pairwise relationships among all intervals efficiently. It utilizes the arrangement of endpoints as defined in Definition 3.3, and considers the information of entire event sequence instead of individual event interval. Given two different event intervals *A* and *B*, the endpoint representation of Allen's 13 relations between *A* and *B* is categorized as in Fig. 3.3. The three major merits of proposed representation are discussed as follows,

- **Scalability:** We only require 2*k* space for describing a *k*-interval temporal pattern, since each interval has two endpoints. Comparing with other representations, the endpoint representation still scales well even if plenty of intervals appear in a pattern.
- **Nonambiguity:** According to [5], we can find that the endpoint representation has no ambiguous problem. First, by Definition 3.3, a unique endpoint sequence can be built by transforming every event sequence into endpoint representation. In other words, the temporal relations among intervals can be mapped one-to-one to an endpoint sequence. Second, in an endpoint sequence, the order relation of the starting and finishing endpoints of *A* and *B* can be categorized as shown in Fig. 3.3. We can infer the original temporal relationships between intervals *A* and *B* nonambiguously.
- **Simplicity:** Obviously, the complex relations between intervals are the major bottleneck of closed temporal pattern mining since the mining may need to generate or examine explosive number of intermediate subsequences. The relation between two endpoints is simple, just "*before*," "*after*" and "*equal*." The simpler the relations, the less number of intermediate candidate sequences are generated and processed. Therefore, with endpoint representation, we can discover closed temporal patterns more efficiently.

3.4 CEMiner

We focus on the discussions of closed temporal pattern mining due to the widespread applicability of this technique and the lack of research on this topic. In this chapter, we develop a new algorithm, called **CEMiner** (standing for Closed Endpoint temporal Miner), to discover closed temporal patterns efficiently. CEMiner utilizes the arrangement of endpoints to accomplish the closed temporal pattern mining. In section 3.4.1, we outline the main idea of closure checking to assure a temporal pattern is closed or not. Section 3.4.2 details the algorithm and also discusses some pruning mechanisms for reducing the search space effectively.

Before introducing the algorithm, we give some definitions first. Let α be an endpoint sequence in a temporal database *DB* in endpoint representation. The α - projected database, denoted as $DB_{|\alpha}$, is the collection of postfixes of sequences in *DB* with regards to prefix α.

Considering two endpoint sequence $\alpha = \langle a_1, a_2, \ldots, a_n\rangle$ and $\beta = \langle b_1, b_2, \ldots, b_m\rangle$, α is called a subsequence of β, denoted as $\alpha \sqsubseteq \beta$, if there exist integers $1 \le i_1 \le i_2 \le \ldots \le i_n \le m$ such that $a_1 \subseteq b_{i1}, a_2 \subseteq b_{i2}, \ldots, a_n \subseteq b_{in}$. We also call β a supersequence of α, and β contains α. If β contains α and their supports are the same, we call β absorbs α.

Definition 3.4 (Closed temporal pattern)

Given a temporal database *DB* in endpoint representation, a tuple $\langle SID, q_e\rangle$ is said to contain an endpoint sequence α, if α is a subsequence of q_e. The support of an endpoint sequence α in *DB* is the number of tuples in the database containing α, i.e., $\boldsymbol{support(\alpha) = |\{\langle SID, q_e\rangle \mid (\langle SID, q_e\rangle \in DB) \wedge (\alpha \sqsubseteq q_e)\}|}$. Given a positive integer *min_sup* as the support threshold, the set of temporal patterns, ***TP***, includes all the endpoint sequences whose supports are no less than *min_sup* and all endpoints in sequences appear in pairs. The set of closed temporal patterns is defined as follows, $\boldsymbol{CTP = \{(\alpha \in TP) \wedge (\nexists \beta \in TP)\ such\ that\ (\alpha \sqsubseteq \beta) \wedge (support(\alpha) = support(\beta))\}}$.

Let database in Fig. 3.2 with *min_sup* = 2 be an example. The endpoint sequence $\langle A^+B^+A^-B^-\rangle$ is a temporal pattern since it occurs in sequence 1 and 3 (*support* = 2 ≥ *min_sup*) and each starting endpoint has corresponding finishing endpoint. $\langle A^+B^+A^-\rangle$ is a frequent endpoint sequence but not a temporal pattern, since B^+ does not have corresponding finishing endpoint. The endpoint sequence $\langle A^+B^+A^-B^-\rangle$ is not a closed temporal pattern since it is absorbed by $\langle A^+B^+A^-B^-E^+E^-\rangle$. That means $\langle A^+B^+A^-B^-\rangle \sqsubset \langle A^+B^+A^-B^-E^+E^-\rangle$ and both *support* = 2.

3.4.1 Closure Checking

To verify a new closed temporal pattern *p*, we require checking whether *p* is a sub-sequence or super-sequence of an existing temporal pattern *p*' and the projected database of *p* and *p*' is equal. This operation is also called **closure checking** and is very critical when mining closed temporal patterns. The performance of an algorithm usually hinged on whether the closure checking is well-designed. By borrowing the idea of the BI-Directional Extension [17], the closure checking of CEMiner algorithm is developed in order to discover closed temporal patterns efficiently, which are represented with endpoint representation.

Definition 3.5 (Forward-extension and backward-extension)

Given an endpoint sequence $\alpha = \langle a_1, a_2, \ldots, a_n\rangle$ in a temporal database *DB*, if α is non-closed, there must exist at least one endpoint x, which can be used to extend α to a new endpoint sequence α', which has the same support, i.e., *support* (α) = *support* (α'). α can be extended in five ways: (1) $\alpha' = \langle a_1, a_2, \ldots, a_n \cup x\rangle$; (2) $\alpha' = \langle a_1, a_2, \ldots, a_n, x\rangle$; (3) $\alpha' = \langle x, a_1, a_2, \ldots, a_n\rangle$; (4) $\exists i$, $1 \le i < n$, $\alpha' = \langle a_1, a_2, \ldots a_i \cup x, a_{i+1}, \ldots, a_n\rangle$; (5) $\exists i$, $1 \le i < n$, $\alpha' = \langle a_1, a_2, \ldots a_i, x, a_{i+1}, \ldots, a_n\rangle$. In cases (1) and (2), x occurs after all endpoints in α, we call x a **forward-extension endpoint** and α' a **forward-extension sequence** w.r.t. α. In cases (3), (4) and (5), x occurs before the last endpoint in α, we call x a **backward-extension endpoint** and α' a **backward-extension sequence** w.r.t. α.

With respect to an endpoint sequence α, if there exists no forward-extension endpoint nor backward-extension , α must be a closed endpoint sequence. For example, as the database in Fig. 3.2, endpoint E^+ is a forward-extension endpoint of sequence $\langle A^+B^+A^-B^- \rangle$: 2, since the support of $\langle A^+B^+A^-B^-E^+ \rangle$ is also 2. Hence, $\langle A^+B^+A^-B^- \rangle$ is not closed. The CEMiner checks closure in two directions as follows,

- **Forward directional checking** is used to grow the temporal patterns and also check the forward-extension endpoint and closure of patterns.
- **Backward directional checking** is used to check the backward-extension endpoint and closure of a temporal pattern and prune the search space.

CEMiner partitions database into smaller projected databases and appends locally frequent endpoints to grow patterns recursively and also verify whether they are closed or not.

For a temporal pattern $\alpha = \langle a_1, a_2, \ldots, a_n\rangle$ and a locally frequent endpoint y, a pattern $\alpha' = \langle a_1, a_2, \ldots, a_n, y\rangle$ or $\langle a_1, a_2, \ldots, a_n \cup y\rangle$ is not closed, if there is a forward-extension endpoint x_j in each sequence where α' appears (forward directional checking). And if there is a backward-extension endpoint x_i in each sequence where α' appears, α' is also not closed (backward directional checking). Otherwise, α' is closed.

Definition 3.6 (The *i*-th last-in-first appearance)

For an endpoint sequence α containing an endpoint sequence $\langle a_{1,} a_{2, \ldots,} a_n \rangle$, the i-th last-in-first appearance w.r.t. $\langle a_{1,} a_{2, \ldots,} a_i \rangle$ in α is denoted as LF_i and defined recursively as: 1) if $i = n$, it is the last appearance of a_i in the first instance of $\langle a_{1,} a_{2, \ldots,} a_i \rangle$ in α ; 2) if $1 \leq i < n$, it is the last appearance of a_i in the first instance of $\langle a_{1,} a_{2, \ldots,} a_i \rangle$ in α and LF_i must appear before LF_{i+1}. For example, given the endpoint sequence $\alpha = \langle A_1^+ B_1^+ A_1^- B_1^- (A_2^+ B_2^+) (A_2^- B_2^-) D^+ D^- E^+ E^- \rangle$ and $p = \langle B^+ B^- D^+ D^- \rangle$ as prefix, the 2nd and the 4th last-in-first appearance w.r.t. prefix p in α are B_2^- and D^- respectively. Since the first instance of p in α is $\langle A_1^+ B_1^+ A_1^- B_1^- (A_2^+ B_2^+) (A_2^- B_2^-) D^+ D^- \rangle$ and the second endpoint in p is B^-, the 2nd last-in-first appearance w.r.t. prefix p in α is the last appearance of B^-, i.e., B_2^- in α. Likewise, the 4th last-in-first appearance w.r.t. prefix p in α is D^-.

Definition 3.7 (The *i*-th semi-maximum period)

For a sequence α containing an endpoint sequence $\langle a_{1,} a_{2, \ldots,} a_n \rangle$, we can define the i-th semi-maximum period of $\langle a_{1,} a_{2, \ldots,} a_i \rangle$ in α as: 1) if $1 < i \leq n$, it is the piece of sequence between the end of the first instance of $\langle a_{1,} a_{2, \ldots,} a_{i-1} \rangle$ in α and the i-th last-in-first appearance w.r.t. $\langle a_{1,} a_{2, \ldots,} a_i \rangle$; 2) if i = 1, it is the piece of sequence in α located before the 1st last-in-first appearance w.r.t. $\langle a_{1,} a_{2, \ldots,} a_i \rangle$. For example, given an endpoint sequence $\alpha = \langle A_1^+ B_1^+ A_1^- B_1^- (A_2^+ B_2^+) (A_2^- B_2^-) D^+ D^- E^+ E^- \rangle$ and $p = \langle B^+ B^- D^+ D^- \rangle$ as prefix, the 1st semi-maximum period of prefix p in α is $\langle A_1^+ B_1^+ A_1^- B_1^- A_2^+ \rangle$. Since the first instance of p in α is $\langle A_1^+ B_1^+ A_1^- B_1^- (A_2^+ B_2^+) (A_2^- B_2^-) D^+ D^- \rangle$ and the first endpoint in p is B^+, the 1st last-in-first appearance w.r.t. prefix p in α is B_2^+, the sequence before B_2^+ in α is $\langle A_1^+ B_1^+ A_1^- B_1^- A_2^+ \rangle$. Likewise, the 2nd semi-maximum period of prefix p in α is the piece of sequence between B_1^+ and B_2^-, i.e., $\langle A_1^- B_1^- (A_2^+ B_2^+) A_2^- \rangle$.

Definition 3.8 (EBackScan search space pruning)

Let an endpoint sequence $\alpha = \langle a_1, a_2, \ldots, a_n \rangle$, if $\exists i$, $1 \leq i \leq n$ and there exists an endpoint x which appears in each of the i-th semi-maximum periods of the prefix α in database DB, we can safely stop growing α. Since we can derive a new endpoint sequence $\alpha' = \langle x, a_1, a_2, \ldots, a_n \rangle$ ($i = 1$) or $\alpha' = \langle a_1, a_2, \ldots, a_{i-1} \cup x, a_i, \ldots, a_n \rangle$ ($1 < i \le n$) or $\alpha' = \langle a_1, a_2, \ldots, a_{i-1}, x, a_i, \ldots, a_n \rangle$ ($1 < i \le n$) and all ($\alpha \sqsubset \alpha'$) and (*support* (α) = *support* (α')) hold. Any locally frequent endpoint w.r.t. α is also a locally frequent w.r.t. α'. Hence we can stop growing the endpoint sequence α, since there is no hope to discover closed temporal patterns from α.

3.4.2 Proposed Algorithm

Fig. 3.4 illustrates the main framework of CEMiner. It first transforms the temporal database to endpoint representation and counts the support of each endpoint concurrently. It also removes infrequent endpoints under given minimum support, *min_sup* (Lines 2-3, algorithm 3.1). For each frequent starting endpoint x, we build projected database $DB_{|x}$ and use *EBackScan* to check whether x can be pruned or not (Lines 5-7, algorithm 3.1). If not, we compute the number of backward-extension endpoints and call EBIDE recursively (Line 9, algorithm 3.1). Finally, we output all closed temporal pattern (Line 10, algorithm 3.1).

Algorithm 3.1: CEMiner (*DB*, *min_sup*)

Input: a temporal database *DB*, and the minimum support *min_sup*
Output: all closed temporal patterns *CTP*

1: $CTP \leftarrow \varnothing$;
2: transform *DB* into endpoint presentation;
3: find all frequent endpoints and remove infrequent endpoints;
4: $FSE \leftarrow$ all frequent **starting** endpoint;
5: **for** each interval $x \in FSE$ **do**
6:     construct projected database $DB_{|x}$ with regard to x;
7:     **if** ***EBackScan***(x, $DB_{|x}$) = "false" **then**
8:         BE = backward extension check (x, $DB_{|x}$);
9:         ***EBIDE*** ($DB_{|x}$, x, *min_sup*, *BE*, *CTP*);
10: output all closed temporal patterns *CTP*;

Fig. 3.4: CEMiner algorithm

The pseudo code of EBIDE is shown in Fig. 3.5. For a prefix α, EBIDE scans its projected database $DB_{|\alpha}$ once to discover all local frequent endpoints (Line 1, algorithm 3.2) and computes the number of forward-extension endpoints (Lines 2-3, algorithm 3.2). If α is a temporal pattern and has neither backward-extension endpoint nor forward-extension endpoint, then α is a closed temporal pattern (Lines 4-5, algorithm 3.2). For each frequent endpoint, we can append it to original prefix to generate new sequence α' with the length increased by 1 (Lines 6-11, algorithm 3.2). In this way, the prefixes are forward-extended.

Algorithm 3.2: EBIDE ($DB_{|\alpha}$, α, *min_sup*, *BE*, *CTP*)

Input: a projected database $DB_{|\alpha}$, an endpoint sequence α, the minimum support *min_sup*, and a set of closed temporal patterns *CTP*
Output: a set of closed temporal patterns *CTP*

01: scan $DB_{|\alpha}$ once, remove infrequent endpoints and find every frequent endpoint y such that:
(i) y can be assembled to the last endpoint of α to form a temporal pattern; or
(ii) $\langle y \rangle$ can be appended to α to form a temporal pattern;
02: $LFE \leftarrow$ all local frequent endpoint;
03: $FE = |\{ z \mid (z \in LFE) \wedge (support(z) = support(\alpha)\}|$;
04: **if** ($BE + FE == 0$) and (α is a temporal pattern) **then** // no backward and forward extension
05: $CTP \leftarrow CTP \cup \{\alpha\}$; // α is a closed temporal pattern
06: **for each** $y \in LFE$ **do**
07: **if** y is a "finishing endpoint" **then**
08: **if** exist corresponding starting endpoint in α **then**
09: append b to α to form α'; // pre-pruning strategy
10: **if** y is a "starting endpoint **then**
11: append y to α to form α';
12: construct projected database $DB_{|\alpha'}$ with insignificant postfix elimination; // post-pruning strategy
13: **if** ***EBackScan*** (α', $DB_{|\alpha'}$) = "false" **then**
14: BE = backward extension check (α', $DB_{|\alpha'}$);
15: ***EBIDE*** ($DB_{|\alpha'}$, α', *min_sup*, *BE*, *CTP*);

Fig. 3.5: EBIDE algorithm

With the property of event endpoint, we use three pruning strategies, **pre-pruning**, **post-pruning**, and **pair-pruning** to reduce the searching space efficiently and effectively. First, the starting endpoint and finishing endpoint definitely occur in pairs in an endpoint sequence. We only require projecting the frequent finishing endpoints which have the corresponding start endpoints in their prefixes (Lines 7-9, algorithm 3.2). It is called pre-pruning strategy which can prune off non-qualified patterns before constructing projected database. Second, when we construct a projected database, some endpoints in postfixes need not be considered. With respect to a prefix sequence α, a finishing endpoint in projected postfix is called significant, if it has a corresponding starting endpoint in projected postfix or in α. When constructing the projected database $DB_{|\alpha}$, only the significant endpoints are collected and all insignificant endpoints are eliminated since they can be ignored in the discovery of closed temporal patterns. The second pruning method is called post-pruning strategy which eliminates insignificant endpoints when constructing projected database (Lines 12-13, algorithm 3.2). Finally, if α' is frequent, EBIDE uses ***EBackScan*** to check if α' can be pruned (Line 15, algorithm 3.2). If not, it computes the number of backward-extension endpoints and calls itself recursively (Lines 16-17, algorithm 3.2).

Moreover, we can avoid some unnecessary checking based on the characteristic of endpoint representation. When extending the pattern by a locally frequent endpoint, if the appending endpoint is a finishing endpoint, we require a two-directional closure checking, i.e., backward-extension and forward-extension checking, to verify whether the pattern is closed or not. However, if the appending endpoint is a starting endpoint, we can omit the closure checking. Since the starting endpoint and finishing endpoint always occur in pairs in an endpoint sequence, forward directional checking is unnecessary. Actually, we just require growing the pattern. The last pruning method is called pair-pruning.

We take the database in Fig. 3.2 with *min_sup* = 2 as an example. There are 17 event intervals which can be regarded as 4 event sequences in the database. After transforming database, we can find all frequent endpoints. They are $\langle A^+\rangle$: 3, $\langle A^-\rangle$: 3, $\langle B^+\rangle$: 4, $\langle B^-\rangle$: 4, $\langle D^+\rangle$: 4, $\langle D^-\rangle$: 4, $\langle E^+\rangle$: 4, and $\langle E^-\rangle$: 4, where the notation "$\langle$*pattern*$\rangle$: *count*" represents the sequence and its associated support count. The event sequences with corresponding endpoint representation are shown as in first column in Fig. 3.6. We take the frequent endpoint A^+ and E^+ as examples to further discuss in details.

For an endpoint A^+, the projected database with respect to A^+ has 3 sequences: $\langle B^+A^-B^-D^+E^+E^-D^-\rangle$, $\langle B^+A^-(B^-D^+)E^+E^-D^-\rangle$, and $\langle A^-D^+E^+E^-D^-\rangle$. Since A^+ is a starting endpoint, by pair-pruning, we need not do closure checking. Continuing the recursive process with the $DB_{|A^+}$, we can discover all closed temporal patterns prefixed with A^+. In addition, when projecting frequent endpoint E^+, the endpoint D^- in generated postfix sequences will be eliminated by post-pruning strategy directly since D^- is insignificant. The last column in Fig. 3.6 lists all generated closed temporal patterns. Obviously, the set of closed patterns expresses the same information as the set of temporal patterns, but includes much fewer patterns.

<table>
<tr><th>event sequences with corresponding endpoint representation</th><th>prefix</th><th>projected database
(: insignificant endpoint)</th><th>closed temporal patterns
(: not closed)</th></tr>
<tr><td rowspan="4">S1: $\langle A^+ (B^+C^+) A^- B^- C^- D^+ E^+ E^- D^- \rangle$
S2: $\langle B^+ B^- D^+ (E^+ F^+) (E^- F^-) D^- \rangle$
S3: $\langle A^+ B^+ A^- (B^- D^+) E^+ E^- D^- \rangle$
S4: $\langle B^+ B^- A^+ A^- D^+ E^+ E^- D^- \rangle$

infrequent endpoint elimination
↓

S1: $\langle A^+ B^+ A^- B^- D^+ E^+ E^- D^- \rangle$
S2: $\langle B^+ B^- D^+ E^+ E^- D^- \rangle$
S3: $\langle A^+ B^+ A^- (B^- D^+) E^+ E^- D^- \rangle$
S4: $\langle B^+ B^- A^+ A^- D^+ E^+ E^- D^- \rangle$</td><td>$\langle A^+ \rangle$</td><td>S1: $\langle B^+ A^- B^- D^+ E^+ E^- D^- \rangle$
S3: $\langle B^+ A^- (B^- D^+) E^+ E^- D^- \rangle$
S4: $\langle A^- D^+ E^+ E^- D^- \rangle$</td><td>$\langle A^+A^- \rangle$: 3 (not closed)
$\langle A^+B^+A^-B^- \rangle$: 2 (not closed)
$\langle A^+A^-D^+D^- \rangle$: 3 (not closed)
$\langle A^+A^-E^+E^- \rangle$: 3 (not closed)
$\langle A^+B^+A^-B^-E^+E^- \rangle$: 2
$\langle A^+A^-D^+E^+E^-D^- \rangle$: 3</td></tr>
<tr><td>$\langle B^+ \rangle$</td><td>S1: $\langle A^- B^- D^+ E^+ E^- D^- \rangle$
S2: $\langle B^- D^+ E^+ E^- D^- \rangle$
S3: $\langle A^- (B^- D^+) E^+ E^- D^- \rangle$
S4: $\langle B^- A^+ A^- D^+ E^+ E^- D^- \rangle$</td><td>$\langle B^+B^- \rangle$: 4 (not closed)
$\langle B^+B^-D^+D^- \rangle$: 3 (not closed)
$\langle B^+B^-E^+E^- \rangle$: 4
$\langle B^+B^-D^+E^+E^-D^- \rangle$: 3</td></tr>
<tr><td>$\langle D^+ \rangle$</td><td>S1: $\langle E^+ E^- D^- \rangle$
S2: $\langle E^+ E^- D^- \rangle$
S3: $\langle E^+ E^- D^- \rangle$
S4: $\langle E^+ E^- D^- \rangle$</td><td>$\langle D^+D^- \rangle$: 4 (not closed)
$\langle D^+E^+E^-D^- \rangle$: 4</td></tr>
<tr><td>$\langle E^+ \rangle$</td><td>S1: $\langle E^- D^- \rangle$
S2: $\langle E^- D^- \rangle$
S3: $\langle E^- D^- \rangle$
S4: $\langle E^- D^- \rangle$</td><td>$\langle E^+E^- \rangle$: 4 (not closed)</td></tr>
</table>

Fig. 3.6: An example of projected databases and closed temporal patterns

3.5 Experimental Results

To best of our knowledge, there have been no efficient methods developed for mining closed temporal patterns. Hence, to evaluate the performance of CEMiner, four temporal pattern mining algorithms, CTMiner [8], H-DFS [27], IEMiner [29] and TPrefixSpan [36] are compared with CEMiner. All algorithms were implemented in C++ language and tested on a computer with Pentium D 3.0 GHz with 2 GB of main memory. The performance study has been conducted on both synthetic and real world datasets. First, we compare the execution time using synthetic datasets at different minimum support. Second, we conduct an experiment to observe the memory usage and the scalability on execution time of CEMiner. Finally, CEMiner is applied in real-world dataset, library lending data, to show the performance and the practicability of mining closed temporal patterns.

The synthetic data sets in the experiments are generated using synthetic generation program modified from [1]. Since the original data generation program was designed to generate time point-based data, the generator for closed temporal pattern mining algorithm requires modifications on interval events accordingly. The parameter setting of temporal data generator is shown in Fig. 3.7.

Parameters	Description
$\lvert \boldsymbol{D} \rvert$	Number of event sequences
$\lvert \boldsymbol{C} \rvert$	Average size of event sequences
$\lvert \boldsymbol{S} \rvert$	Average size of potentially frequent sequences
$\boldsymbol{N_S}$	Number of potentially frequent sequences
$\boldsymbol{N}$	Number of event symbols

Fig. 3.7: Parameters of synthetic data generator

We create a set of potentially frequent sequences used in the generation of event sequences. The number of potentially frequent sequences is N_S. A potentially frequent sequence is generated by first picking the size of sequence from a Poisson distribution with mean equal to | S |. Then, the event intervals in potentially frequent sequence are chosen from N event symbols randomly. All the duration times of event intervals are classified into three categories: long, medium and short, which are normally distributed with an average length of 12, 8 and 4, respectively. For each event interval, we first randomly decide its category and then determine its length by drawing a value. The temporal relations between consecutive intervals are selected randomly to form a potentially frequent sequence. Since we adopt normalized temporal patterns [13], the temporal relationships can be chosen from the set {*before*, *meets*, *overlaps*, *is-finished-by*, *contains*, *starts*, *equal*}. After all potentially frequent sequences are determined, we generate | D | event sequences. Each event sequence is generated by first deciding the size of sequence, which was picked from a Poisson distribution with mean equal to | C |. Then, each event sequence is generated by assigning a series of potentially frequent sequences.

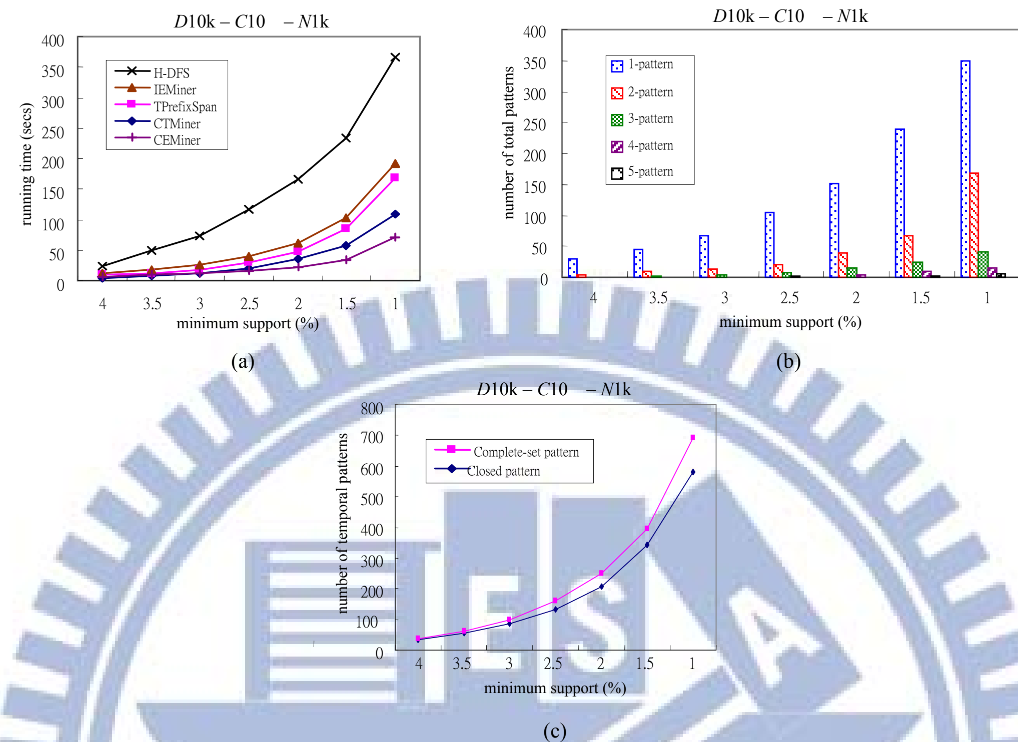


Fig. 3.8: The performance and mining result on data set $D10k – C10 – N1k$

3.5.1 Performance on Synthetic Datasets

In all the following experiments, two parameters are fixed, i.e., $|S| = 4$ and $N_S = 5{,}000$. The other parameters are configured for comparision. The first experiment of the five algorithms is on the data set D10k–C20–N1k. Fig. 3.8(a) shows the running time of the five algorithms with minimum supports varied from 1 % to 4 %. Obviously, when the minimum support value decreases, the processing time required for all algorithms increases. We can see that when the support is greater than 3.5%, CTMiner outperforms CEMiner. However, when we continue to the lower threshold, the runtime for IEMiner, H-DFS and TPrefixSpan increase drastically compared to CEMiner. This is partly because of the generation of an explosive number of frequent patterns for the complete-set mining algorithm. When minimum support is 1 %, CEMiner is about 1.5 times faster than CTMiner, more than 2 times faster than TPrefixSpan, about 3 times faster than IEMiner and more than 5 times faster than H-DFS. Fig. 3.8(b) shows the number of generated closed and complete-set patterns at different support thresholds. Fig. 3.8(c) shows the distribution of closed patterns, from which one can see that when minimal support is no less than 3%, the length of closed patterns is short (only 2-3), and the maximum number of closed patterns in total is 580.

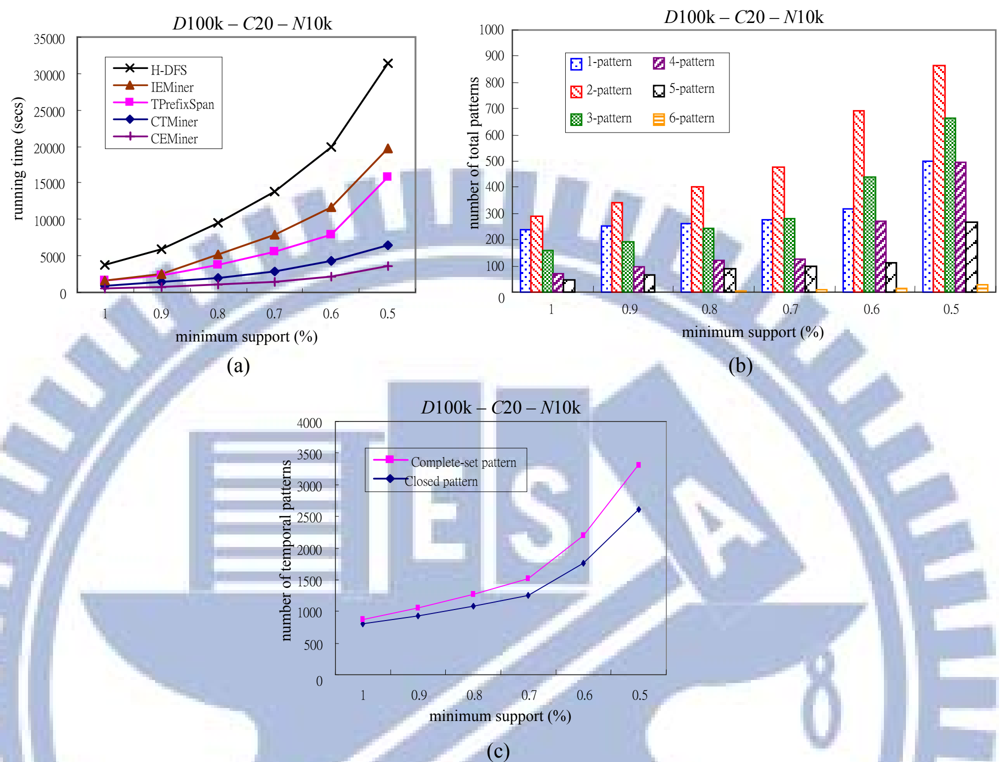


Fig. 3.9: The performance and mining result on data set *D*100k – *C*20 – *N*1k

The second experiment is performed on data set *D*100k–*C*20–*N*10k, which is much larger since it contains 100,000 event sequences and 10,000 event intervals. Figure 9 shows the performance and mining result. Fig. 3.9(a) and 3.9(b) illustrates the running time of the five algorithms and the number of generated closed and complete-set patterns at different support thresholds respectively. However, we vary the minimum support thresholds from 0.5 percent to 1 percent to generate larger number of closed patterns from large data set. The data set contains a large number of closed temporal patterns when minimum support is reduced to 0.5 %. CEMiner is about 2 times faster than CTMiner, more than 4 times faster than TPrefixSpan, more than 5 times faster than IEMiner and about 9 times faster than H-DFS. The distribution of closed patterns is shown in Fig. 3.9(c), and the maximum number of closed patterns in total is 2,616.

3.5.2 Scalability and Memory Usage Studies

In this section, we study the scalability and memory usage of the CEMiner algorithm. Here, we use the data set C = 20, N = 10k with varying different database size. Fig. 3.10 shows the results of scalability tests of the CEMiner algorithm, with the database size growing from 100K to 500K sequences, and with different minimum support threshold varying from 3 % to 1 %. As the size of database increases and minimum support decreases, the processing time of CEMiner increases, since the number of frequent patterns also increases. As can be seen, CEMiner is linearly scalable with different minimum support threshold. When the number of generated closed patterns is large, the runtime of CEMiner still increases linearly with different database size.

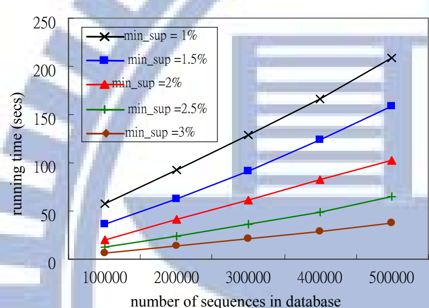


Fig. 3.10: The performance with different database size

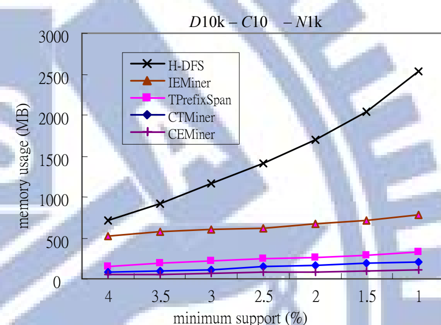


Fig. 3.11: The memory usage of five algorithms

Then, we compare the memory usage among the five algorithms, CEMiner, CTMiner, TPrefixSpan, IEMiner and H-DFS using synthetic data set D10k–C10–N1k. Fig. 3.11 shows the results, from which we can observe that CEMiner is not only more efficient, but also more stable in memory usage than the other four algorithms. For example, when minimum support threshold is reduced to 1%, CEMiner is about 2 times smaller than CTMiner, more than 3 times smaller than TPrefixSpan, almost 7 times smaller than IEMiner and more than 25 times smaller than H-DFS. This also explains why in our previous performance tests when the support threshold becomes extremely low, why CEMiner is still efficient and outperforms state-of-the-art algorithms. Based on our analysis, CEMiner only requires memory space to hold the closed sequence data which is much less than frequent complete-set sequence data. CTMiner and TPrefixSpan still consume memory space to hold the generation of an explosive number of frequent patterns for the complete-set mining. Same as IEMiner and H-DFS, both of them need memory space to hold candidate sequences in each level. When the minimal support threshold drops, the set of candidate sequences grows up quickly, which results in memory consumption upsurging.

In summary, performance study shows that CEMiner has the best overall performance among the algorithms tested. The scalability study also shows that CEMiner scales well even with large databases and low thresholds. The memory usage analysis shows the efficient memory consumption of CEMiner and part of the reason why other algorithms become slow since the candidate sequences may consume a huge amount of memory.

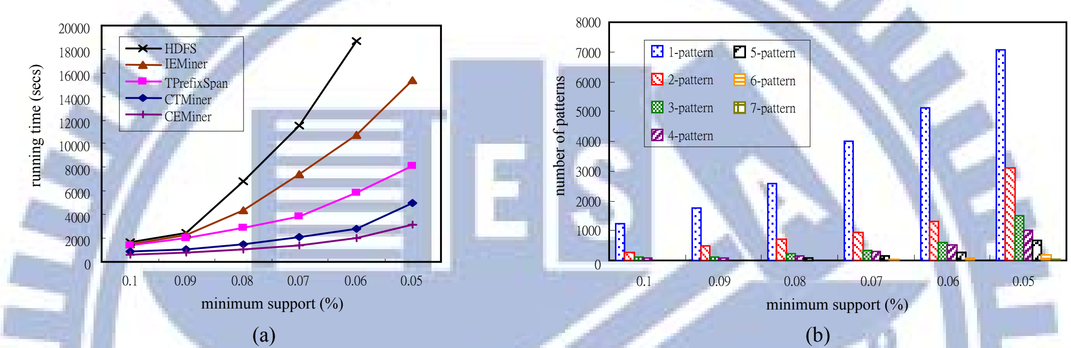


Fig. 3.12: The performance and mining result on library data set from NCTU

3.5.3 Real-World Dataset Analysis

In addition to using synthetic data sets, we have also performed an experiment on real world dataset to compare the performance and indicate the applicability of closed temporal pattern mining. The database used in the experiment consists a collection of 1,098,142 library records (lending and returning) for three years from the National Chiao Tung University Library. The experimented database includes 206,844 books and 28,339 readers. An event interval is constructed by a book ID and corresponding lending and returning time. The size of database is the number of sequences in database (same as the number of readers, 28,339). The maximal and the average length of sequences are 262 and 38 respectively.

Figure 3.12 shows the performance and mining result. Fig. 3.12(a) indicates the running time of five mining algorithms with varying minimum support thresholds from 0.1 % to 0.05 % and the number of generated patterns under different thresholds is shown in Fig. 3.12(b). As the minimum support drops down to 0.05 %, there are 13,550 closed patterns and the running time of CEMiner is about 1.5 times faster than CTMiner, more than 2 times faster than TPrefixSpan, about 5 times faster than IEMiner and H-DFS has never terminated.

3.6 Summary

Previous studies of mining closed sequential pattern mainly are focused on time point-based data. Little attention has been paid to the mining of closed temporal patterns from time interval-based data. Since the processing for complex relations among intervals may require generating and examining large amount of intermediate subsequences, mining closed temporal patterns from time interval-based data is an arduous problem. In this chapter, we develop an efficient algorithm, **CEMiner**, to discover closed temporal patterns without candidate generation, based on proposed endpoint representation. The algorithm further employs three pruning methods to reduce the search space effectively. The experimental studies indicate that CEMiner is efficient and scalable. Both running time and memory usage of CEMiner outperform state-of-the-art algorithms. Furthermore, we also apply CEMiner on real world dataset to show the efficiency and the practicability of mining time interval-based closed pattern.

Chapter 4

Incremental Mining Temporal Patterns from Interval-based Database

4.1 Introduction

Sequential pattern mining is an essential data mining technique with broad applications, such as market and customer analysis, network intrusion detection, analysis of Web access, and finding of tandem repeats in DNA sequences, to name a few. Several efficient algorithms exhibit excellent performance in discovering sequential patterns from a static database, i.e., mine the entire database and acquire the results in a one-stop solution. Nevertheless, the assumption of having a static database may not hold in a number of applications. The database usually grows incrementally over time, i.e., some new data may be added. The algorithms based on static database do not consider the evolution of database and the maintenance of discovered sequential patterns. The result mined from the original database may no longer be valid since existing sequential patterns will be invalid, and new sequential patterns may be introduced with the evolution of databases. Obviously, re-mining the updated databases from scratch each time is inefficient because it wastes computational resources and neglects the previous mining result.

Previous research of the incremental mining algorithm [4, 5, 7, 9, 12, 14, 19, 23, 26, 28, 42] mainly focused on sequential patterns discovered from time point-based data. Prior works have claimed that in reality, mining time interval-based patterns is more practical [8]. Interval-based sequential patterns, also referred to as **temporal patterns**, occasionally can reveal more precise information. In many real-world applications, some events, which intrinsically persist for periods of time instead of instantaneous occurrences, cannot be treated as "time points." In such cases, the data is usually a sequence of interval events with both start and finish times. Examples include library lending, stock fluctuation, patient diseases, and meteorology data, to name a few.

Table 4.1: Part of temporal patterns discovered from of NCTU library

PID	temporal patterns	support
1	"The Know-It-All" "The Curious Incident of the Dog in the Night-time" "The Hitchhiker's guide to the galaxy" "The Restaurant at the End of the Universe"	163 (0.57%)
2	"The Know-It-All" "The Curious Incident of the Dog in the Night-time" "Le Cosmicomiche" "The One Hundred Years of Solitude"	43 (0.15%)
3	"The End of the Affair" "I Served the King of England" "The Pearl in the Deep"	92 (0.32%)
4	"The End of the Affair" "I Served the King of England" "The Inheritance of Loss"	35 (0.12%)

Consider an example of mining temporal patterns from the NCTU library lending datasets. Usually, there is duration between the time of a reader borrowing a book and the time he/she returning the book. Thus, the lending dataset, in general, is time interval-based. By extracting some users' lending patterns, we could develop a recommendation system for library. This information would be more helpful than conventional sequential time point-based pattern. Table 4.1 illustrates some temporal patterns (part of mining results) discovered from the NCTU library. We used pattern 1 and 2 for discussion. Suppose that two readers, Mary and Sue, both check out the books "The Know-It-All" and "The Curious Incident of the Dog in the Night-time." If Mary checks out two books simultaneously, the library can send her an e-mail to notify her that the book "The Hitchhiker's Guide to the Galaxy" is still on the shelf, or that the book "The Restaurant at the End of the Universe" will be returned by June 23, 2011. However, if Sue checks out two books at different times, the library may send her an e-mail to notify her about the availability of books "Le Cosmicomiche" or "The One Hundred Years of Solitude." The temporal patterns offer a more expressive result to present correlations among data than conventional sequential patterns.

Allen’s 13 temporal logics [2] are usually adopted to describe the complex relations among intervals, as follows: “before,” “after,” “overlap,” “overlapped by,” “contain,” “during,” “start,” “started by,” “finish,” “finished by,” “meet,” “met by,” and “equal.” However, Allen’s temporal logics are binary relations and may experience several problems when describing relationships among more than three event intervals. An appropriate representation is crucial for this circumstance. Various representations [8, 13, 16, 24, 25, 29, 36] have been proposed; however, most of them have a restriction on either ambiguity or scalability. In this chapter, we utilize the endpoint arrangements to effectively simplify the processing of complex relations, which is the major bottleneck of incremental mining of temporal patterns. Since the endpoints are non-overlapped, Allen’s 13 temporal logics can be reduced to 3 relations, i.e. “*before*,” “*equal*” and “*after*.”

As mentioned early, new time interval-based data is generated. To truly capture temporal patterns, one should re-execute existing algorithms of mining temporal patterns from the updated database, where the new data is appended or the new record is inserted. In this chapter, we target at designing algorithms to incrementally mine temporal patterns. To the best of our knowledge, no methods have been discussed on how to discover frequent sequential patterns from time interval-based data in an incremental environment. Since the feature of time intervals differs considerably from that of time points, the pairwise relationships between any two interval events are intrinsically complex. This complex relation is a crucial problem in the design of an efficient and effective algorithm for maintaining temporal patterns. When appending an interval to an event sequence, the complex relations may lead to the generation of a larger number of possible candidates and consume more memory space.

Two types of incremental updates for interval sequence database are used, 1) inserting new sequences into database, denoted as INSERT; 2) appending new intervals to existing sequences, denoted as APPEND. A real world application may include all types of updates. When the database is updated with a combination of INSERT and APPEND, we can regard the INSERT as a special case of APPEND, for inserting a new sequence is equivalent to appending a new sequence to an empty sequence, as shown in Fig. 4.1. This chapter proposes an efficient algorithm, ***Inc_CTMiner*** which stands for *I*ncremental *T*emporal *Miner*, to address the crucial problem and incrementally discover temporal patterns based on the coincidence representation. Furthermore, Inc_CTMiner employs some pruning strategies to reduce the search space and avoids non-promising database projection. Experimental studies on both synthetic and real datasets indicated that, in the incremental environment, Inc_CTMiner is efficient and outperforms the state-of-the-art algorithms, which are based on static database. Our experiments also revealed that the proposed approach is scalable and consumes a smaller memory space. We also applied Inc_CTMiner on real world datasets to demonstrate the practicability of maintaining the temporal patterns.

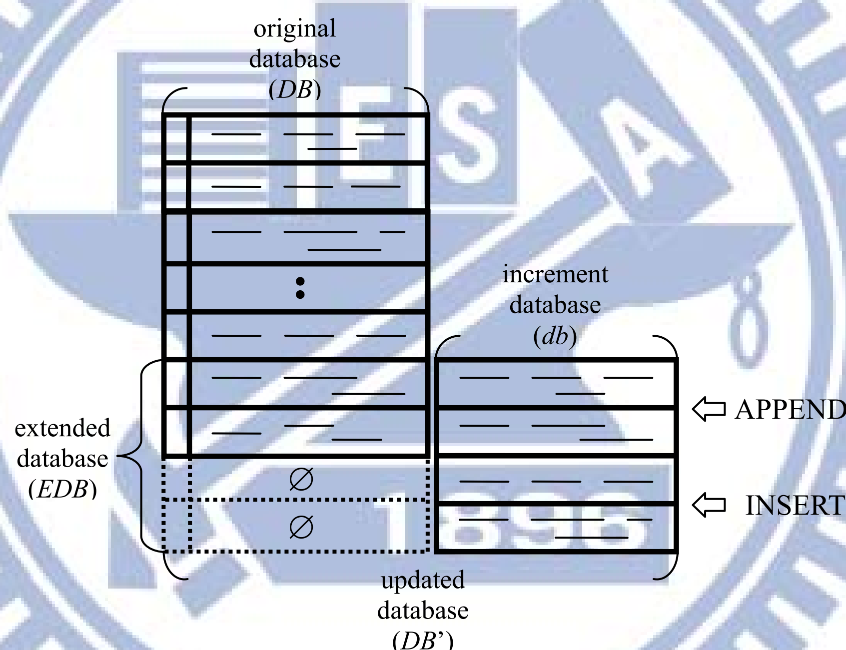


Fig. 4.1: Concept of INSERT and APPEND updates interval sequence

The remainder of this chapter is organized as follows: Section 4.2 presents the related work; Section 4.3 introduces the preliminaries; Section 4.4 provides incremental mining algorithms; Section 4.5 presents the experimental results and performance study; and finally, Section 4.6 summerizes this chapter.

4.2 Related Work

A number of studies have investigated the mining of temporal patterns [2, 8, 13, 17, 24, 25, 27, 29, 31, 33, 35, 36, 37, 41] in a static environment. Kam et al. [16] proposed a hierarchical representation and designed an algorithm to discover temporal patterns. Although hierarchical representation is a compact encoding method, it may suffer from two ambiguous problems, as follows: 1) the same relationships among event intervals can be mapped to different temporal patterns; and 2) the same temporal pattern can represent different relationships among event intervals. Hoppner [13] proposed a nonambiguous representation, relation matrix, which exhaustively lists all binary relationships between event intervals in a pattern. The mining algorithm needs to scan the database repeatedly, which considerably lowers its efficiency, and the relation matrix does not scale effectively if numerous intervals appear in a pattern.

H-DFS [27] was proposed to discover frequent arrangements of temporal intervals. This approach transforms an event sequence into a vertical representation using id-lists. However, H-DFS does not scale effectively when the temporal pattern length increases. TSKR [24] expressed the temporal concepts of coincidence and partial order for interval patterns. The pattern represented in TSKR format is easily understandable and robust; however, it may reveal the relationship between pairwise event intervals ambiguously. Based on MEMISP [20], ARMADA [35] was proposed to find temporal patterns from large databases. Since it is based on relation matrix representation, memory usage is a substantial bottleneck when the database is very large. TPrefixSpan [36] uses temporal representation to discover temporal patterns nonambiguously, but it does not use any pruning strategy to reduce the search space. Augmented hierarchical representation [29] uses additional counting information to achieve a lossless expression. Every Allen describer must take space to store five counters. Based on this representation, IEMiner [29] was proposed by using optimization strategies and removing non-promising candidate sequences, but it must scan the database multiple times.

A robust representation, SIPO [25], used the partial order of intervals and considers the noise tolerance to express relationships among intervals. Nevertheless, the proposed algorithm requires discovering both closed sequential pattern and closed itemset, and therefore, is time consuming. CTMiner [8] is an efficient algorithm for mining temporal patterns. It utilizes a non-ambiguous and compact representation, coincidence representation [8] to facilitate the mining process. It first segments all intervals to disjoint slices based on the global information in a pattern, and subsequently groups all event slices occurring simultaneously to form a coincidence to represent a sequence.

A few prior works [4, 5, 7, 9, 12, 14, 19, 23, 26, 28, 42] have focused on incremental mining sequential patterns from time point-based data. ISM [28] uses a sequence lattice of original database for incrementally mining of sequential patterns. The sequence lattice includes all of the frequent sequences and all of the sequences in the negative border. Two problems occur when using negative border. First, the combined number of sequences in the frequent set and the negative border is large. Second, the sequences in negative border are generated based on the structural relation between sequences. However, these sequences do not necessarily have high support. Therefore, using negative border is very time and memory consuming. Zhang et al. [42] developed two candidate generate-and-test algorithms, GSP+ and MFS+, for incremental mining of sequential patterns when sequences are inserted into or deleted from the original database. ISE [23] is another incremental mining algorithm based on candidate generate-and-test approach. The weakness of these three algorithms is that the candidate set may be very large and the level-wise working manner requires multiple database scans. When the frequent sequences are long, the testing phase is usually slow and costly.

The IncSpan [9] buffers a set of semi-frequent sequences as the candidates in the updated database which can accelerate the maintaining process efficiently. Two optimization techniques, reverse pattern matching and shared projection, were proposed to improve the performance. However, IncSpan fails to find the complete set of sequential patterns from an updated database because several properties are incorrect. Nguyen et al. [26] proved the incompleteness of IncSpan and proposed an algorithm, IncSpan+, to correct the weaknesses of IncSpan. IncSP [12] solved the maintenance problem through effective implicit merging and efficient separate counting over appended sequences. The proposed early candidate pruning technique, further speeds up the discovery of new patterns. PBIncSpan [7] uses a prefix tree to record all frequent sequences and corresponding projected databases to maintain the discovered sequential patterns; however such a method requires extremely huge storage space when the database is large. The proposed pruning strategy is based on the Apriori property and is inefficient when the prefix tree has numerous nodes.

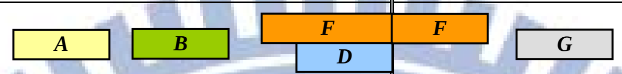
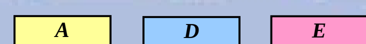
All previous studies for incremental mining are mainly focused on time point-based data which has no concept of duration of time. Limited attention has been paid to updating temporal patterns from interval-based database. In this chapter, we design a new algorithm, Inc_CTMiner, which can incrementally discover temporal patterns effectively and efficiently.

4.3 Preliminary

Let $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$ be the set of event symbols. Without loss of generality, we define a set of uniformly spaced time points based on the natural number N. We say the triplet $(e_i, s_i, f_i) \in \mathcal{E} \times N \times N$ is an event interval, where $e_i \in \mathcal{E}$, $s_i, f_i \in N$ and $s_i < f_i$. The two time points s_i, f_i are called event times, where s_i is the starting time and f_i is the finishing time. The set of all event intervals over $\mathcal{E}$ is denoted by $\mathcal{I}$. An event sequence is a series of event interval triplets $\langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_n, s_n, f_n)\rangle$, where $s_i \leq s_{i+1}$, and $s_i < f_i$. A temporal database is a set of tuple $\langle SID, Q\rangle$ where *SID* is a sequence-id and *Q* is an event sequence. For example, in Table 4.2, the temporal database *ĐB* has 3 event sequences. Given two event sequences *Q* and *Q'*, $Q'' = Q \diamond Q'$ means *Q''* is the concatenation of *Q* and *Q'*. *Q'* is called **appended sequence** of *Q* and *Q''* is called **updated sequence** of *Q* appended with *Q'*.

Definition 4.1 (Increment and updated database)
Given a temporal database *DB* appended with a few event sequences after some time, *DB* is called **original database**. The **increment database** *db* is referred to as the set of newly appended data sequences. The *SID*s of the data sequences in *db* may already exist in *DB*. A database combining all the data sequences from *DB* and *db* is referred to as the **updated database** *DB'*. An **extended database** *EDB* of an updated temporal database *DB*' is a set of event sequences in *DB'* which are the concatenations of sequences in *DB* and *db*. The concept of Definition 4.1 is given as Fig. 4.1.

Table 4.2: An example of temporal database

SID	original database ***DB***	increment database ***db***
	event interval	**event interval**
	pictorial example	**pictorial example**
	coincidence representation	
1	(A, 1, 3), (B, 4, 6), (F, 7, 10), (D, 8, 10)	(F, 10, 13), (G, 14, 18)
		
	(A) (B) (F^+) (F^-D) ◇ **(F) (G)** → **(B) (F^+) (D) (F^-) (G)**	
2	(A, 1, 3), (D, 4, 6), (E, 7, 9)	
		
	(A) (D E) ◇ ∅ → **(A) (D E)**	
3	(A, 1, 3), (D, 4, 6), (E, 7, 10)	
		
	(A) (D E) ◇ ∅ → **(A) (D E)**	
4		(B, 11, 14), (F, 15, 20), (D, 16, 18)
		
	∅ ◇ **(B) (F^+) (D) (F^-)** → **(B) (F^+) (D) (F^-)**	

4.4 Coincidence Representation

The incremental mining of temporal patterns is more difficult than that of conventional sequential patterns. Since the time period of two intervals may overlap, the relation among event intervals is more complex than that of the event points. An appropriate representation is very important for describing relationships among more than three events. Various representations have been proposed but most of them have restriction on either ambiguity or space usage. The existing representations are compared in Table 4.3.

Table 4.3: Comparisons of existing representation

	Hierarchy Representation	Relation Matrix (List)	Temporal Representation	TSKR	Augmented Hierarchy Representation	Coincidence Representation
proposed time	2000 (DaWak)	2002 (IDA)	2007 (TKDE)	2007 (DMKD)	2008 (SIGMOD)	2010 (CIKM)
space usage (for k events)	$k + (k-1) = 2k-1$	$k \times (k-1) = (k^2 - k)$	$2k + (2k-1) = 4k-1$	Best case: k Worst case: k^2	$k + (6 \times (k-1)) = 7k-6$	Best case: k Worst case: $2k$
ambiguous problem	yes	no	no	yes	no	no
relations between events	complex	complex	complex	simple	complex	simple

Given an event sequence $Q = \langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_n, s_n, f_n)\rangle$, the set $T = \{s_1, f_1, s_2, f_2, \ldots, s_i, f_i, \ldots, s_n, f_n\}$ is called a **time set** corresponding to sequence Q where $1 \le i \le n$. If we order all the elements in T and eliminate redundant elements, we can derive a sequence $TS = \langle t_1, t_2, t_3, \ldots, t_k\rangle$ where $t_i \in T$, $t_i < t_{i+1}$. TS_Q is called a **time sequence** corresponding to sequence Q.

Definition 4.2 (Incising Function and Event Slice)

Given an event sequences $Q = \langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_i, s_i, f_i), \ldots, (e_n, s_n, f_n)\rangle$ where $(e_i, s_i, f_i) \in \mathcal{I}$, and $a, b \in TS_Q$,

an incising function $\boldsymbol{\Psi}(\boldsymbol{a}, \boldsymbol{b}, (\boldsymbol{e_i}, \boldsymbol{s_i}, \boldsymbol{f_i}))$ =
$$\begin{cases} e_i & \text{if } (s_i = a) \wedge (f_i = b) \\ e_i^+ & \text{if } (s_i = a) \wedge (f_i > b) \\ e_i^- & \text{if } (s_i < a) \wedge (f_i = b) \\ e_i^* & \text{if } (s_i < a) \wedge (f_i > b) \\ \varnothing & \text{otherwise.} \end{cases}$$

- An event slice $S = \Psi(a, b, (e_i, s_i, f_i))$ is called **starting slice**, if $a = s_i$, $b = \min\{ t \mid t \in TS_Q, s_i < t < f_i \}$, and denoted as e_i^+.
- An event slice $S = \Psi(a, b, (e_i, s_i, f_i))$ is called **finishing slice**, if $a = \max\{ t \mid t \in TS_Q, s_i < t < f_i\}$, $b = f_i$, and denoted as e_i^-.
- An event slice $S = \Psi(a, b, (e_i, s_i, f_i))$ is called **intermediate slice**, if $a \ne s_i$, $b \ne f_i$, $s_i < a < b < f_i$ and $b = \min\{ t \mid t \in TS_Q, a < b < f_i\}$, and denoted as e_i^*.
- An event slice $S = \Psi(a, b, (e_i, s_i, f_i))$ is called **intact slice**, if $a = s_i$ and $b = f_i$ and $\nexists\ t \in TS_Q$ such

that $s_i < t < f_i$, and denoted as e_i.

Let S and S' be two event slices. We say that S is **similar** to S', denoted as $S \approx S'$, if the event symbol of S is identical to the event symbol of S'.

For example, as *db* in Table 4.2, sequence 4 has three event intervals, (B, 11, 14), (F, 15, 20) and (D, 16, 18) and its corresponding time sequence = ⟨11, 14, 15, 16, 18, 20⟩. Event interval F can be incised into three event slices, start slice $F^+ = \Psi(15, 16, (F, 15, 20))$, $F^* = \Psi(16, 18, (F, 15, 20))$ and finish slice $F^- = \Psi(18, 20, (F, 15, 20))$. Event interval B has only one intact slice $B = \Psi(11, 14, (B, 11, 14))$. F^+ and F^- have the same event symbol, F, hence $F^+ \approx F^-$. By Definition 4.2, we know that there are four kinds of event slice. Obviously, an event interval can only have one start slice and one finish slice but can have many intermediate slices.

Definition 4.3 (Grouping Function, Coincidence and Coincidence Sequence)

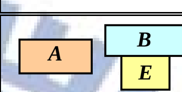
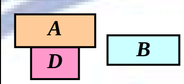
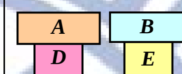
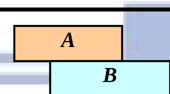
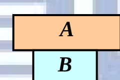
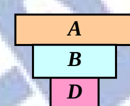
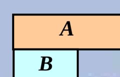
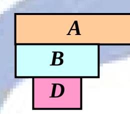
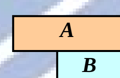
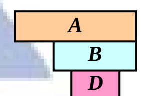
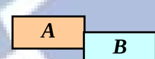
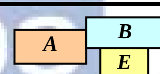
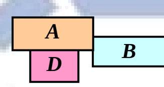
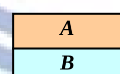
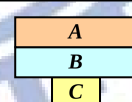
Given an event sequences $Q = \langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_i, s_i, f_i), \ldots, (e_n, s_n, f_n)\rangle$ where $(e_i, s_i, f_i) \in \mathcal{I}$, and $a, b \in TS_Q = \langle t_1, t_2, t_3, \ldots, t_k\rangle$, $1 < k \leq 2n$, a grouping function,

$\boldsymbol{\Phi(a, b, q) = \{\Psi(a, b, (e_1, s_1, f_1)), \Psi(a, b, (e_2, s_2, f_2)), \ldots, \Psi(a, b, (e_n, s_n, f_n))\}}$.

A **coincidence** $C_i = \Phi(t_i, t_{i+1}, Q) = (S_{i1}, S_{i2}, \ldots, S_{ij}, \ldots)$, where t_i and t_{i+1} is two consecutive event times in TS_Q and S_{ij} is an event slice, $1 < i \leq k-1$, $1 \leq j \leq n$. C_i is an ordered set of event slices sorted by **lexicographic order**. A **coincidence sequence** Q_c is denoted by $\langle C_1, C_2, \ldots, C_{k-1}\rangle$ and also called the coincidence representation of Q. To deal with multiple occurrences of events, we attach **occurrence number** to event slices to distinguish multiple occurrences of the same event type in a coincidence sequence. For example, $\langle(A_1^+)(B_1^+)(B_1^- D^+)(D^-)(A_1^- B_2^+)(B_2^-)(EF)(A_2)\rangle$ is a coincidence sequence with occurrence number where both event A and B occur twice.

To facilitate the incremental maintenance of temporal patterns, we also preserve the starting and the finishing time of Q, s^Q and f^Q, respectively. s^Q is the starting time of the first event interval in Q and f^Q is the finishing time of the last event interval in Q, i.e., if $Q = \langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_n, s_n, f_n)\rangle$, $s^Q = s_1$ and $f^Q = f_n$. For a temporal database DB, by Definition 4.2 and 4.3, we can transform it into a set of tuples $\langle SID, Q_c, [s^Q, f^Q]\rangle$ where SID is the sequence-id of each event sequence Q in DB, Q_c is the coincidence representation of Q, and s^Q and f^Q are the starting and finishing time of Q. For example, in Table 2, we can transform three event sequences in *ĐB* into corresponding coincidence sequences. For better readability, later in this chapter, we suppose that the temporal database has been transformed into coincidence representation.

Table 4.4: The coincidence representation of Allen's relations between two intervals

Temporal Relation	Inversed Relation	Pictorial Example	Coincidence representation	Pictorial Example	Coincidence representation
A* before *B	***B* after *A***		$A\ B$		$A\ B^+B^-$
			A^+A^-B		$A^+A^-B^+B^-$
A* overlaps *B	***B* overlapped-by *A***		$A^+(A^-B^+)\ B^-$		
A* contains *B	***B* during *A***		$A^+B\ A^-$		$A^+B^+B^-A^-$
A* starts *B	***B* started-by *A***		$(A^+B)\ A^-$		$(A^+B^+)\ B^-A^-$
A* finished-by *B	***B* finishes *A***		$A^+(A^-B)$		$A^+B^+(A^-B^-)$
A* meets *B	***B* met-by *A***		$A\ @\ B$		$A\ @\ B^+B^-$
			$A^+A^-\ @\ B$		$A^+A^-\ @\ B^+B^-$
A* equal *B	***B* equal *A***		$A\ B$		$(A^+B^+)\ (A^-B^-)$

We adopt coincidence representation [8] to express a temporal pattern since it can accelerate the process of updating temporal patterns when new intervals are appended to the original interval sequences. The coincidence representation has several benefits, and the most important one is that it can simplify the processing of complex pairwise relationships among all intervals effectively. It utilizes the concept of slice-and- coincidence as defined in Definition 4.2 and 4.3, and considers the information of an entire event sequence instead of individual event intervals.

Given two different event intervals A and B, the coincidence representation of Allen's 13 relations between A and B is categorized as in Table 4.4.

4.5. Inc_CTMiner Algorithm

In this section, we develop a new algorithm, named **Inc_CTMiner** (**Incremental Coincidence Temporal Miner**), for incremental mining of temporal patterns, by utilizing the concepts of slice-and-coincidence. Section 4.5.1 gives some basic concepts and a glance of CTMiner algorithm. Section 4.5.2 details the Inc_CTMiner algorithm and also discusses the proposed optimization mechanisms for reducing the search space.

4.5.1 Basic Concepts of Inc_CTMiner

Before introducing the algorithm, we give some definitions first. Let Q_c be a coincidence sequence in a temporal database DB. The Q_c-projected database, denoted as $DB_{|Q_c}$, is the collection of postfixes of coincidence sequences in DB with regards to prefix Q_c. Considering two coincidence sequences $Q_c = \langle C_1, C_2, \ldots, C_n\rangle$ and $Q_c' = \langle C_1', C_2', \ldots, C_m'\rangle$, Q_c is called a subsequence of Q_c', denoted as $Q_c \sqsubseteq Q_c'$, if there exist integers $1 \le i_1 \le i_2 \le \ldots \le i_n \le m$ such that $C_1 \subseteq C_{i1}'$, $C_2 \subseteq C_{i2}'$, …, $C_n \subseteq C_{in}'$. We also call Q_c' a supersequence of Q_c, and Q_c'contains Q_c.

Definition 4.4 (Temporal Pattern)

Given a temporal database DB, a tuple $\langle SID, Q_c, [s^Q, f^Q]\rangle$ is said to contain a coincidence sequence α, if α is a subsequence of Q_c. The support of a coincidence sequence α in DB is the number of tuples containing α, i.e., $\boldsymbol{support\ (\alpha) = |\{\langle SID, Q_c, [s^Q, f^Q]\rangle \mid (\langle SID, Q_c, [s^Q, f^Q]\rangle \in DB) \wedge (\alpha \sqsubseteq Q_c)\}|}$**.** Given a positive integer *min_sup* as the support threshold, the set of temporal patterns includes all coincidence sequences whose supports are no less than *min_sup*.

Let the temporal database DB in Table 4.2 with *min_sup* = 2 be an example. The coincidence sequence $\langle(A)(D)\rangle$ is a temporal pattern since it occurs in sequence 1, 2, and 3, and its *support* = 3 ≥ *min_sup*. A coincidence sequence $\langle(B)(D)\rangle$ is not a temporal pattern since it occurs only in sequence 1, and its *support* = 1 ≤ *min_sup*.

A **frequent pattern tree (*FPT*)** T is a tree that represents the set of temporal patterns in a temporal database. A node d in T stores an event slice and has a tag labeled with "*p*" or "*i*". Label "*p*" means node d corresponding to a temporal pattern that starts from the root node to d. Label "*i*" means node d corresponding to an intermediate sequence of a temporal pattern that starts from the root node to d. Coincidence cutting is captured by using labeled edges. Each tree edge in T has a tag labelled with "solid" or "dash". Solid edge means two connected nodes are in different coincidences; dash edge means two connected nodes are in the same coincidence. Each node also preserves two information, say **support value** and ***sequence_list***. The support value represents the support count of the intermediate sequence or temporal pattern. The *sequence_list* stores a list of sequence-ids, i.e., *SID*s, to represent the sequences containing this intermediate sequence or temporal pattern. The example is as shown in Fig. 4.2(a).

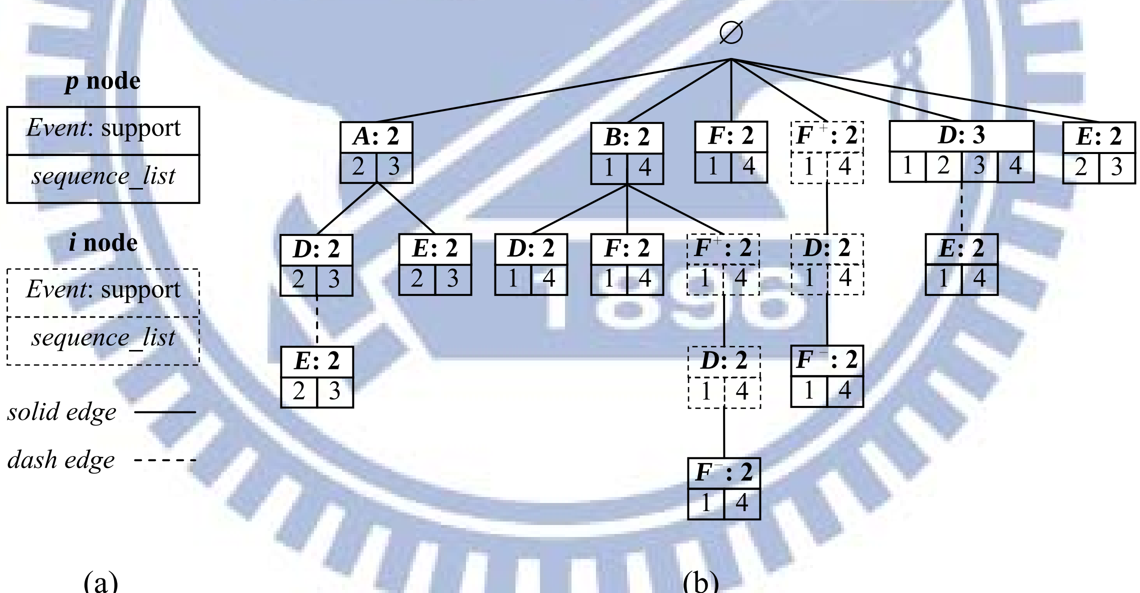


Fig. 4.2: The frequent pattern tree built from updated database *DB*+*db* in Table 4.2

Fig. 4.2(b) shows the frequent pattern tree built from the updated database *DB*+*db* in Table 4.2. The temporal patterns and intermediate sequences are represented by a node with the solid squares and dotted squares, respectively. Coincidences can be captured by using edge label. For instance, $\langle(B)(F^+)(D)(F^-)\rangle$ is a temporal pattern and the solid link illustrates that B, F^+, D and F^-

are all in different coincidence. The formal definition of our problem is given as follows.

Definition 4.5 (Problem Statement)

Given a temporal database *DB*, a minimum threshold *min_sup*, the set of temporal patterns *FPT* in *DB*, and a updated temporal database *DB'* of *DB*, the problem of incremental temporal pattern mining is to mine the set of temporal patterns *FPT'* in *DB'* based on *FPT* instead of re-mining on *DB'* from scratch.

4.5.1.1 Sequence Transformation

The maintenance of time interval-based patterns is much more difficult than conventional time point-based patterns. Since the time period of the two intervals may overlap, the relation among event intervals is more complex than that of the event points. Hence, we use an efficient method, **incision strategy**, to transform the new appending sequences into coincidence representation and accelerate the maintaining process.

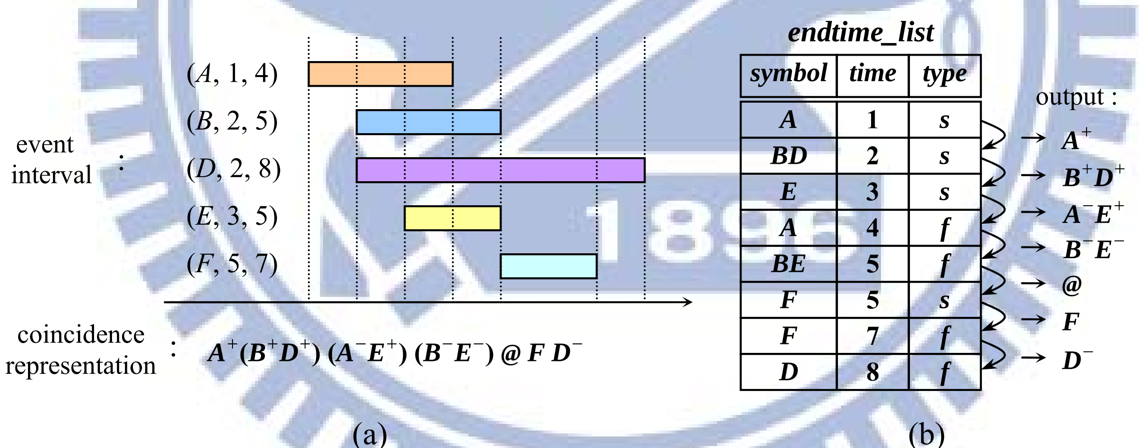


Fig. 4.3: An example of incision strategy

The incision strategy segments all intervals to disjoint slices based on the global information in a sequence. For example, considering an event sequence with five intervals shown in Fig. 4.3(a), we first put all ten end time points into *endtime_list* and sort them in non-decreasing order based on their times and types (start or finish). We merge the event symbol of end time points together if both time and type of end time points are the same. As in Fig. 4.3(b), since the finishing time of interval *B* is identical to the finishing time of interval *E*, we can merge them together. But we can not merge the finish time of interval *E* with the start time of interval *F*, since the type of end time points are not the same. Then we compare each record in *endtime_list* one-by-one to segment event slice. By traversing all the sorted end time points in *endtime_list*, we can generate the event slices effectively.

Algorithm 4.1: *incision_strategy* (Q)

Input: Q: an event sequence
Output: Q_c: a coincidence sequence
Variable: *endtime_list*, *last_endtime*, and *coincidence*

```
endtime_list ← ∅ , last_endtime ← ∅ , coincidence ← ∅ , Q_c ← ∅ ;
add all the end time points of every event interval in Q into endtime_list;
sort every endtime in endtime_list by endtime. time in nondecreasing order;
merge all endtime. symbols together with identical endtime. time and endtime. type;
for each endtime T in endtime_list do
    coincidence ← ∅ ;
    if last_endtime. time = T . time then
        coincidence ← coincidence ∪ "@"; // meet token
    else     // last_endtime. time ≠ T . time
        if last_endtime. type = "s" then
            coincidence ← coincidence ∪ every symbol in last_endtime. symbol add "+";
            // start slice
        if T.type = "f" then
            coincidence ← coincidence ∪ every symbol in T. symbol add "–";   // finish slice
    combine start slice and finish slice with same symbol in coincidence; // intact slice
    Q_c ← Q_c ◇ ⟨ coincidence ⟩ ;
    last_endtime ← T;
output Q_c ;
```

Fig. 4.4: Algorithm of incision strategy

Coincidence representation uses meet token "@" to express the *meet* relation among two adjacent intervals. As the example in Fig. 4.3(a), interval *E* meets interval *F*, hence we add a "@" between two coincidences ($B^- E^-$) and (F). In general, reducing memory usage and saving computation time are two important issues for algorithm design. Since the meet token has been used to distinguish two adjacent intervals, incision strategy can totally avoid the generation of intermediate slices. Given an example as Fig. 4.3(a), the event interval *D* can be segmented into five event slices, one start slice D^+, three intermediate slices D^*, and one finish slice D^-. By trimming the intermediate slices, we can still express the relationship between any two intervals correctly, as shown in Fig. 4.3(a). Utilizing meet token can reduce the memory usage and the computation cost effectively and efficiently, thereby improves the performance of our incision strategy.

The pseudo code of incision strategy is shown as Fig. 4.4. By the merge operation of incision strategy, the event slices occur simultaneously in the same time period can be grouped together to form a coincidence easily. Given an event sequence, we can transform it to an equivalent coincidence sequence by incision strategy. Collecting all coincidence sequences can form a coincidence database which is equivalent to original temporal database.

```
Algorithm 1: CTMiner (DB, min_sup)

Input: DB: a temporal database, min_sup: the minimum support threshold
Output: FPT_DB: frequent pattern tree of a database DB

FPT_DB ← ∅;
use incision_strategy transforming DB into coincidence representation;
call CPrefixSpan (DB , ⟨⟩, min_sup, FPT_DB );
output FPT_DB ;

Procedure CPrefixSpan (DB|α , α , min_sup, FPT_DB )
scan DB|α once, remove infrequent slices and find every frequent slice b such that:
(i)  b can be assembled to the last slice of α or (ii) ⟨b⟩ can be appended to α to form
 a frequent coincidence sequence;    // support(b) ≥ ( min_sup×|DB| )
for each frequent slice b do
    if b is a "finish slice" then
        if exist corresponding start slice in α then    // pre-pruning
            append b to α to form β ;
    if b is a "start slice" or "intact slice" then
        append b to α to form β ;
for each β do
    construct β – projected database DB|β with insignificant postfix elimination;
    // post-pruning
    if |DB|β| ≥ ( min_sup×|DB| ) then
        if β is a temporal pattern then
            insert β into FPT_DB ;
        call CPrefixSpan (DB|β, β, min_sup, FPT_DB );
```

Fig. 4.5: CTMiner algorithm

4.5.1.2 CTMiner Algorithm

CTMiner [8] is an efficient temporal mining algorithm based on static database. It transforms event intervals into non-overlapped event slices and mined all temporal patterns recursively based on the projection technique [30]. Furthermore, CTMiner employs two optimization strategies, pre-pruning and post-pruning, to reduce the search space and avoids non-promising projection. Since the event start slices and finish slices definitely occur in pairs in a sequence, CTMiner only projects the frequent finish slices which have the corresponding start slices in their prefixes. It is called pre-pruning strategy which can prune off non-qualified patterns before constructing projected database. When constructing a projected database, some postfixes need not be considered. With respect to a prefix $\langle p \rangle$, a projected postfix is called significant, if all finish slices in postfix have corresponding start slices in $\langle p \rangle$. CTMiner constructs the projected database $DB_{|\langle p \rangle}$ by collecting significant postfixes only. All insignificant postfixes are eliminated since they can be ignored in the discovery of temporal patterns. This pruning method is called post-pruning strategy which eliminates insignificant sequence when constructing projected database. The pseudo code of CTMiner algorithm is given in Fig. 4.5.

4.5.1.3 Interval Extension

As mentioned above, appending an event sequence is more challenging than conventional sequence. Since an interval has duration, an interval in existing event sequence may merge with an interval in appended event sequence. Given two intervals I_1 and I_2 with the same event symbol and I_1 is in existing event sequence and I_2 is in appended sequence, if the end time of I_1 is the same with the start time of I_2, I_1 and I_2 will merge together. The interval-extension may vary the relation among intervals in the event sequence, hence also modify the coincidence representation of the event sequence. For example, as the event sequence 1 in Table 4.2, the relation between interval F and D is "*finished-by*" in original event sequence, but becomes "*contains*" after concatenation. The coincidence representations of original event sequence and appended sequence are $\langle (A)(B)(F^+)(F^- D) \rangle$ and $\langle (F)(G) \rangle$ respectively. However, the representation of updated sequence is not just the concatenation of two coincidence sequence since the last coincidence of $\langle (A)(B)(F^+)\underline{(F^- D)} \rangle$ will modify the first coincidence of $\langle \underline{(F)}(G) \rangle$, i.e., $\langle (A)(B)(F^+)\underline{(D)(F^-)}(G) \rangle$. Fig. 4.6 indicates all possible variations of Allen relation for concatenating two event sequences.

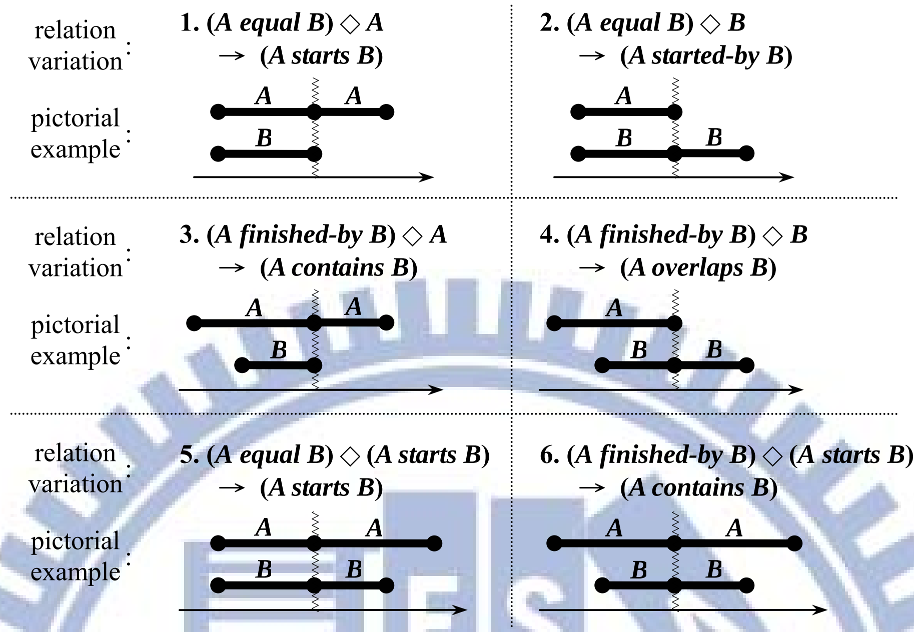


Fig. 4.6: Possible variations of relation and coincidence representation for concatenating two event sequences

Definition 4.6 (Concatenation of coincidence sequence)

Given two coincidence sequences and their corresponding time information, $Q_c = \langle C_1, C_2, \ldots, C_n \rangle$, $[s^Q, f^Q]$ where $C_n = (S_{n1}, \ldots, S_{nx})$ and $Q_c' = \langle C_1', C_2', \ldots, C_m' \rangle$, $[s^{Q'}, f^{Q'}]$ where $C_1' = (S_{11}', \ldots, S_{1y}')$, $Q_c \diamond Q_c'$ means Q_c concatenates with Q_c'. There are three kinds of concatenation for coincidence sequence,

1) **Sequence-extension:** $Q_c \diamond_{seq} Q_c' = \langle C_1, C_2, \ldots, C_n, C_1', C_2', \ldots, C_m' \rangle$, if $f^Q \neq s^{Q'}$;

2) **Entire coincidence-extension:** $Q_c \diamond_{ent} Q_c' = \langle C_1, C_2, \ldots, C_{n\text{-}1}, C_a, C_2', \ldots, C_m' \rangle$, if
   - $f^Q = s^{Q'}$ and $x = y$
   - $\forall S_{ni} \in C_n, S_{1i}' \in C_1', S_{ni} \approx S_{1i}'$ where $1 \leq i \leq x$,

$$C_a = (S_{a1}, \ldots, S_{ai}, \ldots, S_{ax}),\ S_{ai} = \begin{cases} e_{ni} & \text{if } (S_{ni} = e_{ni}) \wedge (S_{1i}' = e_{ni}) \\ e_{ni}^{+} & \text{if } (S_{ni} = e_{ni}) \wedge (S_{1i}' = e_{ni}^{+}) \\ e_{ni}^{-} & \text{if } (S_{ni} = e_{ni}^{-}) \wedge (S_{1i}' = e_{ni}) \\ \varnothing & \text{if } (S_{ni} = e_{ni}^{-}) \wedge (S_{1i}' = e_{ni}^{+}). \end{cases}$$

3) **Partial coincidence-extension:** $Q_c \diamond_{par} Q_c' = \langle C_1, C_2, \ldots, C_{n\text{-}1}, C_a, C_b, C_2', \ldots, C_m' \rangle$, if
   - $f^Q = s^{Q'}$

- $\exists S_{nk} \in C_n, S_{1\ell}' \in C_1', S_{nk} \approx S_{1\ell}'$ where $1 \le k \le x, 1 \le \ell \le y$
- $\exists S_{ng} \in C_n$ s.t. $\forall S_{1h}' \in C_1', S_{ng} \not\approx S_{1h}'$, or $\exists S_{1h}' \in C_1', \forall S_{ng} \in C_n, S_{1h}' \not\approx S_{ng}$ where $1 \le g \le x, 1 \le h \le y$,

$C_a = (S_{a1}, \ldots, S_{ai}, \ldots, S_{ax})$,

$$S_{ai} = \begin{cases} e_{ni}^{+} & \text{if } \exists S_{ni} \approx S_{1\ell}' \text{ s.t. } (S_{ni} = e_{ni}) \wedge (S_{1\ell}' = e_{ni}) \\ & \quad \text{or } (S_{ni} = e_{ni}) \wedge (S_{1\ell}' = e_{ni}^{+}) \\ \varnothing & \text{if } \exists S_{ni} \approx S_{1\ell}' \text{ s.t. } (S_{ni} = e_{ni}^{-}) \wedge (S_{1\ell}' = e_{ni}) \\ & \quad \text{or } (S_{ni} = e_{ni}^{-}) \wedge (S_{1\ell}' = e_{ni}^{+}) \\ S_{ni} & \text{otherwise,} \end{cases}$$

$C_b = (S_{b1}, \ldots, S_{bj}, \ldots, S_{by})$,

$$S_{bj} = \begin{cases} e_{1j}^{-} & \text{if } \exists S_{nk} \approx S_{1j}' \text{ s.t. } (S_{nk} = e_{1j}) \wedge (S_{1j}' = e_{1j}) \\ & \quad \text{or } (S_{nk} = e_{1j}^{-}) \wedge (S_{1j}' = e_{1j}) \\ \varnothing & \text{if } \exists S_{nk} \approx S_{1j}' \text{ s.t. } (S_{nk} = e_{1j}) \wedge (S_{1j}' = e_{1j}^{+}) \\ & \quad \text{or } (S_{nk} = e_{1j}^{-}) \wedge (S_{1j}' = e_{1j}^{+}) \\ S_{1j}' & \text{otherwise.} \end{cases}$$

If both $\exists S_{ng} \in C_n$ s.t. $\forall S_{1h}' \in C_1', S_{ng} \not\approx S_{1h}'$ and $\exists S_{1h}' \in C_1', \forall S_{ng} \in C_n, S_{1h}' \not\approx S_{ng}$ where $1 \le g \le x, 1 \le h \le y$, a meet token "@" must be inserted between C_a and C_b, i.e., $Q_c \diamond_{par} Q_c' = \langle C_1, C_2, \ldots, C_{n\text{-}1}, C_a, @, C_b, C_2', \ldots, C_m' \rangle$.

Let us take eight coincidence sequences $Q_1, Q_2, \ldots, Q_8$ in Fig. 4.7 for example. In Fig. 4.7(a), when Q_1 appending Q_2, since the finishing time of Q_1 is different from the starting time of Q_2, we can just concatenate two coincidence sequences without modification (the case 1 in Definition 6). In Fig. 4.7(b), when Q_3 appending Q_4, since the finishing time of Q_3 is equal to the starting time of Q_4, and the slices in the last coincidence of Q_3 and in the first coincidence of Q_4 are all similar to each other, the concatenation of Q_3 and Q_4 is the entire coincidence-extension (the case 2 in Definition 6). By Definition 4.6, $(A^- B\ B\ E^-) \diamond_{ent} (A\ B^+ D\ E^+) = (A^- B^+ D)$, i.e., $A^- \diamond_{ent} A = A^-$, $B \diamond_{ent} B^+ = B^+$, and $D \diamond_{ent} D = D$. Note that, since $E^- \diamond_{ent} E^+ = E^*$, we need not presenting E^* in $(A^- B^+ D)$.

Actually, the partial coincidence-extension (the case 3 in Definition 6) has three conditions. As the coincidence sequences Q_5 and Q_6 in Fig. 4.7(c), since 1) the finishing time of Q_5 is equal to the starting time of Q_6, and 2) there are event slices, B^-, D, E and F^-, in the last coincidence of Q_5 similar to event slices, B, D^+, E and F^+ in the first coincidence of Q_6, respectively, and 3) an event slice A in the last coincidence of Q_5 is not similar to any slice in the first coincidence of Q_6, the concatenation of Q_5 and Q_6 is the partial coincidence-extension, i.e., $(A\ B^-D\ E\ F^-) \diamond_{par} (B\ D^+ E\ F^+) = (A\ D^+E^+)(B^-E^-)$. However, in Fig. 7(d), although the concatenation of Q_7 and Q_8 is also partial coincidence-extension, $(A\ B^-D\ E\ F^-) \diamond_{par} (B\ D^+E\ F^+G) = (A\ D^+E^+)\ @\ (B^-E^-G)$. Since slice A in last coincidence of Q_7 and slice G in the first coincidence Q_8 are not extended, we need to add token "@" to express meet relation between A and G.

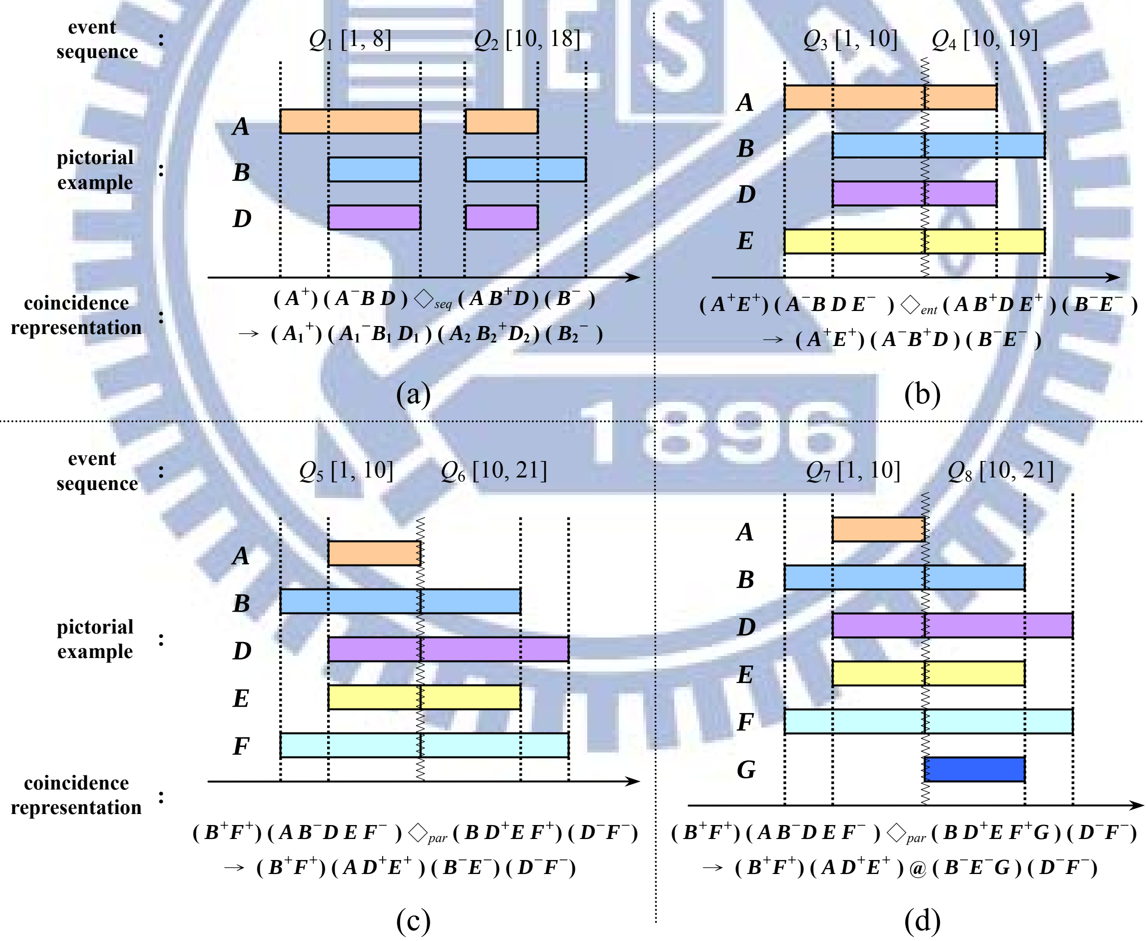


Fig. 4.7: An example of concatenation of two coincidence sequences

4.5.2 Proposed Algorithm: Inc_CTMiner

When a temporal database *DB* is updated to *DB'*, there are three possible cases for the temporal patterns in *DB'*,

Case 1: A pattern is frequent in *DB'*, and also frequent in *DB*.

Case 2: A pattern is frequent in *DB'*, and infrequent in *DB* but has a frequent pattern in *DB* as a prefix.

Case 3: A pattern is frequent in *DB'*, and infrequent in *DB* and has no any frequent patterns in *DB* as a prefix.

Case 1 is easy to handle since we have already stored the information of previous mining results into FPT_{DB}. We can obtain the temporal patterns in Case 1 by checking and adjusting the support of every pattern in FPT_{DB} in *DB'*. As the example database *DB* and *db* in Table 4.2, the temporal pattern ⟨(*A*)(*D*)⟩: 2 is frequent, where the notation "⟨*pattern*⟩ : *count*" represents the pattern and its associated support. And it is still frequent after updated.

Although we have not preserved any information of infrequent sequences in *DB*, in Case 2, all temporal patterns have at least one prefix subsequence which is frequent in *DB*, i.e., the frequent prefix is stored in FPT_{DB}. Hence, we can utilize every temporal pattern in FPT_{DB} as prefix to recursively discover the temporal patterns in Case 2. Since, in Case 3, the temporal patterns have no information stored in previous mining results, FPT_{DB}, we need to scan *DB'* for all new frequent 1-slices, and then use each new frequent 1-slice as prefix to construct projected database and recursively mine all temporal patterns in Case 3. For example, in Table 4.2, ⟨(*B*)(*F*)⟩: 2 is frequent after updated and has no frequent pattern in *DB* as prefix in FPT_{DB}.

Before introducing Inc_CTMiner algorithm, we first give an intuitive approach, *Naïve_Method*, for incremental mining temporal patterns. Naïve_Method will also be used for baseline comparisons to assess the merit of Inc_CTMiner later. Fig. 4.8 illustrates the pseudo code. It first determines the extended database, *EDB*, and uses ***incision_strategy*** to transform all event sequences in *DB'* to coincidence representation (Lines 1 and 2, algorithm 4.3). Then it calls ***CPrefixSpan***, which is the sub-procedure of *CTMiner*, on *EDB*, and store mined results in a pattern tree, PT_{EDB} (Line 3, algorithm 4.3). Note that, when mining *EDB*, the mined results should include both frequent and infrequent patterns, i.e., the *min_sup* is set as 1. Since even a pattern is infrequent in *EDB*, it still may become frequent in the updated database *DB'*. For each temporal pattern in FTP_{DB}, we update its support count if it also exists in PT_{EDB} and check whether it is still frequent in *DB'* (Lines 4-10, algorithm 4.3). Finally, we verify each remaining pattern in PT_{EDB} in $DB-EDB$ to adjust the support and output if it is frequent in *DB'* (Lines 11-17, algorithm 4.3).

Algorithm 4.3: *Naïve_Method* (*DB'*, *min_sup*, FPT_{DB})

Input: *DB'*: updated temporal database, *min_sup*: the minimum support,
FPT_{DB}: frequent pattern tree of original *DB*
Output: $FPT_{DB'}$: frequent pattern tree of updated database *DB'*
Variable: PT_{EDB} : pattern tree of *EDB*

01: determine *EDB* ;
02: use ***incision_strategy*** to transform *DB'* to coincidence presentation;
03: $PT_{EDB} \leftarrow$ ***CPrefixSpan*** (EDB, $\langle\,\rangle$, $1/|EDB|$, PT_{EDB});
// **sub-procedure of CTMiner**

04: **for each** node α in FPT_{DB} **do**
05: **if** $\alpha \in PT_{EDB}$
06: update support(α) and delete node α in PT_{EDB} ;
07: **if** support (α) $\geq$ ($min\_sup \times |DB'|$)
08: insert node α to $FPT_{DB'}$;
09: **else**
10: delete node α and all its descendent node in FPT_{DB} ;

11: scan $DB - EDB$ once for updating the support of node in PT_{EDB} ;
12: **for each** node β in PT_{EDB} **do**
13: **if** support(β) $\geq$ ($min\_sup \times |DB'|$)
14: insert node β to $FPT_{DB'}$;
15: **else**
16: delete node β and all its descendent node in PT_{EDB} ;
17: Output $FPT_{DB'}$;

Fig. 4.8: Pseudo code of Naïve_Method

In order to calculate the support of all patterns which are infrequent in *DB* but frequent in *DB'*, Naïve_Method keeps the information of all possible candidate set, i.e., mining *EDB* with *min_sup* = 1 (Line 3, algorithm 4.3). This awkward approach induces large memory usage and may involve many non-promising database projection. To remedy this problem, we design a more elegant algorithm, Inc_CTMiner, which performs two optimization techniques to reduce unnecessary space searches.

Definition 4.7 (Search Space Reduction)

Given a temporal pattern α in *DB* (node α in FPT_{DB}), when *DB* is updated to *DB'*, *incre_sid* is defined as a set of all sequence IDs in increment database *db* and $incre\_slice_{|\alpha}$ is defined as a set of all event slices in $db_{|\alpha}$. We have two kinds of search space reduction,

1) **Sequence-reduction:** If $\{\alpha$' s sequence list$\} \cap incre_sid = \varnothing$, then $DB_{|\alpha}$ is identical to $DB'_{|\alpha}$. The support of α and all temporal patterns prefixed with α, i.e., node α and all child nodes of α in FPT_{DB}, are unchanged in *DB'*. Hence there is no temporal pattern which is infrequent in *DB* but becomes frequent in *DB'* with α as prefix. We can stop searching α and all α's child nodes in FPT_{DB}.
2) **Slice-reduction:** If α' s parent node in in FPT_{DB} does not insert any node as child node when *DB* is updated to *DB'*, and the set of $\{\alpha$ and all α' s sibling nodes$\} \cap incre_slice_{|\alpha} = \varnothing$, then the support of α and all temporal patterns prefixed with α, i.e., node α and all child nodes of α in FPT_{DB}, are unchanged in *DB'*. Hence there is no temporal pattern which is infrequent in *DB* but becomes frequent in *DB'* with α as prefix. We can stop searching α and all child nodes of α in FPT_{DB}.

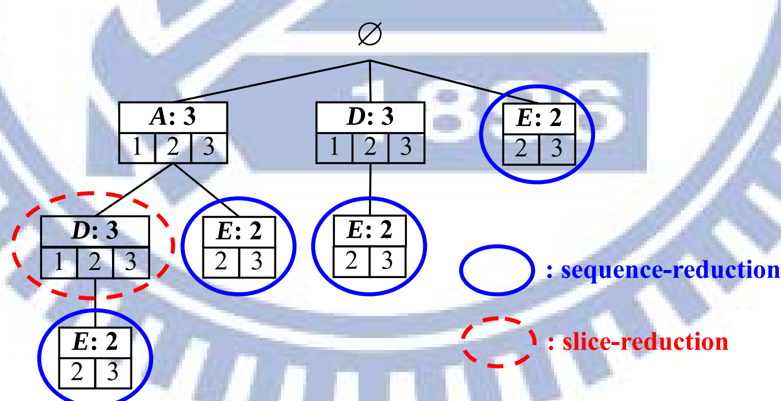


Fig. 4.9: The search space reduction on FPT_{DB} of example database *DB* in Table 4.2

Now we give an example to demonstrate the correctness of Definition 4.7. Given *DB* updated with *db* in Table 4.2 (*min_sup* = 2) and corresponding FTP_{DB} in Fig. 4.9, the *incre_sid* = {1, 4} and *incre_ slice* = $\{B, D, F, F^+, F^-, G\}$. By sequence-reduction, since all the *sequence_list*s of three nodes $(A)(D)(E)$, $(A)(E)$, $(D)(E)$ and (E) are {2, 3}, and $\{2, 3\} \cap$ *incre_sid* $= \{2, 3\} \cap \{1, 4\} = \varnothing$, we can stop searching these three nodes when discovering FTP_{DB+db}, as shown in Fig. 4.9. The *sequence_list* of node $(A)(D)$ is {1, 2, 3}. Hence, we cannot stop checking and growing the node $(A)(D)$ by sequence-reduction, due to $\{1, 2, 3\} \cap \{1, 4\} = \{1\} \neq \varnothing$. However, since the parent node of $(A)(D)$, i.e., node (A) does not insert any new child node and the set of $(A)(D)$ and $(A)(D)$'s sibling nodes $\cap$ $incre\_slice|_{\langle(A)(D)\rangle} = \{D, E\} \cap \{F, G\} = \varnothing$, we still can stop checking and growing node $(A)(D)$ and all its child nodes by the slice-reduction, as shown in Fig. 4.9.

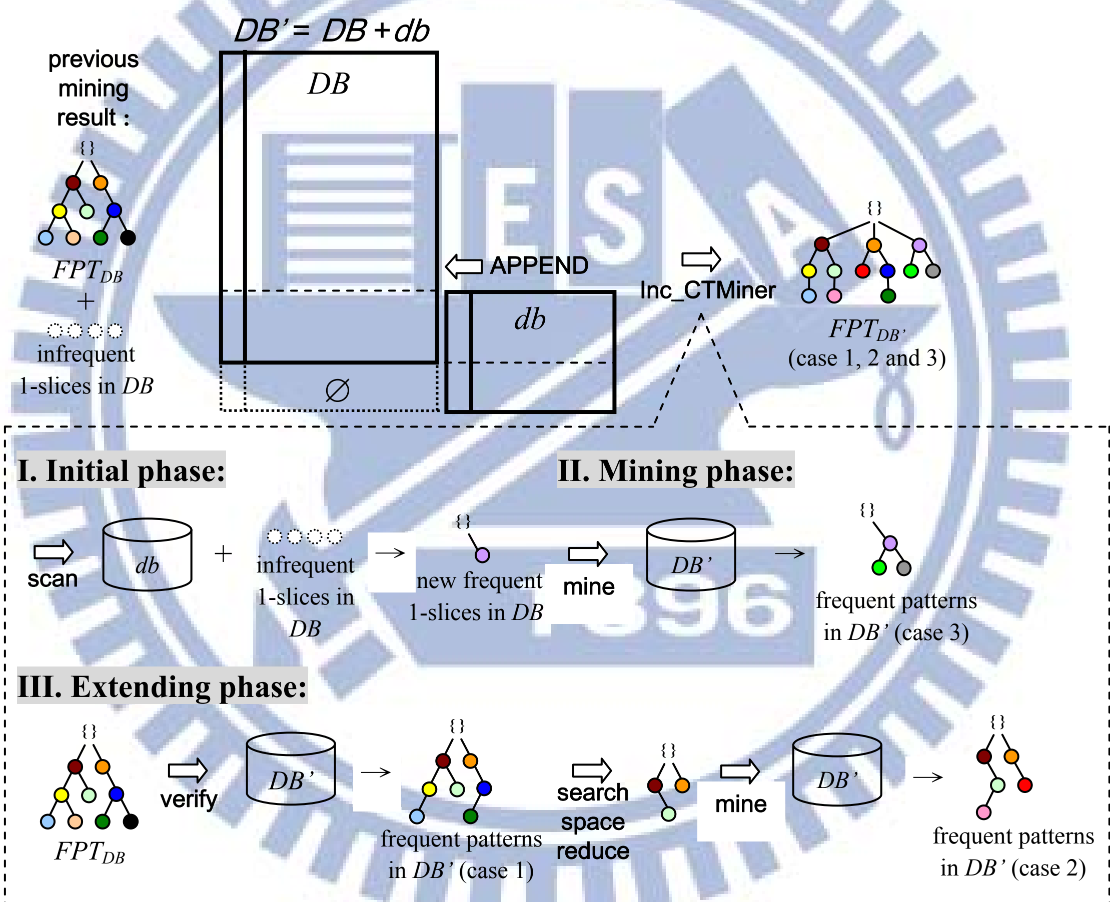


Fig. 4.10: An algorithmic overview of Inc_CTMiner

The search space reduction in Definition 4.7 plays an important role in Inc_CTMiner. When the minimum support goes lower and the maintained patterns turn to be longer, many unnecessary searches can be avoided effectively. As observed in our experiments, the search space reduction can skip more than 60% nodes in FPT_{DB}, especially when minimum support is extremely low. This is also the main reason why Inc_CTminer not only outperforms other algorithms in runtime performance, but also consumes less memory space. The algorithmic overview and the pseudo code of Inc_CTMiner are shown as in Fig. 4.10 and Fig. 4.11, respectively.

Algorithm 4.4: *Inc_CTMiner* (*DB'*, *min_sup*, FPT_{DB})

```
Input: DB' : updated temporal database, min_sup: the minimum support,
       FPT_DB : frequent pattern tree of original DB
Output: FPT_DB' : frequent pattern tree of updated database DB'

01: determine EDB; // initial Phase
02: use incision_strategy with interval_extension to transform DB' into
    coincidence presentation
03: NFS ← scan db and check infrequent 1-slices in DB for new frequent
    1-slices in DB' ; // frequent 1-slice in DB' ∉ FPT_DB

04: for each slice b in NFS do // mining phase
05:     insert b into FPT_DB' ;
06:     call Inc_CT (DB'|b , b , min_sup, FPT_DB' );

07: scan DB' once for update the support of node in FPT_DB ; // extending phase
08: for each node α in FPT_DB whose support ≥ ( min_sup×|DB'| ) do
09:     insert α into FPT_DB' ;;
10:     if search_pruning (α , DB'|α ) = " false " // search space pruning
11:         call Inc_CT (DB'|α , α , min_sup, FPT_DB' );
12: Output FPT_DB' ;

Procedure Inc_CT ( DB'|α , α , min_sup, FPT_DB' )
13: scan DB'|α once to find every frequent slice c ; // support ≥ ( min_sup×|DB'| )
14: for each slice c do
15:     if c is a "finish slice" then
16:         if exist corresponding start slice in α then // pre-pruning
17:             append c to α to form β;
18:     if c is a "start slice" or "intact slice" then
19:         append c to α to form β;
20: for each β not existed in FPT_DB do
21:     construct DB'|β with insignificant postfix elimination; // post-pruning
22:     if |DB'|β | ≥ ( min_sup×|DB'| ) then
23:         insert β into FPT_DB';
24:         if search_pruning (β , DB'|β ) = " false "    // search space pruning
25:             call Inc_CT (DB'|β , β , min_sup, FPT_DB' );
```

Fig. 4.11: Algorithm of Inc_CTMiner

There are three phases in Inc_CTMiner, initial phase, mining phase and extending phase. Initial phase first uses the incision strategy and considers the interval extension to transform all sequences into coincidence representation (Line 2, algorithm 4.4), and scans *db* once to discover all new frequent 1-slices in *DB'*. Notice that, due to the storing of infrequent 1-slices in *DB*, we can find the complete set of new frequent slices in *DB'* without rescanning *DB* again (Line 3, algorithm 4.4). Then, in mining phase, we use each new frequent slice as prefix to construct projected database and call sub-procedure *Inc_CT* to discover the temporal patterns (Lines 4-6 algorithm 4.4). Finally, in extending phase, Inc_CTMiner updates the support of every frequent pattern in *DB*. If a pattern is still frequent in *DB'*, we use *search_reduction* in Definition 7 to check if growing can stop. If not, sub-procedure *Inc_CT* is called to discover the temporal patterns (Lines 7-11, algorithm 4.4).

Sub-procedure *Inc_CT* recursively calls itself and works as follows. For a patter α as prefix, we scan its projected database $DB_{|\alpha}$ once to find its locally frequent slices (Line 13, algorithm 4.4) and adopt pre-pruning and post-pruning strategies to avoid non-promising projection (Lines 14-23, algorithm 4.4). We also use *search_reduction* to check whether growing can stop. If not, call *Inc_CT* recursively to discover the temporal patterns (Lines 24-25, algorithm 4.4).

4.6 Experimental Results and Performance Study

To evaluate the performance of Inc_CTMiner, one temporal pattern mining algorithms, CTMiner [8] and one incremental temporal pattern maintaining approach, Naïve method are compared with Inc_CTMiner. All algorithms were implemented in C^{++} language and tested on a computer with Pentium D 3.0 GHz with 2 GB of main memory. The performance study has been conducted on both synthetic and real world datasets. We perform three kinds of experiments in order to assess the efficiency of Inc_CTMiner. First, we compare the execution time and memory usage using synthetic datasets at extreme low minimum support. Second, we run Inc_CTMiner on different scenario to reflect the influence on performance of updated environments. Third, we conduct an experiment to observe the scalability on execution time of Inc_CTMiner. Finally, Inc_CTMiner is applied in real-world dataset, library lending data, to show the performance and the practicability of incremental maintenance for temporal patterns.

4.6.1 Data Generation

The synthetic data sets in the experiments are generated using synthetic generation program modified from [1]. Since the original data generation program was designed to generate time point-based data, the generator for the temporal pattern maintaining algorithm requires modifications on interval events and incremental scenario accordingly. The parameter setting of temporal data generator is shown in Table 4.5.

Table 4.5: Parameters of synthetic data generator

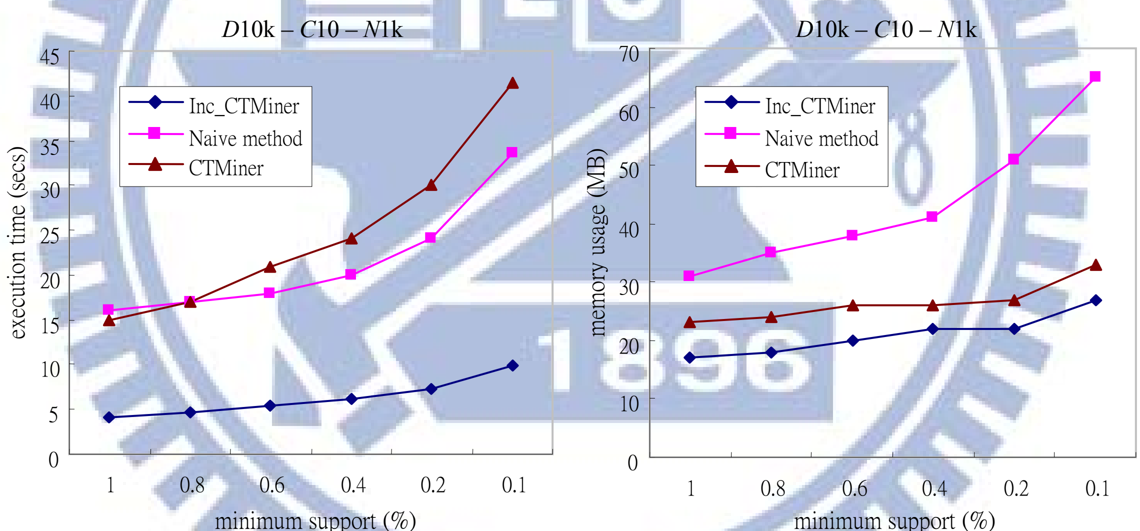
Parameters	Description
| *D* |	Number of event sequences
| *C* |	Average size of event sequences
| *S* |	Average size of potentially frequent sequences
N_S	Number of potentially frequent sequences
N	Number of event symbols
R_{inc}	Ratio of the number of sequences in increment database *db* to updated database *DB'*
R_{ext}	Ratio of the number of existed sequences extended to new sequences inserted in increment database *db*
R_{app}	Ratio of the number of intervals of an existed sequence appearing in original database *DB* to increment database *db*

The updated database *DB'* is generated first and then divided into the original database *DB* and increment database *db*. We create a set of potentially frequent sequences used in the generation of event sequences. The number of potentially frequent sequence is N_S. A potentially frequent sequence is generated by first picking the size of sequence from a Poisson distribution with mean equal to | *S* |. Then, the event intervals in potentially frequent sequence are chosen from *N* event symbols randomly. All the duration times of event intervals are classified into three categories: long, medium and short, which are normally distributed with an average length of 12, 8 and 4, respectively. For each event interval, we first randomly decide its category and then determine its length by drawing a value. The temporal relations between consecutive intervals are selected randomly to form a potentially frequent sequence. Since we adopt normalized temporal patterns [13], the temporal relationships can be chosen from the set {*before*, *meets*, *overlaps*, *is-finished-by*, *contains*, *starts*, *equal*}. After all potentially frequent sequences are determined, we generate | *D* | event sequences. Each event sequence is generated by first deciding the size of sequence, which was picked from a Poisson distribution with mean equal to $| C |$. Then, each event sequence is generated by assigning a series of potentially frequent sequences.

Finally, we partition the updated database *DB'* into the original database *DB* and increment database *db*, as the example in Fig. 4.1. Different settings of three parameters are used to reflect different updating scenarios. Parameter R_{inc}, called *increment ratio*, decides the size of the increment database *db*. We pick $| D | \times R_{inc}$ sequences randomly into *db* and place remaining $| D | \times (1 - R_{inc})$ sequences into *DB*. Furthermore, we use *extended ratio*, R_{ext}, to divide event sequences in *db* to "old" sequences, which's *sid* have appeared in *DB*, and "new" inserted sequences. Total $| db | \times R_{ext}$ sequences were randomly chosen from *db* as "old" sequence which were to be split further. The splitting of event sequences is to simulate that some intervals are conducted formerly (thus in *DB*), while the remaining intervals are newly appended (thus in *db*). The splitting is controlled by the third parameter R_{app}, the *appended ratio*. If a sequence with total *m* intervals is to split, we placed the leading $m \times (1 - R_{app})$ intervals in *DB* and the remaining $m \times R_{app}$ intervals in db_a. For example, a *DB'* with R_{inc} = 20%, R_{ext} = 30% and R_{app} = 40% means that 20% of sequences in *DB'* is in *db*; 30% of the sequences in *db* have *sid*s occurring in *DB*; and that for each "old" sequence, (1 – 40%) = 60% of intervals were conducted before database updating. Note that the calculation is integer-based with "ceiling" function.

4.6.2 Execution Time and Memory Usage on Synthetic Datasets

In all the following experiments, two parameters are fixed, i.e., the average size of potentially frequent sequences, $| S |$ = 4, and the number of potentially frequent sequences, N_S = 5,000. We set R_{inc} = 10%, R_{ext} = 50% and R_{app} = 20% to model common database updating scenario. The effect of various minimum supports on performance, including runtime and memory usage is evaluated. The first experiment for comparison of five algorithms is on the dataset *D*10k–*C*10–*N*1k with the minimum support thresholds varying from 0.01 % to 0.005 %. Obviously, re-mining from scratch with non-incremental algorithm is less efficient than using incremental maintaining algorithm, as illustrated in Fig. 4.12(a). When we continue to lower the minimum threshold, the runtime for TPrefixSpan and IEMiner increase drastically compared to CTMiner, Naïve method and Inc_CTMiner while Inc_CTMiner outperforms the other four algorithms. We can see that when the support is larger than 0.009 %, CTMiner outperforms Naïve method partly because of the generation of a fewer number of frequent patterns for the maintenance. When minimum support is 0.005 %, Inc_CTMiner is about 3 times faster than Naïve method, 4 times faster than CTMiner, about 10 times faster than IEMiner, more than 38 times faster than TPrefixSpan. The memory usages of five algorithms are showed as in Fig. 4.12(b). We can see that Inc_CMiner consume less memory than the other four algorithms. For example, when minimum support threshold is reduced to 0.005%, Inc_CTMiner consumes 27 MB which is more than 1.2 times smaller than CTMiner (33 MB), more than 1.7 times smaller than TPrefixSpan (48 MB), about 2.4 times smaller than Naïve method (65 MB), and almost 5.8 times smaller than IEMiner (104 MB).

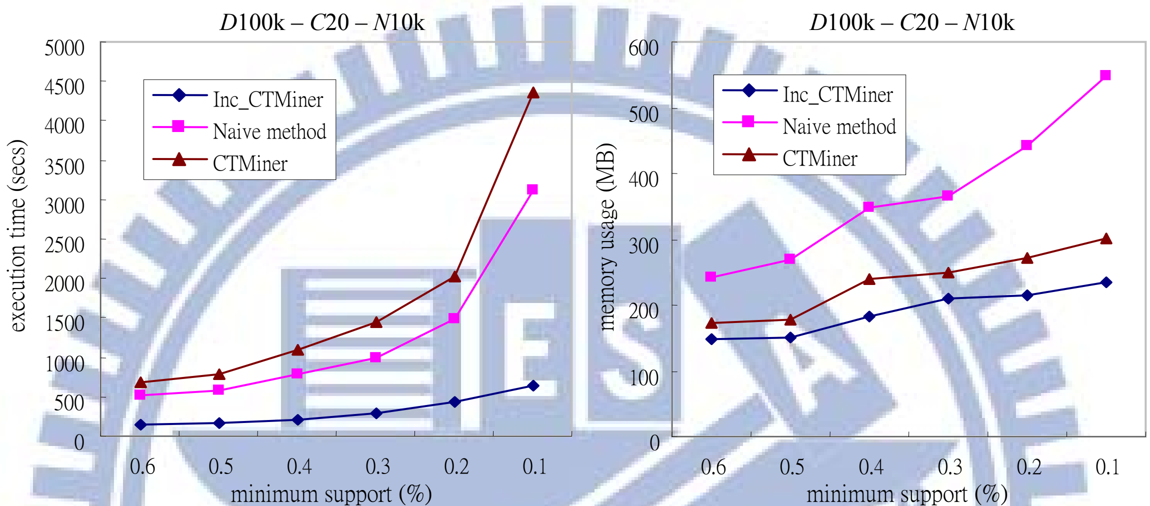


(a) The execution time of three algorithms (b) The memory usage of three algorithms

Fig. 4.12: The performance on data set D10k – C10 – N1k (with R_{inc} = 10%, R_{ext} = 50% and R_{app} = 20% updating scenario)

The second experiment is performed on data set D100k–C20–N10k, which contains 100,000 event sequences, average length 40 and 10,000 event intervals with common database updating scenario. The execution time of different algorithms is shown in Fig. 4.13(a). We can see that when the support is 0.005%, Inc_CTMiner takes 610 seconds, which is more than 2.4 times faster than Naïve method (1515 sec.), more than 4.1 times faster than CTMiner (2526 sec.), about 10.5

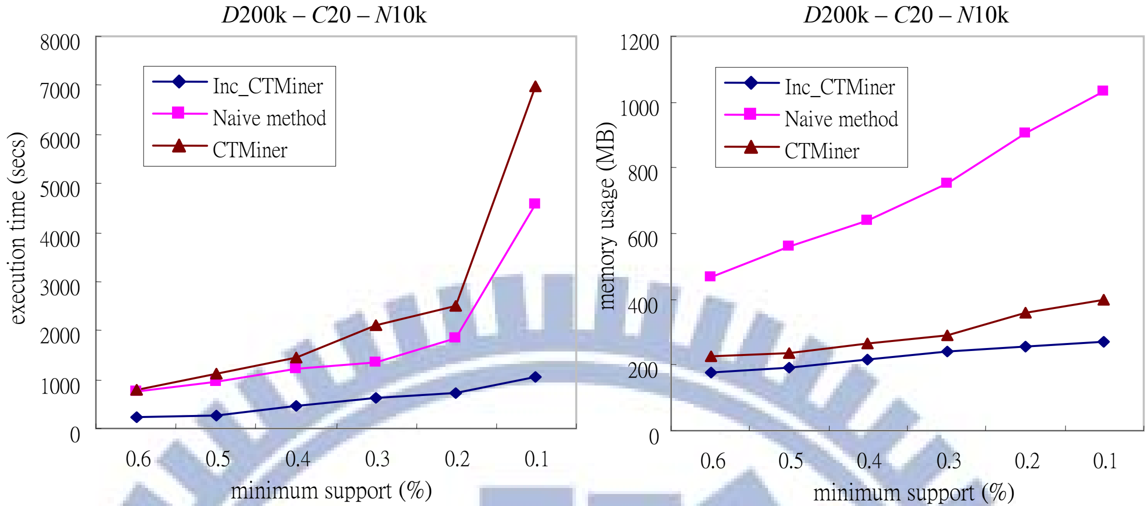
times faster than IEMiner (6439 sec.), about 38 times faster than TPrefixSpan (23232 sec.). Fig. 4.13(b) shows the memory usages of five algorithms with different minimum support thresholds. We can see that although Naïve method has better performance on execution time than re-running CTMiner from scratch, it involves larger memory space for execution partly because of storing every possible frequent sequences and doing many non-promising database projection.



(a) The execution time of three algorithms (b) The memory usage of three algorithms

Fig. 4.13: The performance on data set *D*100k – *C*20 – *N*10k (with R_{inc} = 10%, R_{ext} = 50% and R_{app} = 20% updating scenario)

The third performance measurement is performed on a larger data set *D*200k–*C*20–*N*10k. The data set contains a large number of temporal patterns when minimum support is reduced to 0.005 %. Fig. 4.14(a) illustrates the execution time of different algorithms at different minimum supports. When minimum support lowers to 0.005%, Inc_CTMiner takes 1,759 sec., which is almost 2 times faster than Naïve method (3371 sec.), more than 3.3 times faster than CTMiner (5804 sec.), about 10 times faster than IEMiner (17543 sec.), more than 23.5 times faster than TPrefixSpan (41364 sec.). Fig. 4.14(b) shows the results of memory consuming, from which we can observe that Inc_CMiner is not only more efficient, but also more stable in memory usage than the other four algorithms. For example, when minimum support threshold is reduced to 0.005%, Inc_CTMiner consumes 271 MB which is more than 1.4 times smaller than CTMiner (397 MB), about 3.8 times smaller than Naïve method (1,031 MB), about 4.7 times smaller than TPrefixSpan (1,294 MB) and almost 5.8 times smaller than IEMiner (1,579 MB).



(a) The execution time of three algorithms (b) The memory usage of three algorithms

Fig. 4.14: The performance on data set *D*200k – *C*20 – *N*10k (with R_{inc} = 10%, R_{ext} = 50% and R_{app} = 20% updating scenario)

Three experiments above indicate that, when some sequences are appended and some new sequences are inserted, even with an extremely low minimum support and a large number of temporal patterns, Inc_CTMiner algorithm is still efficient and outperforms other algorithms in both execution time and memory usage.

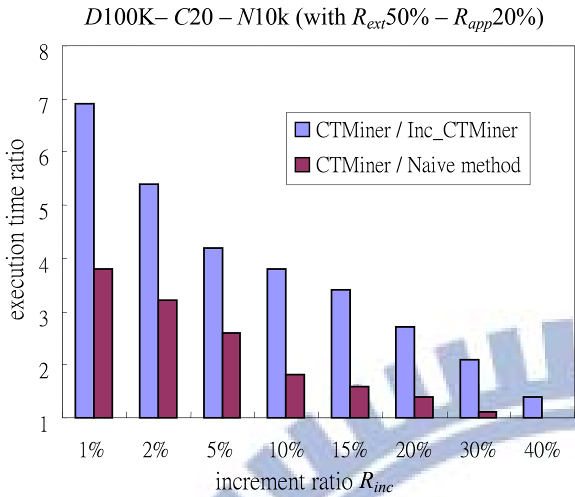
4.6.3 Performance on Different Updating Scenario

In this section, in order to reflect the influence of incremental environment on time performance, three parameters, increment ratio, extended ratio and appended ratio, are configured to generate different updating scenarios for comparing the execution times of five algorithms. Generally, incremental maintaining algorithms gain less at higher increment ratio because larger increment ratio means more sequences appearing in *db* and causes more pattern updates. If most of the frequent sequences in *DB* turn out to be invalid in *DB'*, the information stored by maintenance algorithms in pattern updating might become useless. Fig. 4.15 is the results of varying increment ratio, R_{inc}, from 1% to 40% on *D*100k – *C*20 – *N*10k. The *min_sup* is fixed at 0.01%. Note that we use the execution time ratio to show the improvement of incremental maintaining algorithms over CTMiner, i.e., the execution time of incremental maintaining algorithm / the execution time of Inc_CTMiner. As indicated in Fig. 4.15(a), the smaller the increment database *db* is, the more time Inc_CTMiner could save. Inc_CTMiner is still faster than CTMiner even when R_{inc} reaches 40%. When R_{inc} becomes much larger, say over 40%, Inc_CTMiner is slower than CTMiner. When the size of the increment database becomes larger than the size of the original database, i.e. the database has accumulated dramatic change, re-mining from scratch might be a better choice for the totally new sequence database.

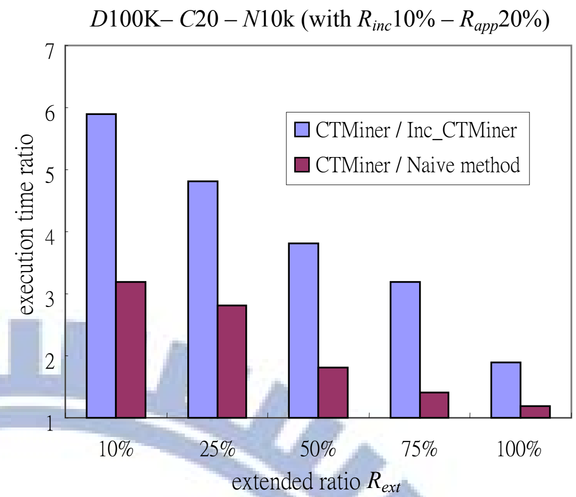
The impact of the extended ratio, R_{ext}, is presented in Fig. 4.15(b) on D100k – C20 – N10k dataset with min_sup = 0.01%. Note that, for better illustration, we adopt the execution time ratio to show the improvement of incremental maintaining algorithms over CTMiner. As shown in Fig. 16, Inc_CTMiner updates patterns more efficiently than Naïve method and CTMiner. Higher R_{ext} means that there are more event sequences in the original database expended in the increment database. Consequently, the speedup ratio decreases as the R_{ext} increases because more appended sequence need to be processed. We can observe that Inc_CTMiner is efficient even when the R_{ext} is increased to 100%, i.e., all the sequences in the increment database are extended from original database. Fig. 4.15(c) depicts the performance comparisons of Inc_CTMiner and Naïve method with CTMiner concerning appended ratios, R_{app}, on D100k – C20 – N10k dataset. We can see from the figure that Inc_CTMiner is constantly about 5.3 times faster than CTMiner over various R_{app}, ranging from 10% to 90%.

4.6.4 Scalability Studies

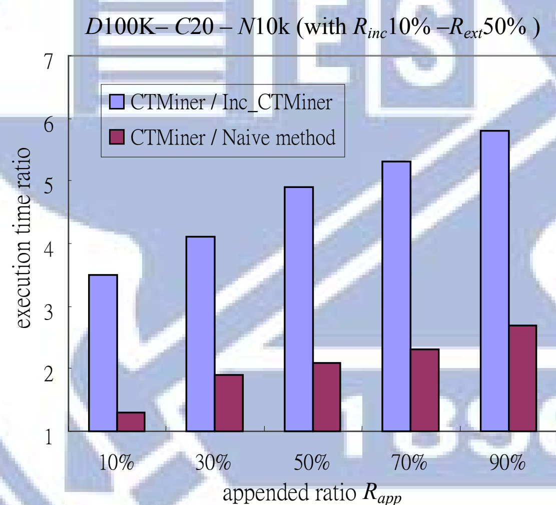
In the following experiments, we study the scalability on the execution time of Inc_CTMiner algorithm. Here, the total number of event sequences is increased from 100K to 500K, with fixed parameters C = 20, N = 10k, R_{inc} = 10%, R_{ext} = 50% and R_{app} = 20%. Fig. 4.16(a) shows the results of scalability tests of the Inc_CTMiner algorithm, with different minimum support threshold varying from 0.03 % to 0.01 %. As the size of database increases and minimum support decreases, the processing time of Inc_CTMiner increases, since the number of patterns maintained also increases. As can be seen, under different minimum support threshold, Inc_CTMiner is still linearly scalable with different database size.



(a) Total execution time over various increment ratios



(b) Total execution time over various extended ratios

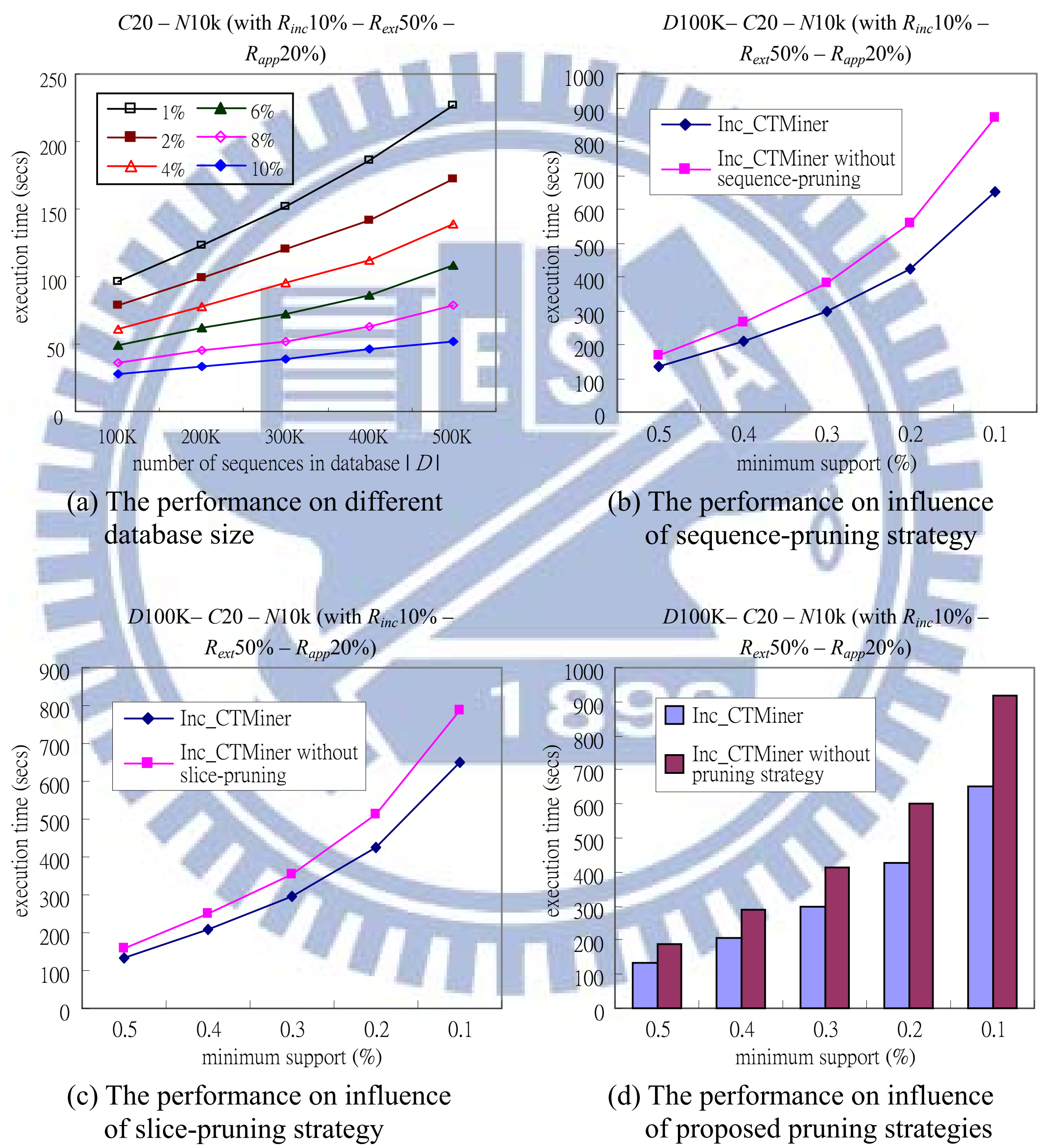


(c) Total execution time over various appended ratios

Fig. 4.15: Total execution time with various increment ratios, extended ratios and appended ratios

4.6.5 Impact of Pruning Strategy

In this section, to reflect the speedup of proposed pruning methods, we measure the Inc_CTMiner with two pruning strategies and without pruning strategy on time performance. The experiment is performed on the data set *D*100k–*C*20–*N*10k, which contains 100,000 event sequences, the average length of sequence is 20 and the number of events is 10,000. Fig. 4.16(b) is the results of varying minimum support thresholds from 0.5 percent to 0.1 percent. As shown in Figure, sequence-pruning strategy can improve 25.6% to 33.8% of the performance of Inc_CTMiner. Because of removing unnecessary sequences before maintenance, sequence-pruning can efficiently speedup the execution time.



(a) The performance on different database size

(b) The performance on influence of sequence-pruning strategy

(c) The performance on influence of slice-pruning strategy

(d) The performance on influence of proposed pruning strategies

Fig. 4.16: The performance on different database size and on influence of proposed pruning strategies

The impact of the slice-pruning strategy is presented in Fig. 4.16(c). As can be seen from the graph, when Inc_CTMiner is without slice-pruning, the execution time is about 21.2% slower than Inc_CTMiner in average. We can find that slice-pruning strategy can improve the performance of Inc_CTMiner by effectively eliminating all useless sequences for maintaining temporal pattern. Fig. 4.16(d) depicts the influence on two proposed pruning strategies. We can see that Inc_CTMiner is constantly about 40.2% faster than the one without any pruning strategy. Consequently, the proposed pruning strategies not only effectively reduce the searching space but also efficiently improve the performance of Inc_CTMiner.

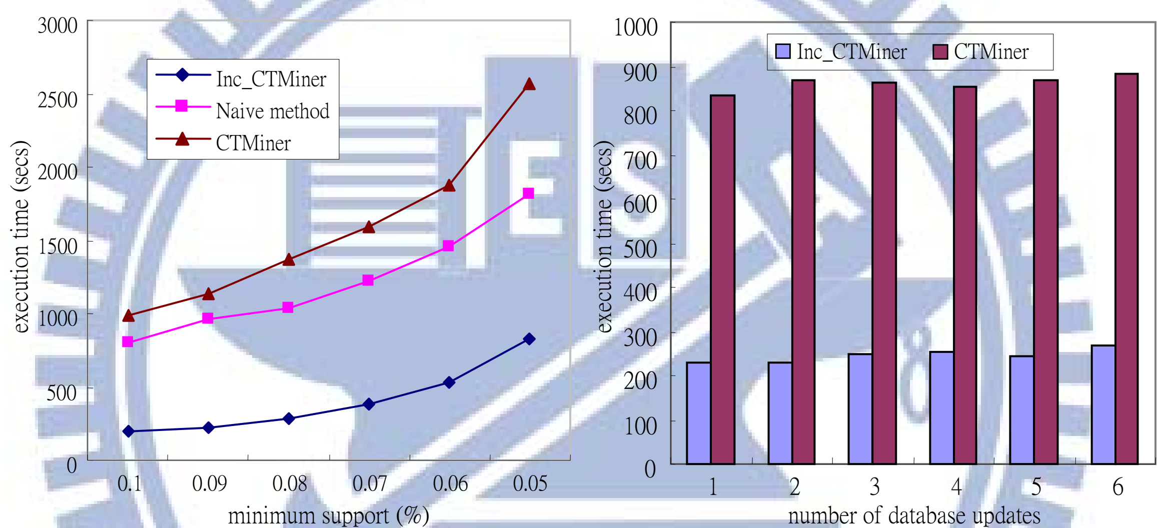
In summary, our performance study shows that Inc_CTMiner has the best overall performance among the algorithms tested. The memory usage analysis shows the efficient memory consumption of Inc_CTMiner. The scalability study also shows that proposed algorithm scales well even with large databases and low thresholds.

4.6.6 Real Dataset Analysis

In addition to using synthetic data sets, we have also performed an experiment on real world data set to compare the performance and indicate the applicability of temporal pattern mining. The database used in this experiment consists of a collection of 1,098,142 library records, includes lending and returning records, for three years from the National Chiao Tung University Library. The database includes 206,844 books and 28,339 readers. An event interval is composed by a book ID and corresponding lending and returning time. The size of database is the number of sequences in database (same as the number of readers, 28,339). The maximum and the average length of sequences are 302 and 36, respectively. First, we collect the records of first two and half years to construct the original database *DB* and use the record of last half year to build the increment database *db*. The *DB* with 1,053,276 library records can be viewed as 26,738 user sequences and the *db* with 44,866 library records can be viewed as 3,514 user sequences. Fig. 4.17(a) shows the performance of execution time with varying minimum support thresholds from 0.1 % to 0.05 %, respectively. As the minimum support drops down to 0.05 %, Inc_CTMiner is almost 2 times faster than Naïve method and more than 2.7 times faster than CTMiner.

Finally, we discuss the performance of Inc_CTMiner to process multiple database updates.

We still use the records of first two and half years to construct *DB* and divide the records of the rest half years by every one month to build six different *db*. Fig. 4.17(b) shows the performance of Inc_CTMiner, with *min_sup* = 0.1%, to incrementally maintain multiple database updates, i.e., 6 months, six updates in this case. Each time the database is updated, we also run CTMiner to re-mine from scratch for comparison. We can see from the figure, when the increments accumulate, the time for incremental mining also increases, but increase is very small. The incremental mining still outperforms re-mining with CTMiner by a factor of 2.5 or 3.5. This experiment shows that Inc_CTMiner is really efficient for multiple updates of database.



(a) Performance of three algorithms on library dataset from NCTU

(b) Multiple updates of library dataset from NCTU

Fig. 4.17: Execution time of three algorithms and multi updates on library dataset from NCTU

4.7 Summary

Previous studies of updating sequential pattern mainly are focused on time point-based data. Little attention has been paid to the incremental mining of temporal patterns from time interval-based data. Since the processing for complex relations among intervals may require generating and examining large amount of intermediate subsequences, maintaining temporal patterns from time interval-based data is a challenging problem. In this chapter, we investigate the issue for incremental mining the temporal patterns. **Inc_CTMiner** is proposed to balance the efficiency and reusability based on a proper expression, coincidence representation. The algorithm also employs two optimization techniques, sequence-reduction and slice-reduction, to further reduce the search space effectively. The experimental results indicate that both execution time and memory usage of Inc_CTMiner outperform previous algorithms designed based on static database. We also show the graceful scalability of Inc_CTMiner. Furthermore, we apply the algorithm on real world dataset to show the efficiency and the practicability of maintaining temporal patterns.

Chapter 5

Conclusion

In this dissertation, we propose two new representations, **coincidence representation** and **endpoint representation** to simplify the processing of complex relations among event intervals. Then, three efficient algorithms are developed to discover several types of temporal patterns from interval-based data. These algorithms employ some pruning techniques to reduce the search space effectively. The experimental studies indicate that all proposed algorithm is efficient and scalable and outperforms state-of-the-art algorithms. Furthermore, we also apply our algorithms on real world data to show the efficiency and validate the practicability of interval-base temporal mining.

In Chapter 2, a novel technique, **incision strategy** and a new representation, **coincidence representation** are proposed to remedy the critical issue of temporal pattern mining. We simplify the processing of complex relations among event intervals effectively. Coincidence representation is nonambiguous and has several advantages over existing representations. Based on coincidence representation, we develop an efficient algorithm, **CTMiner** to discover frequent temporal patterns without candidate generation. The algorithm further employs two pruning techniques, pre-pruning and post-pruning, to reduce the search space effectively. By analyzing the differences between mining sequential patterns and temporal patterns, we also propose a new projection technique, **multi-projection** to correctly project a database into a set of smaller projected databases. The experimental studies indicate that CTMiner is efficient and scalable. Both running time and memory usage of CTMiner outperform state-of-the-art algorithms.

Previous studies of mining closed sequential pattern mainly are focused on time point-based data. Little attention has been paid to the mining of closed temporal patterns from time interval-based data. Since the processing for complex relations among intervals may require generating and examining large amount of intermediate subsequences, mining closed temporal patterns from time interval-based data is an arduous problem. In Chapter 3, we develop an efficient algorithm, **CEMiner**, to discover closed temporal patterns without candidate generation, based on proposed endpoint representation. The algorithm further employs three pruning methods, pre-pruning, post-pruning and pair-pruning, to reduce the search space effectively. The experimental studies indicate that CEMiner is efficient and scalable. Both running time and memory usage of CEMiner outperform the state-of-the-art algorithms. Furthermore, we also apply CEMiner on real world dataset to show the efficiency and the practicability of mining time interval-based closed pattern.

Little attention has been paid to the incremental mining of temporal patterns from time interval-based data. Since the processing for complex relations among intervals may require generating and examining large amount of intermediate subsequences, maintaining temporal patterns in interval-based database is a challenging problem. In Chapter 4, we investigate the issue for incremental mining of the temporal patterns. **Inc_CTMiner** is proposed to balance the efficiency and reusability based on a proper expression, coincidence representation. The algorithm also employs two optimization techniques, sequence-reduction and slice-reduction to further reduce the search space effectively. The experimental results indicate that both execution time and memory usage of Inc_CTMiner outperform previous algorithms designed based on static database. We also show the graceful scalability of Inc_CTMiner. Furthermore, we apply the algorithm on real world dataset to show the efficiency and the practicability of maintaining time interval-based patterns.

Bibliography

[1] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proceedings of 11th International Conference on Data Engineering (ICDE'95)*, pp. 3-14, 1995.

[2] J. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of ACM*, vol.26, issue 11, pp.832-843, 1983.

[3] J. Ayres, J. Gehrke, T. Yu, and J. Flannick, "Sequential Pattern Mining Using a Bitmap Representation," *The 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'02)*, pp. 429-435, 2002.

[4] L. Chang, T. Wang, D. Yang and H. Luan, "SeqStream: Mining Closed Sequential Patterns over Stream Sliding Windows," *International Conference on Data Mining (ICDM'08)*, pp. 83-92, 2008.

[5] L. Chang, T. Wang, D. Yang, H. Luan and S. Tang, "Efficient algorithms for incremental maintenance of closed sequential patterns in large databases," *Data & Knowledge Engineering*, vol. 68, issue 1, pp. 68-106, 2009.

[6] J. Chen, "An Up Down Directed Acyclic Graph Approach for Sequential Pattern Mining," *IEEE Transactions on Knowledge and Data Engineering*, vol.22, no. 7, pp.913-928, 2010.

[7] Y. Chen, J. Guo, Y. Wang, Y. Xiong and Y. Zhu, "Incremental Mining of Sequential Patterns using Prefix Tree," *The 11th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'07)*, pp. 433-440, 2007.

[8] Y. Chen, J. Jiang, W. Peng and S. Lee, "An Efficient Algorithm for Mining Time Interval-based Patterns in Large Databases," *19th ACM International Conference on Information and Knowledge Management (CIKM'10)*, pp 49-58, 2010.

[9] H. Cheng, X. Yan and J. Han, "IncSpan: incremental mining of sequential patterns in large database," *The 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'04)*, pp.527-232, 2004.

[10] M. Garofalakis, R. Rastogi, and K. Shim, "SPIRIT: Sequential Pattern Mining with Regular Expression Constraints," *25th International Conference on Very Large Data Bases (VLDB '99)*, pp. 223-234, 1999.

[11] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. Hsu, "FreeSpan: Frequent

Pattern-Projected Sequential Pattern Mining," *The 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'00)*, pp. 355-359, 2000.

[12] C. Ho, H. Li, F. Kuo and S. Lee, "Incremental Mining of Sequential Patterns over a Stream Sliding Window," *International Conference on Data Mining - Workshops (ICDMW'06)* pp.677-681, 2006.

[13] F. Hoppner, "Finding informative rules in interval sequences," *Intelligent Data Analysis*, vol. 6, no. 3, pp. 237-255, 2002.

[14] J. Huang, C. Tseng, J. Ou, and M. Chen, "A General Model for Sequential Pattern Mining with a Progressive Database," *IEEE Transactions on Knowledge and Data Engineering*, vol.20, issue 9, pp. 1153-1167, 2008.

[15] K. Huang, C. Chang, J. Tung, C. Ho, "COBRA: closed sequential pattern mining using bi-phase reduction approach," *Proceedings of the 2006 International Conference on Data Warehousing and Knowledge Discovery (DaWaK'06)*, pp. 280-291, 2006.

[16] P. Kam and W. Fu, "Discovering Temporal Patterns for Interval-based Events," *International Conference on Data Warehousing and Knowledge Discovery (DaWaK'00),* vol. 1874, pp. 317-326, 2000.

[17] S. Laxman, P Sastry and K. Unnikrishnan, "Discovering Frequent Generalized Episodes When Events Persist for Different Durations," *IEEE Transactions on Knowledge and Data Engineering*, vol.19, issue 9, pp. 1188-1201, 2007.

[18] M. Lin, S. Hsueh, and C. Chang, "Fast discovery of sequential patterns in large databases using effective time-indexing," *Information Sciences: An International Journal*, vol. 178/22, pp. 4228-4245, 2008.

[19] M. Lin and S. Lee, Incremental update on sequential patterns in large databases by implicit merging and efficient counting, *Information Systems*, vol. 29, issue 5, pp. 385-404, 2004.

[20] M. Lin and S. Lee, "Fast Discovery of Sequential Patterns by Memory Indexing and Database Partitioning," *Journal of Information Sciences and Engineering*, Vol. 21, No. 1, pp. 109-128, 2005.

[21] H. Mannila, H. Toivonen, and I. Verkamo, "Discovery of frequent episodes in event sequences," *Data Mining and Knowledge Discovery*, vol. 1, issue 3, pp. 259-289, 1997.

[22] F. Masseglia, F. Cathala and P. Poncelet, "The PSP Approach for Mining Sequential Patterns," *European Conference on Principles of Data Mining and Knowledge Discovery*

(PKDD'01), vol. 1510, pp176-184, 1998.

[23] F. Masseglia, P. Poncelet and M. Teisseire, "Incremental mining of sequential patterns in large databases," *Data & Knowledge Engineering*, vol.46, issue 1, pp.97–121, 2003.

[24] F. Morchen and A. Ultsch, "Efficient Mining of Understandable Patterns from Multivariate Interval Time Series," *Data Mining Knowledge Discovery*, vol. 15, number 2, pp.181-215, 2007.

[25] F. Morchen and D. Fradkin, "Robust mining of time intervals with semi-interval partial order patterns," *Proceedings of 10th SIAM International Conference on Data Mining (SDM'10)*, pp.315-326, 2010.

[26] S. Nguyen, X. Sun, M. Orlowska, "Improvements of IncSpan: Incremental Mining of Sequential Patterns in Large Database," *The 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'05)*, pp. 442-451, 2005.

[27] P. Papapetrou, G. Kollios, S. Sclaroff, and D. Gunopulos, "Discovering frequent arrangements of temporal intervals," *International Conference on Data Mining (ICDM'05)*, pp. 354-361, 2005.

[28] S. Parthasarathy, M. Zaki, M. Ogihara, and S. Dwarkadas, "Incremental and interactive sequence mining," *Proceedings of the 8th International Conference on Information and Knowledge Management (CIKM'99)*, pp. 251-258, 1999.

[29] D. Patel, W. Hsu and M. Lee, "Mining Relationships Among Interval-based Events for Classification," *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 393-404, 2008.

[30] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal and M. Hsum, "Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 10, pp.1424-1440, 2004.

[31] C. Rainsford and J. Roddick, "Adding temporal semantics to association rules," *In Proceedings of the 3rd European conference on principles and practice of knowledge discovery in databases (PKDD'99)*, pp. 504-509, 1999.

[32] R. Srikant and R. Agrawal, "Mining Sequential patterns: Generalizations and Performance Improvements," *Proceedings of 5th International Conference on Extended Database Technology (EDBT'96)*, pp. 3-17, 1996.

[33] R.Villafane, K. Hua and D. Tran, "Knowledge Discovery from Series of Interval Events,"

Journal of Intelligent Information Systems, vol.15, pp.71-89, 2000.

[34] J. Wang, J. Han, “BIDE: Efficient mining of frequent closed sequences,” *Proceedings of the 20th International Conference on Data Engineering (ICDE’04)*, pp. 79-90, 2004.

[35] E. Winarko and J.F Roddick, “ARMADA-An algorithm for discovering richer relative temporal association rules from interval-based data,” *Data & Knowledge Engineering*, vol. 63, issue 1, pp. 76-90, 2007.

[36] S. Wu and Y. Chen, “Mining Nonambiguous Temporal Patterns for Interval-Based Events,” *IEEE Transactions on Knowledge and Data Engineering*, vol.19, issue 6, pp. 742-758, 2007.

[37] S. Wu and Y. Chen, “Discovering hybrid temporal patterns from sequences consisting of point- and interval-based events,” *Data & Knowledge Engineering*, vol.68, issue 11, pp.1309–1330, 2009.

[38] X. Yan, H. Cheng, J. Han and D. Xin, “CloSpan: Mining Closed Sequential Patterns in Large Datasets,” *Proceedings of 3rd SIAM International Conference on Data Mining (SDM’03)*, pp 166-177, 2003.

[39] M. Zaki, “SPADE: An Efficient Algorithm for Mining Frequent Sequences,” *Machine Learning*, vol. 42, numbers 1-2, pp. 31-60, 2001.

[40] M. Zaki and C. Hsiao, “CHARM: An Efficient algorithm for Closed Itemset Mining,” *Proceedings of 2nd SIAM International Conference on Data Mining (SDM’02)*, pp. 457-478, 2002.

[41] L. Zhang, G. Chen, T. Brijs and X. Zhang, “Discovering during-temporal patterns (DTPs) in large temporal databases,” *Expert Systems with Applications*, vol. 34, pp.1178-1189, 2008.

[42] M. Zhang, B. Kao, D. Cheung, and C. Yip, “Efficient algorithms for incremental updates of frequent sequences,” *The 6th Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD’02)*, pp.186-197, 2002.