時間區間事件之有效率的循序樣式探勘

研究生：姜季強　　　　　　　　　　　　　　指導教授：李素瑛

國立交通大學資訊科學與工程研究所

摘要

現在已經發展的循序樣式探勘演算法皆假設事件的發生是在時間點上。然而，在現實生活上發生的事件通常是持續一段時間的，稱之爲“以時間區間爲基礎的事件”。但是由於時間區間事件間複雜的關係，造成了在設計有效率以時間區間事件爲基礎之循序樣式探勘演算法上的困難。因此，我們提出了“同時發生的事件片段”的概念來解決時間區間事件間複雜關係的問題。首先根據時間區間的事件間“同時發生”的部份將時間區間事件切割成互斥的更小事件片段，即“同時發生”的一段時間區間內可能有許多事件片段，而原本的事件序列可表示成我們所提出新的事件序列表示方式：以“同時發生”時間排列的有序序列，稱之爲“同時發生事件片段序列表示法”。因此，我們考慮事件片段間的相互關係變地相當簡單，即前後、同時。我們提出一個演算法 CTMiner 基於“同時發生事件片段序列表示法”來表示事件序列並利用知名的循序樣式探勘演算法 PrefixSpan 的概念來找出頻繁的時間區間事件循序樣式，並能完全避免產生候選樣式。最後，爲了能理解頻繁的“同時發生事件片段序列”樣式的意義，我們利用關係序列來呈現此頻繁樣式中時間區間事件間所有的關係。並且，我們還根據“同時發生的事件片段”的特性，設計了一些策略來提升CTMiner演算法的效率。在實際的圖書館借閱資料和合成資料的實驗結果皆表現出此演算法的效率和適應性。

檢索詞：以時間區間爲基礎的事件之探勘、同時發生的事件片段、時間樣式

An efficient interval-based sequential pattern mining

Student: Ji-Chiang Jiang Advisor: Suh-Yin Lee

Institute of Computer Science and Information Engineering
National Chiao-Tung University

Abstract

Existing sequential pattern mining algorithms assume that events occur instantaneously. However, events in real world applications usually have durations which are called interval-based events. But complex relationship among event intervals causes difficulty in designing an efficient interval-based event mining algorithm. Therefore, the concept of "coincidence-slice" is proposed to solve the problem caused by the complex relationship among event intervals. The event intervals are incised to disjoint smaller "event slices" according to the coincidences among event intervals, that is, several event slices may occur in the same time period called "coincidence". Therefore, an original event sequence can be represented as a list of ordered "coincidences" which contains event slices. This new representation proposed is called "coincidence sequence representation". We transform the problem of complex relationship among event interval to consider the simple relationship among event slices. The proposed interval-based sequential pattern mining algorithm called CTMiner is based on the "coincidence sequence representation". The CTMier also uses the concept of well-known sequential pattern mining algorithm PrefixSpan to find temporal patterns without candidate generation. Finally, to comprehend the frequent temporal pattern represented by "coincidence sequence representation", we discover and use relation list to present all the relationships in a pattern. We also implement some pruning strategies to improve the performance of CTMiner by considering the characteristics of the "Coincidence-slice". Experiments on both synthetic datasets and real dataset of library

lending indicate the efficiency and scalability of the proposed algorithm.

Index terms: Interval-based event mining, coincidence-slice, temporal patterns

Acknowledgement

I greatly appreciate the kind guidance of my advisor, Prof. Suh-Yin Lee. She not only helps with my research but also inspires and takes care of me. Without her graceful suggestion and encouragement, I cannot complete the thesis. Besides I want to give my thanks to all members in the Information System Laboratory for their suggestion and instruction, especially Mr. Yi-Cheng Chen, Miss Yu-Jiun Liu, Miss Chao-Ying Wu and Mr. Chang-Yei Pong. I would especially express my thanks to the nicest and most beautiful lady Miss Yi-chien Lee who accompanies me and lights up my life. Finally I would like to express my deepest appreciation to my parents. This thesis is dedicated to them.

Table of contents

Abstract (Chinese)……………………………………………………………………...i

Abstract (English) …………………………………………… ……………….....iii

Acknowledgement……………………………………………………………………v

Table of Contents…………………………………………………………………vi

List of Figures……………………………………………………………………viii

List of Tables………………………………………………………………………xi

Chapter 1 Introduction………………………………………………………………1

Chapter 2 Motivation and Related Works……………………………………………4

2.1 Motivation…………………………………………………………………………4

2.1.1 Representations of a temporal pattern……………………………………………4

2.1.2 The Problems of Complex Relationship on Temporal Pattern Mining Approaches………………………………………………………………………7

2.2 Related Works……………………………………………………………………...9

2.2.1 Sequential Pattern Mining Algorithm: PrefixSpan……………………………11

Chapter 3 Problem Definitions and Incision Strategy…………………………………14

3.1 Incision Strategy…………………………………………………………………15

Chapter 4 Projection Scheme…………………………………………………………21

4.1 Multi-Projection Technique………………………………………………………25

Chapter 5 Proposed Interval-based Event Mining Algorithm: CTMiner……………..…29

5.1 phase I: Incision and Projection…………………………………………………..30

5.2 phase II: Coincidence Mining……………………………………..…………33

5.3 phase III: Temporal Relation Discovery…………………………………………42

5.4 Handling large databases…………………………………………………………44

Chapter 6 Experimental Results……………………………………………………………45

6.1 Experiments on synthetic datasets………………………………………………..46

6.1.1 Runtime comparisons…………………………………………………..…46

6.1.2 Discussion of memory usage ……………………………..………………52

6.1.3 Scalability……………………………………………………………………53

6.2 Experiment on Real world dataset………………………………………………..…55

Chapter 7 Conclusion and Future works…………………………………………..…….58

Bibliography…………………………………………………………………………......60

List of Figures

Figure 2-1 Illustration for different representations of temporal pattern…….……………….....5

Figure 2-2 Two Ambiguity of representing temporal patterns in hierarchical representation……………………………………………………………6

Figure 3-1 All possible interval layouts between any consecutive time points.……………..18

Figure 3-2 Illustration for occurrence numbers in event sequence and its corresponding Csequence…………………………………………………….…………20

Figure 4-1 Illustration for merging event slice A^+ and A^- to form A and adjusting related events…………… …………………………….……….…………………23

Figure 4-2 (a) Original Csequence α, (b) α' = merge (A^+, α), (b) α" = merge (B^+, α)….25

Figure 4-3 Multi-projecting prefix ⟨(A^+)(B^+)(C)⟩ in (a) creates two postfix sequences (b) and (c) to obtain complete frequent coincidence patterns…….…………………...27

Figure 5-1 The *CTMiner* algorithm. …………………………….………………………..30

Figure 5-2 The *Incision_and_Projection* algorithm. ………….……………………………31

Figure 5-3 Illustration for *Incision_and_Projection* on event sequence 3 in Table 3-1. The *cur_time_list* and *last_time_list* point to the 6th and 5th *time_list*, respectively. …………………………………………………….…………33

Figure 5-4 The *CPrefixSpan* algorithm. ……………………….…………………………35

Figure 5-5 Illustrating three pruning strategies. ……………….……………………………36

Figure 5-6 Illustration for *elimination_test* on A^- successfully with all possible correlative event slices in the prefix. ……………………………...……………..38

Figure 5-7 Database of Csequences and projected databases w.r.t intact slice *B*………….....41

Figure 5-8 The *Temporal_Relation_Discovery* algorithm. ……………………….…………....43

Figure 5-9 Illustration for discovering temporal relations. (a) illustrates the Csequence and (b) illustrates the relations determined by comparing the pseudo time points………..……………………………………………………….………....44

Figure 6-1 Performance of the four algorithms on data set with *D*10k – *C*10 – *I*1.25 – *Ns* 500 – *Ni* 2,500 – *N*10k………………...…………………………………..47

Figure 6-2 The number of generated frequent pattern on dataset with *D*10k – *C*10 – *I*1.25 – *Ns* 500 – *Ni* 2,500 – *N*1k..…………………….…………………………….47

Figure 6-3 The distribution of frequent patterns of dataset with *D*10k – *C*20 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*500……………………………………………………….48

Figure 6-4 Performance of the four algorithms on data set with *D*100k – *C*10 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k…………………………………………….…….49

Figure 6-5 The number of generated frequent patterns on dataset with *D*100k – *C*20 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k…………….……..……………………………49

Figure 6-6 The pattern length distribution of frequent patterns on dataset with *D*100k – *C*20 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k..……………….……………………50

Figure 6-7 Performance of the four algorithms on data set with *D*200k – *C*10 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k………………………….………………………..51

Figure 6-8 The number of generated frequent pattern on dataset with *D*200k – *C*20 – *I*2.5– *Ns* 500 – *Ni* 2,500 – *N*10k……..……………………….………………...51

Figure 6-9 The pattern length distribution of frequent patterns on dataset with *D*200k – *C*20 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k…...…………….……………………52

Figure 6-10 Memory usage comparison of the four algorithms on data set with *D*100k – *C*10 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k……………………………………53

Figure 6-11 Scalability test of the CTMiner algorithm with different database size and minimum supports………………………………………………………54

Figure 6-12 The number of generated frequent patterns with different database sizes and minimum supports. ……………………………….……………………54

Figure 6-13 Experimental result of the CTMiner algorithm with varying minimum supports on real dataset. ……………………………………….……..…………56

Figure 6-14 The number of generated frequent patterns with varying minimum supports on real dataset……………………………….……………………...……………..……56

Figure 6-15 The pattern length distribution of pattern length of real dataset with varying minimum support……………………….……………………………………….57

List of Tables

Table 1-1 The Allen's 13 relations represent relations between any two event intervals. E.s and E,f refer to start point and finish point of event E, respectively....................3

Table 2-1 A sequence database……………………………………..……………………13

Table 2-2 Projected databases and sequential patterns…………….……………………13

Table 3-1 Event sequences in the temporal database and its corresponding coincidence sequences. ……………………………………….…………………………17

Table 3-2 Allen's temporal relations map to coincidence representations...…………………20

Table 5-1 Original database……………………………………………..………………40

Table 5-2 Projected databases and frequent temporal patterns…………..……………...…40

Table 6-1 Parameters of synthetic data generator………………..…………………...45

1896

Chapter 1

Introduction

Sequential pattern mining is an active research topic in data mining domain in last decade, due to its widespread applicability including the analyses of customer purchase behavior, Web access patterns, scientific experiments, disease treatments, natural disasters, DNA sequences, and so on. The sequential pattern mining was first proposed by Agrawal and Srikant [1] and many studies have contributed to the efficient mining of sequential patterns. GSP [2], PSP [3], SPADE [4], PrefixSpan [5], and MEMISP [6] have focused on discovering frequent temporal patterns from instantaneous events, that is, events are treated as time points without duration. For example, consider a medical database, in which a patient's treatment is regarded each time as a time point-based event, indicating the time of the treatment, such as "cough→headache→fever". However, in many applications events are not instantaneous; they instead occur over a time interval. Time point-based sequential patterns are inadequate to express the complex temporal relationships in domains such as medical, multimedia, meteorology and finance where the duration of events provides more specific and richer information. Interval-based pattern (also called temporal pattern) mining is proposed focusing on the domains with interval data.

Mining patterns from interval data is undoubtedly more complex and arduous. It requires a different approach from mining patterns from time point-based data, such as mining traditional sequential patterns or episodes. So far, very little attention has been paid to the issue of mining time interval-based sequential pattern mining. To the best of our knowledge, the related researches in interval-based event mining are based on Allen's temporal logics [7], 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 event intervals based on the arrangements of start points and finish points, as shown in Fig. 1-1. However, complex relationship among event intervals causes the difficulty while designing an efficient interval-based sequential pattern mining algorithm.

In this thesis, a new efficient algorithm called "**CTMiner**" (***C***oincidence ***T***emporal pattern ***Miner***) using the concept of the well-known sequential pattern mining algorithm PrefixSpan is proposed to discover temporal patterns from interval-based data without candidate generation. To address the problem of complex relationship among event intervals, a concept of coincidence-slice is developed. It focuses on *coincidences* among event intervals and incises the event intervals to disjoint smaller event slices according to *coincidences* then gathers the coincident event slices into the *coincidence*. Thus, the relationship among events is transformed to relationship among disjoint, smaller event slices and is simplified to "*before*", "*after*" and "*equal*". The event sequences are transformed to *Coincidence sequences* and thus facilitate the processing of interval-based pattern mining. Then the method called CPrefixSpan (Coincidence PrefixSpan) extends the concept of PrefixSpan to mine the frequent coincidence patterns. Due to the characteristic of coincidence-slice, the PrefixSpan is modified in order to cover all the frequent coincidence patterns. The multi-projection scheme is developed to obtain complete frequent coincidence patterns and three pruning strategies are also developed to improve the performance of our proposed algorithm. Finally, to comprehend a coincidence pattern, we use relation list to present all temporal relations in the coincidence pattern. Experimental studies on both synthetic and real datasets show that the proposed algorithm is efficient, scalable and outperforms the state-of-the-art algorithms.

Temporal Relation	Inversed Relation	Pictorial Example	time points comparision
A before *B*	*B* after *A*	A B	$A.f<B.s$
A overlaps *B*	*B* overlapped-by *A*	A B	$A.s<B.s, A.f<B.f, A.f>B.s$
A contains *B*	*B* during *A*	A B	$A.s<B.s, A.f>B.f$
A starts *B*	*B* started-by *A*	A B	$A.s=B.s, A.f<B.f$
A finished-by *B*	*B* finishes *A*	A B	$A.s<B.s, A.f=B.f$
A meets *B*	*B* met-by *A*	A B	$A.f=B.s$
A equal *B*	*B* equal *A*	A B	$A.s=B.s, A.f=B.e$
A after *B*	*B* before *A*	B A	$B.f<A.s$
A overlapped-by *B*	*B* overlaps *A*	B A	$B.s<A.s, B.f<A.f, B.f>A.s$
A during *B*	*B* contains *A*	B A	$B.s<A.s, B.f>A.f$
A started-by *B*	*B* starts *A*	B A	$B.s=A.s, B.f<A.f$
A finishes *B*	*B* finished-by *A*	B A	$B.s<A.s, A.f=B.f$
A met-by *B*	*B* meets *A*	B A	$B.f=A.s$

Table 1-1 The Allen's 13 relations represent relations between any two event intervals. *E.s* and *E,f* refer to start point and finish point of event E, respectively.

The rest of the thesis is organized as follows. Chapter 2 gives the motivation and related work. Chapter 3 provides the details of problem Definitions and the incision strategy. Chapter 4 describes the similarities and dissimilarities of projection scheme of PrefixSpan and CTMiner. Chapter 5 illustrates the CTMiner algorithm. Chapter 6 gives the experimental results and we conclude in Chapter 7. Note that, an event interval, i.e., interval-based event, in the rest of thesis is denoted as an event and instantaneous event is denoted as time point-based event. The start time point and finish time point of an event interval is denoted stp and ftp, respectively.

Chapter 2

Motivation and Related Works

2.1 Motivation

2.1.1 Representations of a Temporal Pattern

Allen proposed 13 temporal relations between any two events without ambiguity. However, the representation of temporal patterns based on Allen's logics will bear some problems as follows.

- **Various proposed representations suffer from different kinds of drawback.**

Hierarchical representation [8] describes relationships among more than three events the which is a compact but lossy encoding method. There are two kinds of ambiguity problems in representation of a temporal pattern. First, the same relationship among events can be mapped to different temporal patterns. An example is shown in Fig. 2-1(a), in which a pattern can be expressed as "*(((A overlap B) before C) contain D)*" or "*((A overlap B) before (C contain D))*." Second, a temporal pattern can be represented as different relations among events. For example, Fig. 2-1(b) shows that the pattern "*((A overlap C) Overlap B))*" can be represented in two different relations among events.

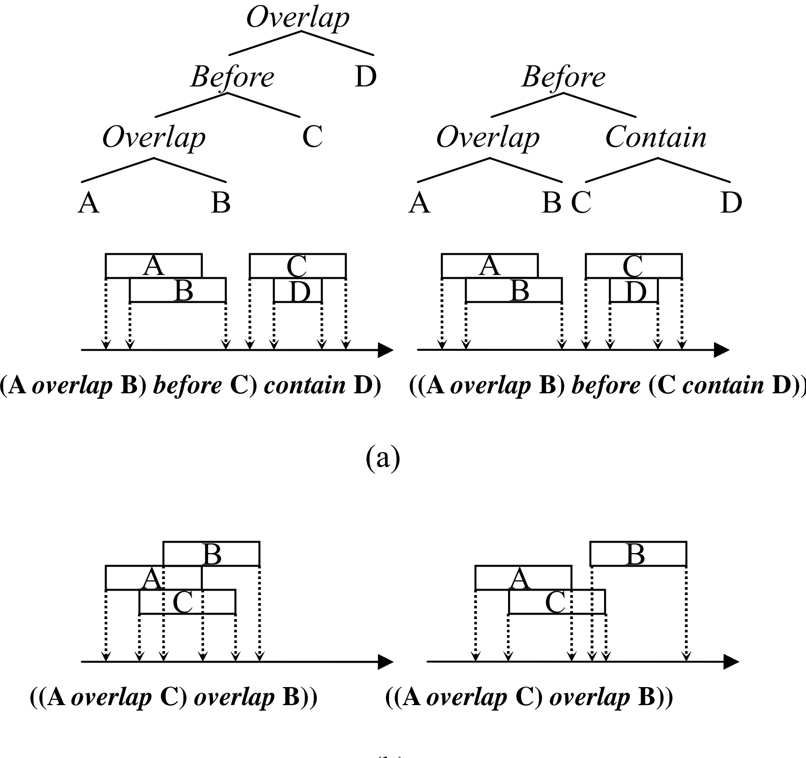


Figure 2-1 Two Ambiguity of representing temporal patterns in hierarchical representation.

Relation matrix [9] and **Relation list** [10] precisely list all binary relationships among events in a pattern, and examples are shown in Fig. 2-2(a) and Fig. 2-2(b), respectively. These unambiguous representations exhaustively list all ($k \times (k-1)/2$) pairwise relations in a k-events pattern, but it suffers from the problem of scalability in long temporal patterns.

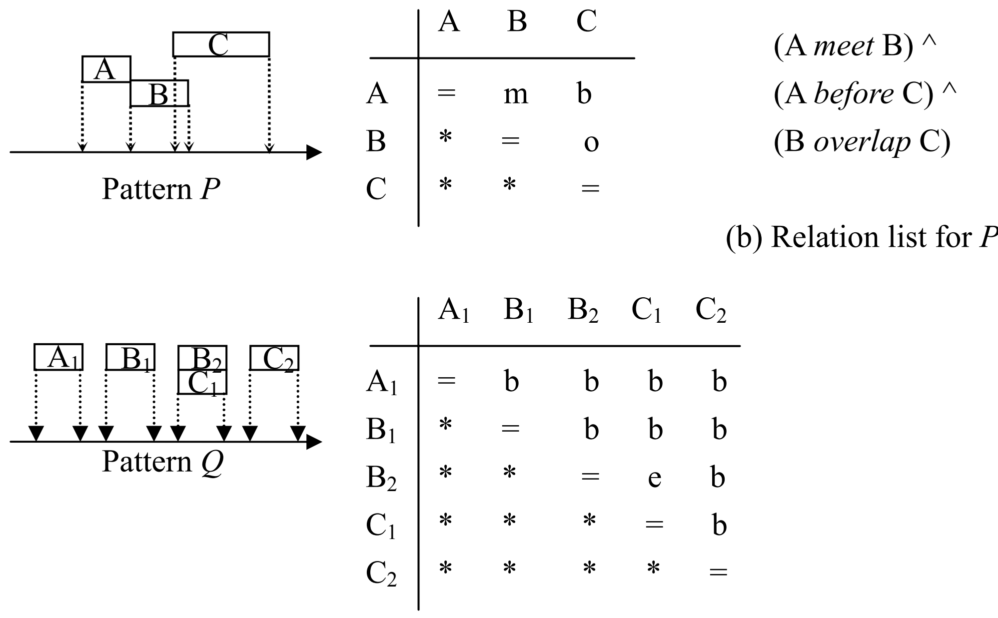


Figure 2-2 Illustration for different representations of temporal pattern.

Temporal representation [11] utilizes time points arrangement to represent a temporal pattern and sequence. For example, the pattern *P* shown in Fig. 2-2 can be represented in the unique expression "$(A^{+}<A^{-}<B^{+}<C^{+}<B^{-}<C^{-})$", where " $^{+}$ " or " $^{-}$ " attached to an event indicates either a start time point or finish time point to the event, respectively. Due to the unique one-one mapping, it describes temporal pattern and sequence unambiguously.

TKSR (Time series Knowledge Representation) [12] expresses the temporal pattern to the temporal concepts of coincidence with partial order. It describes a sequence of disjoint overlapped groups among events in order of time. In the same example as shown in Fig. 2-2, the pattern *P* can be represented as the expression "(A)(B)(BC)(C)" for coincidence order. The TKSR representation of pattern *P* and *Q* are the same because that TKSR does not specify overlap of an event to the other event is entire or part. Obviously, TKSR is not easily comprehensible and suffers the problem of ambiguity.

Augmented hierarchical representation [13] based on the hierarchical representation solves ambiguity by attaching additional counting information to each hierarchy in

chronological order. It has been proven that utilizing additional 5 counters i.e., *contain, finish-by, meet, overlap*, *start* denoted as [*c*, *f*, *m*, *o*, *s*] to accumulate those 5 temporal relations between the event *e* and all events occurring before *e* is sufficient to achieve an unambiguous representation. Take the pattern *P* in Fig. 2-2 as an example. First, we have "A meet B" so we set *meet* counter to 1 and it is represented as (A *meet* [00100] B). Then "B *overlap* C" is attached to the expression and increments the length of the expression then accumulates all 5 relations hence we set *overlap* and *meet* counter to 1. The pattern *P* can be expressed as "((A *meet* [00100] B) *overlap* [00110] C)". The expression is not easily comprehensible and wastes $(k-1)\times 6$ memory space in a k-events pattern.

2.1.2 The Problems of Complex relationship on Different Temporal Pattern Mining Approaches

- **The relationships between two events in the temporal patterns are substantially complex.**

The relationship between any two time point-based events only indicates "*before*", "*after*" and "*equal*" but there are Allen's 13 temporal logics to represent the relationship among interval-based events due to the characteristic of time duration. Although the Allen's 13 temporal logics can be normalized to 7 as the first 7 temporal relations shown in Fig. 1-1, i.e., "*before*", "*overlap*", "*contain*", "*start*", "*finished-by*", "*meet*" and "*equal*", by following the order of start time, finish time and event type. However, normalized Allen's 7 temporal logics remain complex and cause the problem while applying it to different interval-based event mining approaches.

Generation-and-test approach [8, 9, 10, 12, 13]**:** It usually requires multiple iterations to find all sequential patterns. In each iteration, some candidate patterns were generated and testified the frequency. Thus, reducing the number of candidate patterns is the main bottleneck

and challenge. The complicated relationships in interval-based event sequential pattern mining will lead to the generation of huge number of candidate patterns and bear tedious workload of support counting for candidate patterns.

Frequent Pattern-growth based approach [11, 14] is an efficient mining approach which applies to time point-based patterns mining without candidate generation. It recursively partitions the time point-based event sequence database into smaller projected databases and grow the sequential patterns by exploring frequent time point-based events in associated projected database. The sequential pattern grows without candidate generation due to monotonic relationship between frequent 1-events and the corresponding prefix, i.e., only "*before*" and "*equal*". A frequent 1-event can append directly to the prefix. To apply the approach to interval-based event pattern mining still requires candidate generation because of complicated relationship among events.

Based on the above observation, complex relationship is really a critical issue which causes prohibitively cost on time and space in mining processing. If the complicated relationship among events can be reduced in temporal pattern, then the efficient and effect of interval-based event mining algorithm will be improved substantially.

- **Related research themes of temporal pattern mining**

A lot of extended researches [24, 25, 26, 27] of sequential pattern mining are very important and necessary, such as closed pattern mining, maximal patterns mining and incremental mining, sequential patterns classification, to name a few. Those researches developed based on time point-based event may not be suitable for interval-based event since the complex relationships among event intervals may degrade performance dramatically. To the best of our knowledge, there have been a few related researches about the extension of temporal pattern mining. Addressing the issue of complex relationship among events in temporal patterns provides us the opportunity for designing efficient related extensions of temporal pattern mining.

According to our observation, the different extent of overlap causes the complexity among events. Besides relation of "*before*" and "*meet*", other five relations illustrate different extent of overlap. For instance, the "*overlap*" relation shows the overlapping on the front part and rear part of two events. Different correlated positions of an event which is fully overlapped by another event forms different relations, i.e., "*start*", "*contain*", "*finished-by*" or "*equal*". Therefore, we classify different parts of event as event slices into overlap or non-overlap. Then, the relationship between event slices is very simple, i.e., "*before*", "*equal*". In general, event sequence also exhibits many overlaps among events. Therefore, in this paper, the concept of coincidence-slice and incision strategy is proposed which transform events of event sequence to event slices. The complex relationship among events is transformed to simpler relationship among event slices. The proposed concept of coincidence-slice and incision strategy is described in details in Chapter 3.

2.2 Related Works

Some recent researches have investigated the mining of sequential patterns with interval-based events [8, 9, 10, 11, 12, 13, 14]. Kam et al. [8] designed an Apriori-based algorithm that uses the hierarchical representation to discover frequent temporal patterns. The representation only keeps (*k-1*) relations and two time points of earliest and latest time points in a k-pattern. Therefore, the hierarchical representation is ambiguous and many spurious patterns are generated. Hoppner [9] also proposed an Apriori-based algorithm that counts support of all candidate patterns of length k by scanning database once with a sliding window. It also defined the supporting level of a pattern as the total time in which the pattern can be observed in a sliding window to improve the performance of the algorithm. However, a major concern for this approach is how to decide the proper size of the sliding window, since the

sliding window directly dominates the mining results and efficiency of the algorithm. It also needs to scan database repeatedly. Mochen [12] proposed a new representation, called TKSR, which uses the coincidence concept to facilitate the process of temporal patterns discovery. An event sequence can be treated as a series of disjoint overlaps named "coincidences". It treats coincidence and event as itemset and item. Then it adopted CHARM [15] to find frequent itemsets as marginal-closed coincidences then applied the CloSpan [16] to mine the closed sequential patterns as partial ordering marginal-closed coincidences. The pattern represented with TKSR is ambiguous and is not easily comprehensible. Edi et al. [14] developed a pattern-growth algorithm, named ARMADA based on an efficient sequential patter mining algorithm MEMISP to find frequent temporal patterns and also reduces the memory space of projected databases. However, it still uses lots of memory space because temporal patterns are expressed by relation matrix. It requires only two database scans but it also generates a lot of candidates and accesses memory frequently due to complex relationship among events. Papapetrou et al. [10] proposed the Hybrid-DFS algorithm which is Apriori-based approach to mine temporal arrangements of temporal intervals and a relation list to express temporal patterns. It also proposed an enumeration tree structure to improve the performance of the algorithm. First, it scans database twice to obtain frequent 1-patterns and all related records of frequent 2-patterns as first and second levels of the tree respectively by BFS traverse order. To obtain the frequent k-patterns in the level k of the tree, it firstly merges frequent (k-1)-patterns and frequent 1-patterns in level (k-1) and 1, respectively. Because the k-patterns can be treated as the combination of frequent 2-patterns, so it scans related records in level 2 to verify the frequency of a specific k-pattern by DFS traverse order. It transforms an event sequence into a vertical representation using id-lists. The id-list of an event is merged with the id-list of other events to generate temporal patterns. This approach does not scale well when the length of temporal pattern increases. Wu et al. [11] derived a pattern-growth based algorithm called TprefixSpan for mining temporal pattern from interval-based events

and represents temporal patterns in compact but ambiguous temporal representation. It first discovers single frequent events from the projected database. Next, all the possible candidates could be generated while appending a discovered frequent 1-event to the prefix and all the possible relations between them need to be considered. Last, it scans the projected database again for support counting. TPrefixSpan still needs to scan the projected database multiple times and it does not employ any pruning strategy to reduce the search space. Patrel et al. [13] proposed an Apriori-based algorithm named IEMiner. It utilizes the additional counting information to achieve lossless hierarchical representation called augmented representation and generates frequent temporal patterns iteratively and increases the pattern length by one after iteration. It also proposed a support counting method to scan database once to derive the frequency of all candidate patterns in each iteration. When scans the database, many temporal patterns are generated by composing the events in database. If a generated pattern is a candidate pattern then we accumulate the support of the candidate pattern. After scanning, the frequency of candidate patterns is determined. The operation is very costly due to the complexity of augmented representation.

2.2.1 Sequential Pattern Mining Algorithm: PrefixSpan

We use the concept of the well-known pattern-growth based sequential pattern mining algorithm called PrefixSpan (i.e., **Prefix**-projected **S**equential **pa**tter**n** mining), which explores prefix-projection in sequential pattern mining. For the sequential database *S* in Table 2-1 with minimum support = 2, sequential patterns in *S* can be mined by a prefix-projection method in the following steps.

Step 1: Find length-1 sequential patterns. Scan *S* once to find all frequent items in sequences. Each of these frequent items is a length-1 sequential pattern. They are ⟨*A*⟩: 4, ⟨*B*⟩: 4, ⟨*C*⟩: 4, ⟨*D*⟩: 3, ⟨*E*⟩: 3 and ⟨*F*⟩: 3, where ⟨pattern⟩: count indicates the pattern and its

associated support count.

Step 2: Append each frequent 1-item to the prefix and create projected database with respect to the appended prefix. For the first time to append each frequent 1-item to the prefix, we directly set each frequent 1-item as appended prefix. And the complete frequent sequential patterns can be partitioned into $|L1|$ parts, where $|L1|$ is the number of frequent 1-items. Therefore, we have six projected databases with respect to the appended prefix, i.e., $\langle A \rangle$, $\langle B \rangle$,…, and $\langle F \rangle$. The projected databases with respect to prefixes $\langle A \rangle$ and $\langle B \rangle$ are shown in the 2nd row of Table 2-2, respectively.

Step 3: Recursively go back to step 1 to find whole frequent sequential patterns. For the running example above, the frequent length-1 item of projected database with respect to prefix $\langle A \rangle$ are $\langle B \rangle$: 4, $\langle _B \rangle$: 2, $\langle C \rangle$: 4, $\langle D \rangle$: 2, $\langle F \rangle$: 2, where $\langle _B \rangle$ indicates item B occurs simultaneously with the items of last itemset in the prefix. After appending each frequent length-1 item to the prefix, the projected database of appending $\langle _B \rangle$ to the prefix $\langle A \rangle$ to form a new prefix $\langle (AB) \rangle$ is shown in 3rd row of Table 2-2. Similarly, the complete frequent sequential patterns begin with item A are generated and are shown in the last row of Table 2-2 if we keep going on the process.

Sequence_id	Sequence
10	⟨A(ABC)(AC)D(CF)⟩
20	⟨(AD)C(BC)(AE)⟩
30	⟨(EF)(AB)(DF)CB⟩
40	⟨ EG(AF)CBC ⟩

Table 2-1 A sequence database

Prefix	Projected (postfix) database	Prefix	Projected (postfix) database
⟨A⟩	10: ⟨(ABC)(AC)D(CF)⟩ 20: (_D)C(BC)(AE) 30: ⟨(_B)(DF)CB ⟩ 40: ⟨(_F)CBC ⟩	⟨B⟩	10: ⟨(_C)(AC)D(CF)⟩ 20: ⟨(_C)(AE)⟩ 30: ⟨(_B)(DF)CB ⟩ 40: ⟨C⟩
⟨(AB)⟩	10: ⟨(_C)(AC)D(CF)⟩ 30: ⟨(DF)CB⟩	⟨(BC)⟩	10: ⟨(AC)D(CF)⟩ 20: ⟨(AE)⟩
⟨(AB)D⟩	10: ⟨(CF)⟩ 30: ⟨(_F)CB⟩	⟨(BC)A⟩	10: ⟨(_C)D(CF)⟩ 20: ⟨(_E)⟩
⟨(AB)DC⟩	10: ⟨(F)⟩ 30: ⟨B⟩		
Frequent sequential patterns begin with event A		Frequent sequential patterns begin with event B	
⟨A⟩,⟨AA⟩,⟨AB⟩,⟨A(BC)⟩,⟨A(BC)A⟩,⟨ABA⟩,⟨ABC⟩, ⟨(AB)⟩,⟨(AB)C⟩,⟨(AB)D⟩,⟨(AB)F⟩,⟨(AB)DC⟩,⟨AC⟩, ⟨ACA⟩,⟨ACB⟩,⟨ACC⟩,⟨AD⟩,⟨ADC⟩,⟨AF⟩		⟨B⟩,⟨BA⟩,⟨BC⟩,⟨(BC)⟩,⟨(BC)A⟩,⟨BD⟩, ⟨BDC⟩,⟨BF⟩	

Table 2-2 Projected databases and sequential patterns

Chapter 3

Problem Definitions and Incision Strategy

We focused on the discussion of temporal pattern mining due to the widespread applicability and lacking of researches. In this chapter, we define the problem of temporal pattern mining and introduce the incision strategy. The interval-based mining problem is much more arduous than traditional time point-based mining problem. Since relationship among events are more complicated than that of the time point-based events. The complex relation between two events is the major bottleneck for mining temporal pattern. Therefore, the incision strategy is proposed to transform event sequence to Coincidence sequence which addresses the critical issue of temporal pattern mining.

Definition 1 (Event interval) Let $E = \{e_1, e_2, \ldots, e_k\}$ be a set of all event types. Without loss of generality, we define a set of uniformly spaced time points based on the real number R. We say the triplet $(e_i, s_i, f_i) \in E \times R \times R$ is an event interval or temporal interval, where $e_i \in E$, $s_i, f_i \in R$ and $s_i < f_i$. The two time points s_i and f_i are called *start time point* and *finish time point* and denoted as stp and ftp, respectively. The set of all event intervals over E is denoted by I. We write $(e, s, f) \subseteq (e', s', f')$ if $s \leq s', f' \leq f$.

Definition 2 (Event sequence and maximal property) An event sequence $e_i \in$ E is a series of event interval triplets $\langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_m, s_n, f_n)\rangle$, where $m \leq n$ since an event may occurs many times, $s_i \leq s_{i+1}$, and $s_i < f_i \ \ \forall i$. Every interval (e_i, s_i, f_i) must be maximal in sequence, i.e., there is no (e_i, s_j, f_j) in the sequence such that neither s_j nor f_j occurs in the interval $[s_i, f_i]$. We call this assumption, maximal property, defined as follows:

$$\forall\ (e_p, s_i, f_i), (e_q, s_j, f_j) \in I, (e_p, s_i, f_i) \neq (e_q, s_j, f_j): s_i \leq s_j \wedge f_i \geq s_j \wedge e_p \neq e_q \qquad -(1)$$

Equation (1) above is also called the maximality assumption [15]. 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 3 (Temporal database) Considering a database $D = \{r_1, r_2, \ldots, r_m\}$, each record r_i, where $1 \leq i \leq m$, consists of a sequence-id and an event interval (e_i, s_i, f_i). Given a sequence set $\{q_1, q_2, \ldots, q_n\}$, each sequence q_j is an event sequence with grouping the records in D with same sequence-id. $D = \{q_1, q_2, \ldots, q_n\}$ is called a temporal database.

Actually, all events can be grouped together by the same sequence-id and arranged by non-decreasing order of start time, end time and event type into an event sequence. As the result, the database D can be viewed as a collection of event sequences. For example, in Table 3-1, the temporal database consists of 17 events, and 4 event sequences. We use normalized Allen's 7 interval logics to describe the temporal relation between every two events in a sequence.

Definition 4 (Temporal pattern) Given n events (e_i, s_i, f_i), $1 \leq i \leq n$, a temporal pattern of size $n > 1$ is defined as a matrix $M \in A^{n \times n}$ where A is a set of Allen's temporal relations and each index i maps to the corresponding event e_i, the element $M[i, j]$ denotes the relationship between two event intervals (e_i, s_i, f_i) and (e_j, s_j, f_j). The number of intervals in the temporal pattern P is called the dimension of P, denoted as $\dim(P)$. If $\dim(P) = k$, then P is called a k-pattern.

Various representations have been proposed for temporal pattern, as we mentioned above. We adopt relation list to represent temporal pattern since it can precisely and unambiguously list all pairwise relationships among events in a pattern.

3.1 Incision Strategy

The coincidence-slice architecture is implemented by incision strategy. The proposed strategy cuts events into disjoint smaller event slices based on the global information of event sequence, i.e., time points of events, and the simultaneous event slices are collected into a

group called coincidence. We define event slice and coincidence as follows.

Definition 5 (Coincidence, Event time set and Event slice) 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 $T = \{t_1, t_2 \ldots t_{m-1}, t_m\}$,where $t_i \in \{s_1, f_1, s_2, f_2, \ldots, s_n, f_n\}$, $t_i \neq t_{i+1}$, *and* $t_i < t_{i+1}$ for $1 \leq i < m$ without repetition, is called an *event time set* which collects all time points of event intervals with no repeat and in increasing order of time in q. A coincidence of q is a time period denoted as $c_i = (t_i, t_{i+1})$ where $t_i, t_{i+1} \in T$ and $1 \leq i < m$. Therefore, we have (m-1) successive coincidences $c_1, c_2, \ldots, c_{m-1}$ in q. Furthermore, four types of event slices are defined as follows.

1. **Start slice**: A *start slice* of event e_i in q is defined as an interval which belongs to coincidence c_i and denoted as $\boldsymbol{e_i}^+$ if and only if (1) $t_i = s_i$ and (2) $t_{i+1} < f_i$.

2. **Finish slice**: A *finish slice* of event e_i in q is defined as an interval which belongs to coincidence c_i and denoted as $\boldsymbol{e_i}^-$ if and only (1) $t_{i+1} = f_i$ and (2) $t_i > s_i$.

3. **Intermediate slice**: An *intermediate slice* of event e_i in q is defined as an interval which belongs to coincidence c_i and denoted as $\boldsymbol{e_i}^*$ if and only if (1) $t_i > s_i$ and (2) $t_{i+1} < f_i$.

4. **Intact slice**: An *intact slice* of event e_i in q is defined as an interval which belongs to coincidence c_i and denoted as $\boldsymbol{e_i}$ if and only if (1) $t_i = s_i$, (2) $t_{i+1} = f_i$.

For example, in Table 3-1, sequence 2 has 4 events: (B, 1, 5), (D, 8, 14), (E, 10, 13), (F, 10, 13) and its corresponding *event time set* = {1, 5, 8, 10, 13, 14}. There are five successive coincidences c_1=(1,5), c_2=(5,8), c_3=(8,10), c_4(10,13) and c_5=(13,14). Hence the event D have three event slices: 1. start slice $D^+ = (D, c_3)$, 2. intermediate slice $D^* = (D, c_4)$ and 3. finish slice $D^- = (D, c_5)$. The only event slice of event B is $B = (B, c_1)$. Obviously, an event can only have a pair of start slice and finish slice or one intact slice but it could have many intermediate slices. Actually, intermediate slices do not help in the mining processing due to a pair of start slice and finish slice or an intact slice is sufficient to imply the relative time positions of an event in an event sequence.

ID	Event symbol	Start time	Finish time	Relative Positions	Coincidence sequence
1	A	2	7		$\langle(A^{+})(B^{+}C^{+}A^{-})(B^{-})(C^{-})(D^{+})(E)(D^{-})\rangle$
1	B	5	10		
1	C	5	12		
1	D	16	22		
1	E	18	20		
2	B	1	5		$\langle(B)(D^{+})(EF)(D^{-})\rangle$
2	D	8	14		
2	E	10	13		
2	F	10	13		
3	A	6	12		$\langle(A^{+})(B^{+}A^{-})(B^{-})(@D^{+})(E)(D^{-})\rangle$
3	B	7	14		
3	D	14	20		
3	E	17	19		
4	B	8	16		$\langle(B)(A)(D^{+})(E)(D^{-})\rangle$
4	A	18	21		
4	D	24	27		
4	E	25	28		

Table 3-1 Event sequences in the temporal database and its corresponding coincidence sequences.

We obviously perceive that both event interval and event slice are composed of two time points of events in an event sequence. There are 4 kinds of event slices in two consecutive time points in an event sequence, as shown in Fig. 3-1. We only consider the period time between t_i and t_{i+1}, i.e., the coincidence c_i. The event intervals finished at t_i or started at t_{i+1} are not processed since we handle them in $c_{i-1} = (t_{i-1}, t_i)$ and $c_{i+1} = (t_{i+1}, t_{i+2})$, respectively. For instance, in Fig. 3-1, events E and F are processed in c_{i-1} and c_{i+1}, respectively. A start slice A^+ in c_i indicates that event A starts at t_i and finishes after at t_{i+1} and a finish slice B^- in c_i indicates that event B starts before t_i and finishes at t_{i+1}. An intermediate slice C^* in c_i means that event C occurs across c_i and an intact slice D in c_i means that the duration of event D equals c_i. The relationship between any two event slices can only be

"*before*", "*after*" and "*equal*". The relation between two event slices in the same period is "*equal*" and the relation between event slices in different periods is either "*before*" or "*after*".

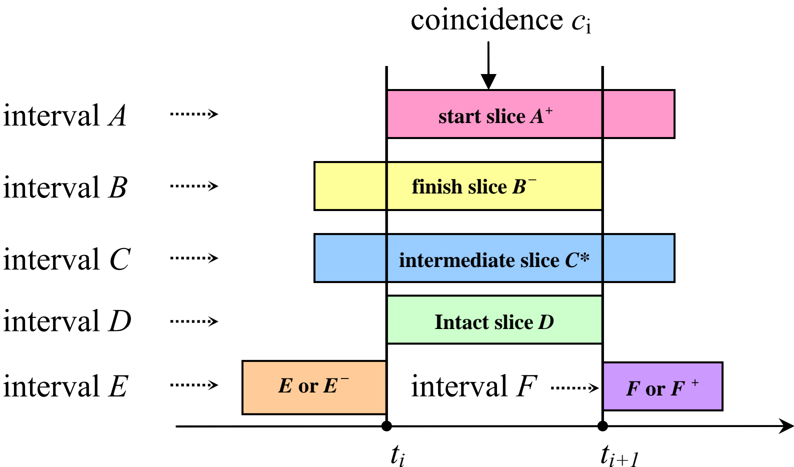


Figure 3-1 All possible interval layouts between two consecutive end time points.

Definition 6 (Coincidence sequence with respect to an event sequence) A *Coincidence sequence* denoted as **Csequence** consists of an ordered set of event slices and *meet* tokens represent an event sequence unambiguously. Given an event sequence $q = \langle(e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_n, s_n, f_n)\rangle$, an *event time set* $\{t_1, t_2 \ldots t_{m-1}, t_m\}$ of q is created. Totally $(m-1)$ continuous *coincidences* $c_1, c_2, \ldots, c_i, \ldots, c_{m-1}$ are created. The event sequence q is transformed to Csequence by the incision strategy. The incision strategy transforms an event sequence q to coincidence sequence by the following operations.

1. Transforms event intervals to event slices
   - An event occurs exactly at coincidence c_i then the corresponding *intact slice* is created and is put into c_i.
   - An event occurs from t_i to t_j, where $t_i < t_j$ and t_i is the start time of c_p and t_j is the finish time of c_q, then its corresponding *start slice* and *finish slice* are created and are put into c_p and c_q, respectively.
2. Place meet token to distinguish adjacent event intervals
   - If there are two events (e_i, s_i, f_i) and (e_i, s_j, f_j) meets at time t, i.e., $t = f_i = s_j$, and t is the finish time of c_p and the start time of c_{p+1} then a meet token "@" is put into c_{p+1}.

3. Finally, empty coincidences are removed and the remaining ordered coincidences form the Csequence.

Note that, event slices in coincidence are ordered by (1.) Intact slice (2.) Start slice (3.) Finish slice and event slices with the same type of event slice are ordered by event type.

We decide to remove all empty coincidences to save memory space. For instance, consider the sequence 4 in Table 3-1, There are two empty coincidences in the sequence which is represented as coincidence representation, i.e., $\langle(B)(\ )(A)(\ )(D^{+})(E)(D^{-})\rangle$. Although the incision strategy looks promising, actually it has a fatal defect. We cannot distinguish between two adjacent intervals and two separate intervals due to empty coincidence elimination. A meet token "@" is used to address this drawback. For example, there are two event sequences "A *before* B" and "A *meet* B" with the same coincidence representation $\langle(A)(B)\rangle$ by applying the incision strategy. A meet token "@" is placed in the coincidence c_{i+1} of Csequence of " A *meet* B" where event A finished at c_i which met event B started at c_{i+1}, denote as $\langle(A)(@B)\rangle$.

The Allen's temporal logics between any two events can be mapped to Csequence representation without ambiguity as shown in Table 3-2. To represent temporal relations among more than two events in an event sequence is still unambiguous since it maintains the relative positions of time points of events and additional meet tokens distinguish two adjacent events .

Temporal Relation	Pictorial Example	Coincidnece Sequence Representation
A before *B*	A B	$\langle(A)(B)\rangle$
A overlaps *B*	A B	$\langle(A^{+})(B^{+}A^{-})(A^{-})\rangle$
A contains *B*	A B	$\langle(A^{+})(B)(A^{-})\rangle$
A starts *B*	A B	$\langle(AB^{+})(B^{-})\rangle$
A finished-by *B*	A B	$\langle(A^{+})(BA^{-})\rangle$
A meets *B*	A B	$\langle(A)(@B)\rangle$
A equal *B*	A B	$\langle(AB)\rangle$

Table 3-2 Allen's temporal relations map to coincidence representations.

The *occurrence number* is attached to every event of an event sequence to distinguish multiple occurrences of the same event type. In the example shown in Fig. 3-2(a), both event A and B occur twice in the event sequence. After transforming the event sequence to the Csequence, the occurrence numbers are also attached to the corresponding event slices as shown in Fig. 3-2(b). Note that, an event incised to a pair of start slice and finish slice has the same occurrence number. The first occurrence of event A and B incised to a pair of start slice and finish slice with the same occurrence number of 1.

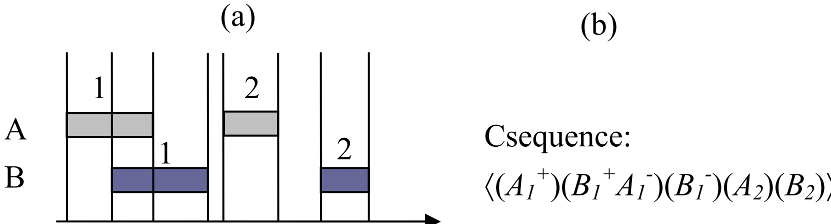


Figure 3-2 Illustration for occurrence numbers in event sequence and its corresponding Csequence.

Chapter 4

Projection Scheme

The projection process is a significant contribution of PrefixSpan algorithm, since it partitions both the data and the sets of frequent patterns to be tested, and confines each test to its 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 local frequent items, and then append them to the prefix to form a new frequent pattern.

We extend the projection process of PrefixSpan and do some modification to adapt to coincidence representation. The major concern is to represent each subsequence of an event sequence by coincidence representation correctly. Given an event sequence and its corresponding Csequence, a subsequence of an event Csequence can be treated as choosing some pairs of start slice and its corresponding finish slice and some intact slices to stay in the Csequence and eliminating the rest of event slices. An event is represented as a pair of start slice and finish slice in the Csequence because the event has different extent overlaps with the other k events where $k \geq 1$. If the k events are eliminated, the event must be represented as an intact slice instead of a pair of start and finish slices in the Csequence. For example, a sequence of event A obviously is a subsequence of the event sequence "A *contains* B". Event A represented in Csequence as $\langle(A)\rangle$ is not a subsequence of $\langle(A^+)(B)(A^-)\rangle$, i.e., $\langle(A)\rangle \neq \langle(A^+)(A^-)\rangle$. Therefore, the *merge* operation is proposed to transform the Csequence into a new Csequence which forms a coincidence representation of subsequence of corresponding event sequence correctly. The *merge* operation is defined as follows.

Definition 7 (*merge*(α,e^+) *operation*) The *merge* operation merges the start slice e^+ and its corresponding finish slice e^- to form an intact slice e and adjusts related event slices in

Csequence α. Then, a new Csequence α' is generated which is also a subsequence of the Csequence α. Essentially, α' is a coincidence representation of subsequence of corresponding event sequence with respect to α. The intact slice *e* is recovered by merging a series of coincidences started with the start slice e^+ and finished by the finish slice e^- and event slices correlated to the new coincidence need to be adjusted.

Fig. 4-1 shows all possible events related with event A which is incised to a pair of event slices, i.e., A^+ and A^-. The event sequence shown in Fig. 4-1(a) contains event A and all its related events B, C, D to M. The coincidence representation of the event sequence is

$$\langle (I)(JK^+L^+M^+)(@A^+B^+C^+\ K^-)(@DE^+F^+)(A^-B^-E^-L^-)(@GC^-F^-M^-)(H) \rangle$$

and we want to merge start slice A^+ and finish slice A^- to an intact slice A by merging a series of coincidences started with A^+ and finished by A^-. Three successive coincidences, c_i, c_{i+1} and c_{i+2}, shown in Fig. 4-1(a) are merged to form a new one as shown in Fig. 4-1(b).

Adjustments of 12 events related to event A, i.e., events B, C, D to M, are classified into 3 categories. First, events occur before or after the new coincidence, i.e. events I, J, G and H, do not need any adjustment. Second, events not fully coincident with the new coincidence, i.e. events D, E, F and K, will be omitted. Third, event slices coincident fully with the new coincidence, i.e. events B, C, L and M, remain in the new coincidence. If both start slice and finish slice are of the same event type in the new coincidence c_i, we merge the event slices to become an intact slice. After *merge* operation on event slices A^+ and A^-, the new Csequence is formed and is represented as $\langle (I)(JL^+M^+)(@ABC^+\ L^-)(@GC^-M^-)(H) \rangle$ as shown in Fig 4-1(b).

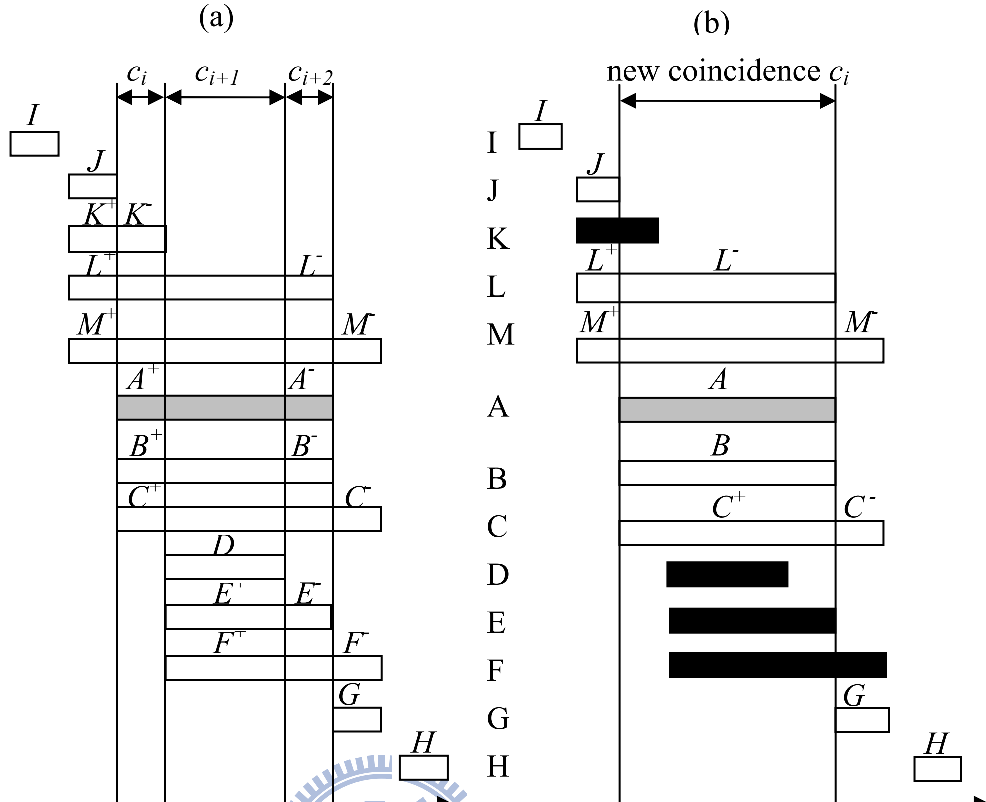


Figure 4-1 Illustration for merging event slice A^+ and A^- to form A and adjusting related events.

Definition 8 (CSubsequence) The subsequence β without single start slice or single finish slice of Csequence α, i.e., either a pair of start slice and its corresponding finish slice or an intact slice of α appears in the subsequence, is called a Csubsequence of α. Furthermore, the Csequence β' is generated after multiple *merge*(e,β) operations where e is a start slice in β, and the β' is a Csubsequence of α too.

Definition 9 (support count of a Csequence) Given a coincidence database D_c, a tuple $\langle sid, s\rangle$ is said to contain a Csequence β, if β is a subsequence of s. The support of a Csequence β in a coincidence database D_c is the number of tuples in the database containing β, i.e.,

$$support\ (\beta) = |\ \{\ \langle sid, s\rangle \mid (\ \langle sid, s\rangle \in D_c) \wedge (\beta \quad s)\ \}\ |$$

Given a positive integer *min_sup* as the support threshold, a Csequence β is called a coincidence pattern if $support\ (\beta) \geq min_sup$.

Definition 10 (prefix, projection and postfix)

1. **prefix**: Given a Csequence $\alpha = \langle e_1, e_2, \ldots e_n \rangle$, a Csequence $\beta = \langle e'_1, e'_2, \ldots e'_m \rangle$ (m ≤ n) is called a **prefix** of α if and only if (1) β is a Csubsequence of α with least *merge* operations, that is, there exists a Csequence $\alpha' = \langle e''_1, e''_2, \ldots e''_q \rangle$ (m ≤ q) which is the largest super-sequence of β after least *merge* operations of α. (2) $e_i' = e''_i$, for i ≤ m-1 (3) $e'_m \subseteq e''_m$, and (4) all the event slices in $(e''_m - e'_m)$ are orderly after those in e'_m.

2. **projection**: Given Csequences α and β such that β is a Csubsequence α and α' is the largest super-sequence of β with respect to α. A subsequence δ of Csequence α' (i.e., $\delta \subseteq \alpha'$) is called a **projection** of α with respect to prefix β if and only if (1) δ has prefix β and (2) there exist no proper super-subsequence δ' of δ, $\delta' \neq \delta$, such that δ' is a subsequence of α' and also has prefix β.

3. **postfix**: Let $\delta = \langle e_1 e_2 \ldots e_n \rangle$ be the projection of α with respect to prefix $\beta = \langle e_1 e_2 \ldots e_{m-1} e'_m \rangle$ (m ≤ n) and α' is the largest super-sequence of β with respect to α. A CSequence $\gamma = \langle e''_m e_{m+1} \ldots e_n \rangle$ is called the **postfix** of α with respect to prefix β and α', denoted as $\gamma = \alpha' / \beta$, where $e''_m = (e_m - e'_m)$. We also denote $\alpha' = \beta \cdot \gamma$. If β is not a subsequence of α', both projection and postfix of α with respect to β and α' are empty.

An example as shown in Fig. 4-2, $\langle (A^+) \rangle$, $\langle (A^+)(B^+) \rangle$, $\langle (A^+)(B^+)(C) \rangle$, $\langle (A^+)(B^+)(C)(A^-) \rangle$, $\langle (A^+)(B^+)(C)(A^-B^-) \rangle$ are the prefixes of sequence $\alpha = \langle (A^+)(B^+)(C)(A^-B^-) \rangle$ and $\langle (A) \rangle$, $\langle (A^+)(B) \rangle$, $\langle (A^+)(B)(A^-) \rangle$ are also prefixes of α due to *merge*$(\alpha, A^+) = \alpha' = \langle (A) \rangle$ and *merge*$(\alpha, B^+) = \alpha'' = \langle (A^+)(BA^-) \rangle$. But neither $\langle (B^+)(C) \rangle$ nor $\langle (C) \rangle$ is considered as a prefix of α. $\langle (B^+)(C)(A^-B^-) \rangle$ is the postfix of the sequence α with respect to prefix $\langle (A^+) \rangle$, $\langle (C)(A^-B^-) \rangle$ is the postfix with respect to prefix $\langle (A^+)(B^+) \rangle$, and $\langle (_A^-) \rangle$ is the postfix with respect to prefix $\langle (A^+)(B) \rangle$.

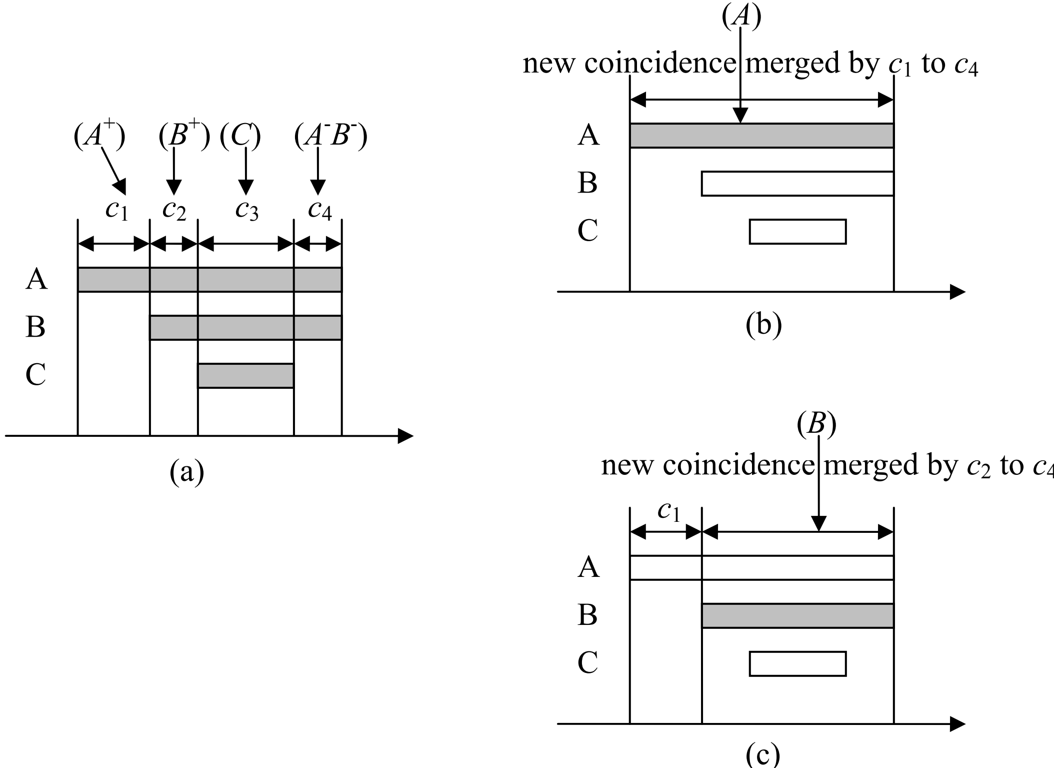


Figure 4-2 (a) Original Csequence α, (b) α' = merge (A^+, α), (b) α'' = merge

1896

4.1 Multi-projection Technique

In PrefixSpan, the relations between items are only *before*, *after* and *equal*. However, CTMiner not only considers the three relations between event slices but also considers the limitation caused by the characteristic of paired start slice and finish slice. If we adopt projection scheme without considering such limitation, then the obtained information is not sufficient. In other words, the obtained frequent temporal patterns are incomplete. We discuss the limitation and the difference of the projection scheme between PrefixSpan and CTMiner in details as follows.

For example, after projecting the time point-based event sequence S_1 = ⟨(A_1)(B_1C_1)(A_2)(B_2D_1)⟩ with respect to the prefix ⟨(A)(B)⟩ and the projected sequence ⟨(_C_1)(A_2)(B_2D_1)⟩ will be generated since the pattern (A_1 *before* B_1) occurs firstly in *S*. The

projected sequence is accurate since the relationship between any two time point-based events is simpler. Take another example as shown in Fig. 4-3, the temporal event sequence S_2 = $\langle(A_1^+)(B_1^+)(C_1)(A_1^-B_1^-)(A_2^+)(B_2^+)(C_2)(A_2^-)(B_2^-)\rangle$ is projected with respect to the prefix $\langle(A^+)(B^+)(C)\rangle$ as shown in Fig. 4-3(a). The pattern A_1^+ *before* B_1^+ occurs firstly in S_2. If we adopt projection approach of PrefixSpan without considering the limitation, the only projected sequence $\langle(\boldsymbol{A_1^-B_1^-})(A_2^+)(B_2^+)(C_2)(A_2^-)(B_2^-)\rangle$ is generated as shown in Fig. 4-3(b), where $\boldsymbol{A_1^-}$ and $\boldsymbol{B_1^-}$ in bold represent the corresponding finish slices of A_1^+ and B_1^+ in the prefix, respectively. Although the projected result looks fine, actually the obtained information is inadequate. The first occurrence of $\langle(A^+)(B^+)(C)\rangle$ in S_2 implies the temporal relation between event A and B is (A *finished-by* B), but the second occurrence of $\langle(A^+)(B^+)(C)\rangle$ implies the temporal relation is (A *overlap* B). After the projection, The postfix sequences only keep the first occurrence pattern and ignore the rest of patterns with the same prefix of $\langle(A^+)(B^+)(C)\rangle$ since A_2^- and B_2^- are not the corresponding finish slices which cannot be appended to the prefix. In other words, for a given pattern, the original projection method forms the projected database from collecting all projected sequences with regards to only the first occurrence of the prefix in each Csequence. In this paper, a new projection strategy called multi-projection is proposed to address this problem.

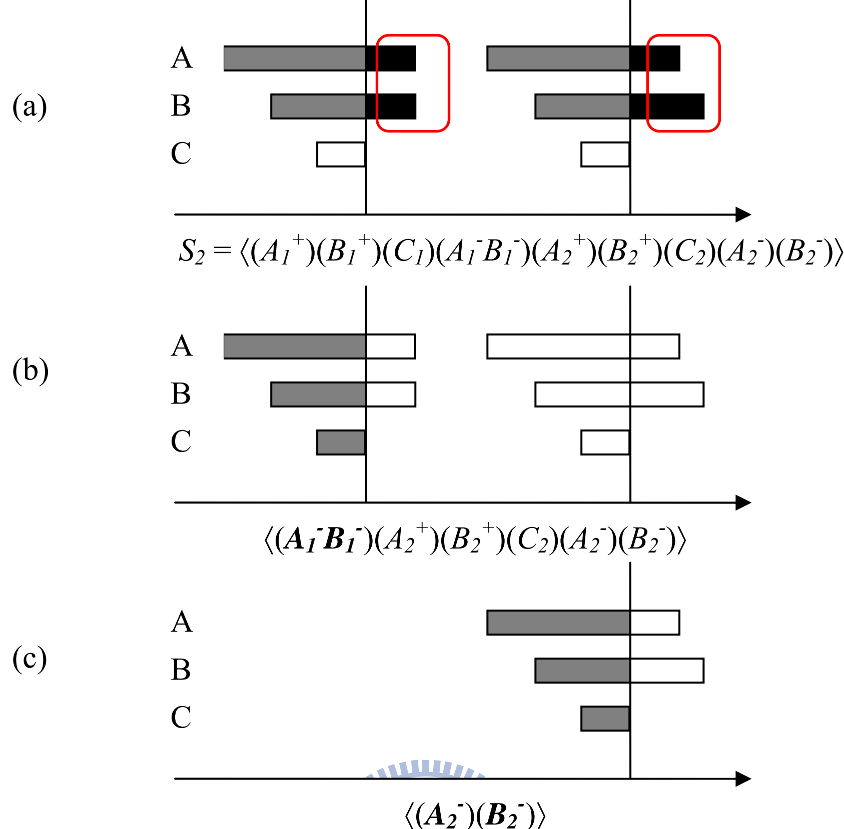


Figure 4-3 Multi-projecting prefix $\langle (A^+)(B^+)(C) \rangle$ in (a) creates two postfix sequences (b) and (c) to obtain complete frequent coincidence patterns.

Multi-projection projects every occurrence of the prefix in Csequences then collects all generated postfix sequences with respect to the prefix with start slices to form the projected database. Therefore, it may generate more than one postfix sequences. From the running example above, multi-projecting a temporal sequence shown in Fig. 4-3(a) with regard to the prefix $\langle (A^+)(B^+)(C) \rangle$ will generate two postfix sequences $\langle (\boldsymbol{A_1^-B_1^-})(A_2^+)(B_2^+)(C_2)(A_2^-)(B_2^-) \rangle$ and $\langle (\boldsymbol{A_2^-})(\boldsymbol{B_2^-}) \rangle$ as shown in Fig. 4-3(b) and Fig. 4-3(c), respectively. Obviously, the size of projected database constructed by multi-projection is dominated by the proportion of frequent multiple start slice repetitions in the Csequence. With regard to the prefix with start slices, the more occurrences of the prefix in the sequence, the more projected sequences with respect to the prefix will be generated. The workload for multi-projection is additional postfix sequences generation and collection.

To reduce the memory usage of projected databases caused by multi-projection scheme, we apply the pseudo-projection technique proposed by Pei et al. [5] to address this problem. Instead of performing physical projection, pseudo-projection registers the identifier of the sequence we need to project and the starting position of the projected suffix in the sequence. Then, a physical projection of the sequence is replaced by registering the sequence identifier and the projected position index. With this technique, the usage of memory can be reduced substantially. The projection scheme of CTMiner utilizes pseudo-projection technique to avoid physically copying postfix sequences. Therefore, the running time and memory usage of CTMiner is efficient. The details of pseudo-projection technique are described in [5]. The experimental result shows that the performance of multi-projection in both synthetic data and real data still scales well when processing considerable temporal sequences.

Chapter 5

Proposed Interval-based Event Mining Algorithm: CTMiner

In this chapter, the new algorithm called CTMiner is proposed to find all frequent temporal patterns. CTMiner utilizes the coincidence-slice concepts to accomplish the frequent time interval-based pattern mining task. It can be decomposed into three phases: **incision and projection**, **coincidence mining** and **temporal relation discovery**. Chapter 3 describes the idea and method of incision strategy to transform an event sequence in temporal database into unambiguous Csequences. And we also define prefix, projection and postfix which are different from traditional PrefixSpan algorithm and describe the projection scheme as preliminary of coincidence mining phase. In this Chapter, we will give a high level description of PrefixSpan and details CTMiner algorithm and the proposed three pruning mechanisms for coincidence pattern mining. Finally, we describe the algorithm which transforms frequent coincidence patterns back to temporal patterns in a frequent temporal pattern then discovers all the temporal relations between any two events expressed as a relation list.

In Fig. 5-1, algorithm 1, CTMiner illustrates the main framework. It first scans the temporal database to discover all frequent events and remove infrequent intervals and empty event sequences (Line 2-3, algorithm 1). Then **Incision_and_projection** is called to transform the original temporal database into coincidence database and get all projected coincidence temporal databases with respect to each frequent 1-pattern (Lines 4, algorithm 1). Then the coincidence mining task **CPrefixSpan** is called for each projected coincidence database to get frequent coincidence patterns (Lines 5, 6, algorithm 1). Finally, we discover a relation list which illustrates all temporal relations among events in a frequent coincidence pattern by called **Temporal_relation_discovery.** (Line 7, algorithm 1).

Algorithm 1: CTMiner (*D,min_sup*)

Input: A temporal database D, and the minimum support threshold *min_ sup*
Output: Frequent temporal sequential patterns L
Variables: All frequent coincidence patterns F, all frequent temporal 1-patterns $L1$, a set P which collects all projected databases with respect each 1-pattern, projected database $D_{\|a} \in P$ with respect to the prefix a
 	
1: $F \leftarrow \varnothing$, $L \leftarrow \varnothing$
2: scan temporal database D to find all frequent 1-patterns and remove infrequent intervals and empty event sequences , $L1 \leftarrow$ all frequent 1-patterns.
3: $L \leftarrow L1$;
4: P = Incision_and_Projection($L1,D,F,$ *min_sup*); //incision phase
5: for each projected database $D\|_a \in P$ do
6: CPrefixSpan($D\|_a$, *min_supp*, F); //coincidence mining phase
7: Temporal_relation_discovery(F, L); //temporal relation discovery phase
8: output frequent temporal sequential patterns in L;

Figure 5-1 The *CTMiner* algorithm.

5.1 Phase I: Incision and Projection

Given a temporal database, the events associated with the same SID can be grouped into an event sequence. In **incision and projection** phase, we scan the temporal database to handle each event sequence. Each event sequence is incised to Csequence then it is projected to the postfix sequences with respect to each frequent 1-event slice as a prefix. The postfix sequences with the same SID are grouped and dispatched to the corresponding projected database. After handling all event sequences, the procedure returns a set of projected coincidence databases with respect to frequent 1-event slices. Algorithm 2 illustrates the details of **incision and projection** as shown in Fig. 5-2.

First, each event sequence is transformed to Csequence by the following operations (Line1-20, algorithm 2). The information of start/finish time points of each event is added to a list called *time_points_list* then sorts all the records in the list in the order of time (Line 3,4, algorithm 2). Then the records are grouped into *time_list* by the same time and Csequence is

generated simultaneously (Line 5-20, algorithm 2). Two pointers *cur_time_list* and *last_time_list* point to the first two *cur_time_list*s (Line 7, algorithm 2) and the coincidence is decided by dealing the records of *cur_time_list* and *last_time_list.*

Algorithm 2: **Incision_and_Projection**(*L1,D,F,min_sup*)

Input: All frequent temporal 1-patterns *L1*, a temporal database *D*, all frequent coincidence patterns *F*, and the minimum support threshold *min_ sup*

Output: A set *P* which collects all projected databases with respect each 1-pattern,

Variable: *time_list , time_points_list,cur_time_list, last_time_list, coincidence, sequence, postfix_seq',postfix_seq'',* $D_{|p'}$, $D_{|p''}$

```
1: for each sequence s in D do
2:     sequence←∅;
3:     for each interval a in s do
4:         add the information of time points of a (a.time,a.event_type,a.type) to
             time_points_list.
5:         sort records in time_points_list by time and event type in increasing order;
6:         grouping the records in time_points_list with the same time into time_list;
7:         cur_time_list and last_time_list point to the first two time_list;
8:         while cur_time_list is not points to a null time_list do,
9:             coincidence←∅;
10:            if there exist two records i, j with type start and finish respectively in
               last_time_list do
11:                coincidence← coincidence ∪ "@";                //meet token
12:            for each record r in last_time_list do
13:                if r.type = start then                       //start slice
14:                    coincidence← coincidence ∪ "r.event_type" with "+";
15:            for each record r in time_list do
16:                if r.type = finish then                      //finish slice
17:                    coincidence← coincidence ∪ "r.event_type" with "-";
18:            merge start slice and finish slice with same event type to an intact
               slice in coincidence;                            //intact slice
19:          sequence ← sequence ◇ ⟨coincidence⟩;
20:          last_time_list and cur_time_list point to next time_list;
21:   for each frequent 1-pattern p in L1 do
22:        create start slice p' and intact slice p'' of p.
23:            postfix_seqes = Multiprojection_to_coincidence_seq(sequence,p');
```

```
24:                 postfix_seq1 = projection_to_coincidence_seq(sequence,p”);
25:                 D|p’ ← D|p’ ∪ postfix_seqes ;
26:                 D|p” ← D|p” ∪ postfix_seq1 ;
27: for each projected coincidence database D|prefix do,
28:        If |D|prefix| ≥ min_supp then
29:             P ← P ∪ D|prefix ;
30: output P;
```

Figure 5-2 The *Incision_and_Projection* algorithm.

If two records with different types exist in *last_time_list* then the meet token “@” is placed into coincidence to distinguish from two adjacent events (Line 10, 11, algorithm 2). Note that, the *type* indicates the time point either a stp or a ftp. The start slices with corresponding stps in *last_time_list* are created and placed into the coincidence and the finish slices with corresponding ftps in *cur_time_list* are also created and placed into the coincidence (Line 12-17, algorithm 2). If the start slice and the finish slice in the coincidence have the same event type, we combine them to form an intact slice (Line 18, algorithm 2). Then the ordered coincidence formed a Csequence (Line 19, algorithm 2). For each frequent 1-pattern p, two projected coincidence databases with respect to a start slice and an intact slice of p are generated to handle the temporal patterns beginning with the start slice and the intact slice, respectively (Line 21-26, algorithm 2). Output all projected coincidence databases with the number of Csequences greater than the *minimal support* (Line 27-30, algorithm 2). Finally, the temporal database has transformed to a set of projected coincidence databases which consist of start slices, finish slices, intact slices and meet toke. For the example as shown in Fig. 5-3, *Incision_and_Projection* operates on event sequence 3 in Table 3-1. The *cur_time_list* is pointing to the 6th *time_list* and the *last_time_list* always points to the previous *time_list* which *cur_time_list* is pointing to, i.e., the 6th time list = $\{E^+\}$. First, we check if there exist both start slice and finish slice in *last_time_list* to distinguish two adjacent events then put all start slices in *last_time_list* and all finish slices in *cur_time_list* into coincidence, i.e., (E^+E^-). The same event with both start slice and finish slice in coincidence

will be transformed to an intact slice, i.e., (E). Finally, the ordered coincidences form a Csequence, i.e., $\langle(A^+)(B^+A^-)(B^-)(@D^+)(E)(D^-)\rangle$.

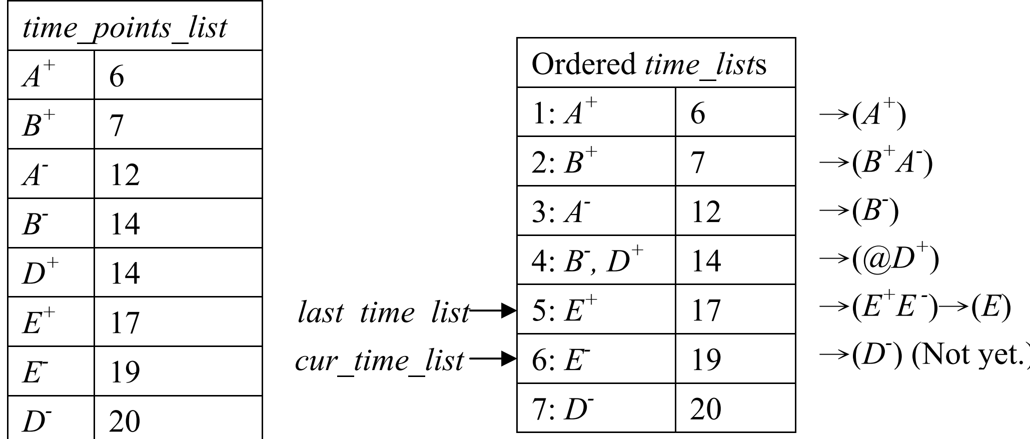


Figure 5-3 Illustration for *Incision_and_Projection* on event sequence 3 in Table 3-1. The *cur_time_list* and *last_time_list* point to the 6th and 5th *time_list*, respectively.

5.2 Phase II: Coincidence Mining

By using the concepts of the sequential pattern mining algorithm PrefixSpan [4], the mining phase of CTMiner, CPrefixSpan algorithm is proposed to discover all frequent coincidence patterns, which are defined in Def. 6. In the following, we first provide a brief description of PrefixSpan, and then we present the CPrefixSpan algorithm in details.

PrefixSpan uses a divide-and-conquer strategy to solve the sequential pattern mining problem on time point-based data. First, it scans the database to find all frequent 1-patterns, i.e., L_1. Second, suppose there are $|L_1|$ patterns in L_1, the original database is divided into $|L_1|$ partitions, where each partition is the projection of the sequence database with respect to each frequent 1-pattern as a prefix. Third, similar to the first step, each partition is treated as the original one and all the local frequent 1-patterns are found in this partition. Appending these frequent 1-patterns to the prefix will generate frequent sequential patterns with the length increased by one. Finally, recursively running step two and step three will derive all frequent

sequential patterns until prefixes cannot be extended any more. From a high-level point of view, CPrefixSpan is similar to PrefixSpan. However, the PrefixSpan uses itemsets and items to represent a time point-based event sequence and CPrefixSpan uses coincidences and event slices to represent an interval-based event sequence.

Here we discuss similarity and dissimilarity of the above mining algorithms: conceptually, the relationship of “event slice versus coincidence” is analogous to “item versus itemset.” Since we have incised each event sequence to event slices and transformed the complex relationship among events to the relationship among event slices. Though the relationship between items is the same as that of event slices i.e., “*after*”, “*before*” and “*equal*”, however, in a lower level, many implementations in CPrefixSpan are still different from those in PrefixSpan. The main reason for these differences is the characteristics of interval-based event and coincidence representation. The optimizations are proposed and adapted to the characteristics of paired start slice and finish slice. Therefore, essentially, PrefixSpan can be adopted as the fundamental extension of the proposed algorithm. The pseudo code of CPrefixSpan is shown in Fig. 5-4.

```
Algorithm 3: CPrefixSpan(D|a,min_supp,F)
Input: A projected database D|a, a prefix is represented in Csequence a, the
minimum support threshold min_supp, a set of frequent coincidence patterns F
Output: A set of updated frequent coincidence patterns F
Variables: coincidence,E1

1: for each sequence s in D|a do
2:     Count_freq_event_slices(s);
3: put all frequent 1-event slices into E1;
4: for each finish slice e in E1 do
5:     elimination_test(e, a);                    // an intact slice is formed w.r.t e
6: for each frequent 1-event slice e in E1 do
7:     a' ← append event slice e to a;
8:     construct projected database D|a';
9:     if form_a_pattern_test(a') then
10:        F←F ∪ a';
11:    if |D|a'| ≥ min_supp then
12:        call CPrefixSpan(D|a', min_supp,F);
```

Figure 5-4 The *CPrefixSpan* algorithm.

Algorithm CPrefixSpan scans projected coincidence database to collect all local frequent 1-event slices and three pruning strategies are performed here to avoid further meaningless processing. Intuitively, all event slices in postfix sequences are counted to obtain all frequent 1-event slices. An event can be represented either as a pair of start and finish slices or as an intact slice. Therefore, when a start slice is appended to the prefix we need to mark its corresponding finish slice in the postfix sequence. The finish slice will append to the prefix in the further processing. In the example as shown in Fig. 5-5, initially, we have a temporal database with frequent 1-event slices $\{A, A^+, C, C^+, D\}$ and minimum support=2. After running the CPrefixSpan once on the original database, five projected databases are created with respect to frequent 1-event slices. We utilize the projected database with prefix $\langle(A^+)\rangle$ and the potential frequent 1-event slices $\{A^-, C, C^+, C^-, D\}$ in the projected database as the example to illustrate the following three pruning strategies.

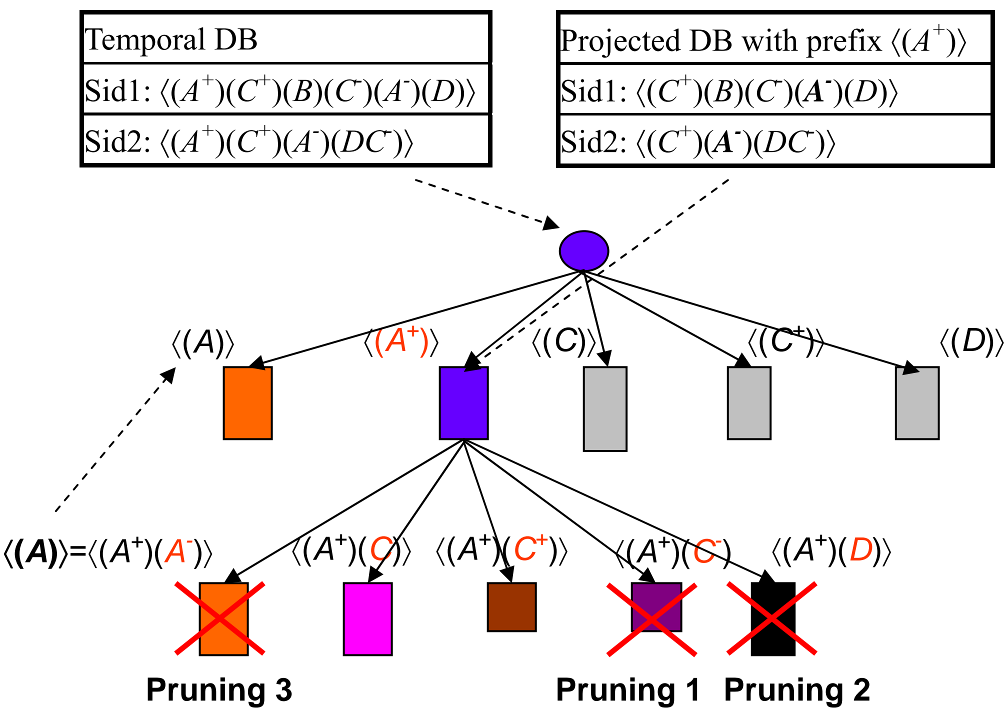


Figure 5-5 Illustrating three pruning strategies.

◆ Pruning strategy 1: The finish slices in a postfix sequence without its corresponding start slices in the prefix are not counted. Once the finish slice is appended to the prefix then the further processing will exclude its corresponding start slice. Existence of either a single start slice or a single finish slice in a pattern is meaningless. The finish slice with its corresponding start slice in the prefix is marked to speedup the further processing. For example, both $\boldsymbol{A}^-$ and C^- occurs in two event sequences in projected database with prefix $\langle(A^+)\rangle$ in Fig. 5-5 but C^- without it corresponding start slice C^+ in the prefix. If we treat C^- as the frequent 1-event slice and append it to the prefix, the new prefix $\langle(A^+)(C^-)\rangle$ and only one postfix sequence $\langle(\boldsymbol{A}^-)(D)\rangle$ are generated. The coincidence pattern $\langle(A^+)(C^-)\rangle$ is incomplete and it will not have the chance to become a complete coincidence pattern due to C^+ has been omitted by projection scheme permanently. Thus, we can prune the C^- as a frequent 1-event slice in this case.

◆ Pruning strategy 2: The event slices which occur before the first marked finish slice in postfix sequence are counted. The first marked finish slice indicates that its corresponding start slice exists in the prefix. Obviously, if the event slice after the first marked finish slice is

appended to the prefix then further processing will omit the first marked finish slice and the property of paired start slice and finish slice is violated. Therefore, further processing with respect to the prefix is meaningless. For the running example in Fig. 5-5, the intact slice D after the first marked event slice $\boldsymbol{A}^{-}$ is a frequent 1-event slice. The new prefix $\langle(A^{+})(D)\rangle$ is formed by appending the frequent intact slice D then the only postfix sequence $\langle(C^{-})\rangle$ is created with respect to the new prefix. We can find that the marked finish slices $\boldsymbol{A_1}^{-}$ is omitted in the further processing permanently. Therefore, the pattern $\langle(A^{+})(D)\rangle$ being a prefix is incomplete and meaningless for the further processing. The function Count_freq_event_slices(α) implements the above two pruning strategies, while counting frequent 1 event slices in postfix sequence α (Line 2, algorithm 3).

◆ Pruning strategy 3: The new prefix is formed by appending each frequent finish slice to the original prefix. The third pruning strategy tests that the finish slice and its corresponding start slice forms the intact slice properly in the new prefix. If the intact slice forms without any event slice elimination, the further processing of the new prefix can be skipped due to divide and conquer strategy. Actually, the further processing with respect to the new prefix is performed by another data partition with the new prefix. Still, for the running example in Fig. 5-5, $\boldsymbol{A}^{-}$ is a frequent 1-event slice. The new prefix $\langle(A^{+})(A^{-})\rangle$ is generated and the pattern actually can instead be an intact slice $\langle(A)\rangle$. According to the divide and conquer strategy, the further processing is the same as the processing on the projected database with prefix $\langle(A)\rangle$ which can be omitted. The third pruning strategy is implemented in function *elimination_test*(e, a) (Line 5, algorithm 3) where e is a frequent finish slice and a is the prefix. For the example shown in Fig. 5-6, if we append A^{-} to the prefix $\langle(A^{+}B^{+}C^{+})\rangle$ to form a new prefix $\langle(A^{+}B^{+}C^{+})(A^{-})\rangle$, the following conditions are checked to determine whether *elimination_test*(A^{-}, a) operation works. We assume A^{+} and A^{-} in coincidence c_i and c_j in a, respectively. (1) There is no coincidence between c_i and c_j in a. If there are coincidences between c_i and c_j, then to form the intact slice A has to eliminate event slices in those

coincidences between c_i and c_j, i.e., A^+ and A^- cannot be merged properly. (2) There is no other event slices in c_i and c_j besides the start slices occurring after A^+ in c_i, i.e., the relation between event A and the event with the start slice may start or equal due to the same start time. After operating the function *elimination_test*, the new prefix can be represented as $\langle(AB^+C^+)\rangle$, i.e., A^+ and A^- form A, and further processing of the prefix can be eliminated.

After the above two functions, each frequent 1-event slices can be appended to the original prefix to generate new pattern with the length increased by one. This way, the prefixes are successfully extended.

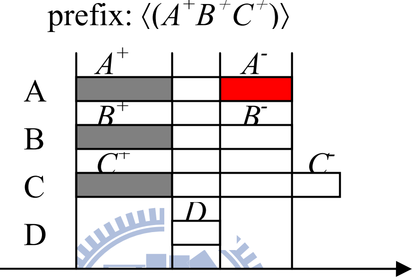


Figure 5-6 Illustration for *elimination_test* on A^- successfully with all possible correlative event slices in the prefix.

In PrefixSpan algorithm, the frequent patterns are output by appending each frequent 1-event to the prefix. But in CPrefixSpan, a lot of prefixes are treated as intermediate coincidence patterns but not frequent coincidence patterns. The function *form_a_pattern_test*(p) verifies the prefix p either an intermediate coincidence pattern or a frequent coincidence pattern (Line 9, algorithm 3). The prefix is treated as a frequent coincidence pattern if all the start slices and finish slices are paired correctly in the prefix. Finally, if the number of sequences in projected database is greater than *min_supp*, then recursively run the projected database with respect to the extended prefix until the prefix cannot be extended successfully. Then, all frequent coincidence patterns will be discovered (Lines 11, 12, algorithm 3).

We take the database in Table 5-1 with *min_sup* = 2 as an example. There are 17 event records which can be regarded as 4 event sequences in the temporal database. After scanning

the original temporal database, we find all the frequent 1-patterns. They are $\langle A \rangle$: 3, $\langle B \rangle$: 4, $\langle D \rangle$: 4, and $\langle E \rangle$: 4, where the notation" $\langle pattern \rangle$: *count*" represents the pattern and its associated support count.

For each original event sequence in Table 5-1, the Csequences are constructed and projected with respect to event slices created by frequent 1-patterns as shown in the first two columns in Table 5-2 and the pictorial examples corresponding to the projected databases are shown in Fig. 5-7. Furthermore, we take the coincidence database with event A as example to discuss in details. We have to consider the patterns prefixed with intact slice $\langle A \rangle$ and start slice $\langle A^{+} \rangle$. Note that when counting the support and constructing projected database with regard to intact slice $\langle A \rangle$, we also require considering the occurrence of start slice $\langle A^{+} \rangle$ in sequences since both represent the existence of event A. The projected sequences with respect to $\langle \mathrm{A} \rangle$ has 3 sequences: SID 1: $\langle (C^{+})(B^{-}C^{-})(D^{+})(E)(D^{-}) \rangle$, SID3: $\langle (B^{-})(@D^{+})(E)(D^{-}) \rangle$ and SID 4: $\langle (D^{+})(E)(D^{-}) \rangle$. Simultaneously, we also project sequence with respect to $\langle A^{+} \rangle$, so we have 2 sequences: $\langle (B^{+}C^{+}\mathbf{A}^{-})(B^{-}C^{-})(D^{+})(E)(D^{-}) \rangle$ and $\langle (B^{+}\mathbf{A}^{-})(B^{-})(@D^{+})(E)(D^{-}) \rangle$. Then we collect the corresponding sequences with respect to the coincidence prefix to coincidence database, so both $D_{|\langle (A) \rangle}$ and $D_{|\langle (A+) \rangle}$ are obtained. Continuing the recursive process with the $D_{|\langle (A) \rangle}$ and $D_{|\langle (A^{+}) \rangle}$, we can discover all frequent coincidence patterns prefixed with $\langle (A) \rangle$ and $\langle (A^{+}) \rangle$, respectively. The last column in Table 5-2 lists all frequent coincidence patterns.

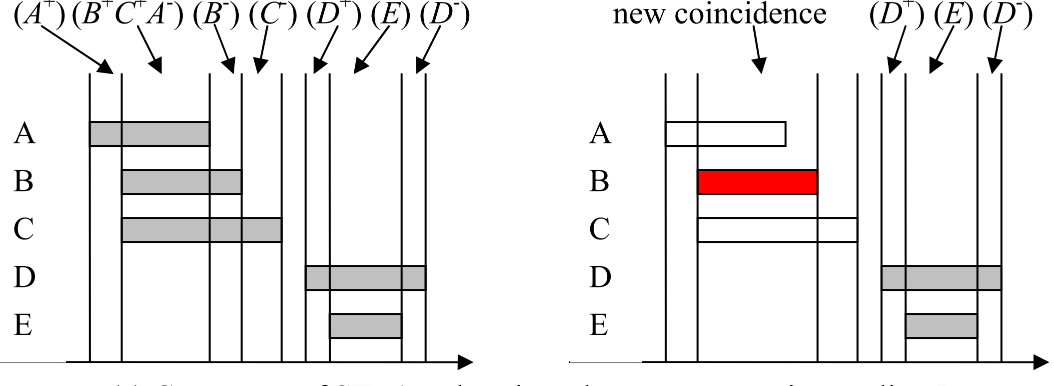
Original coincidence database with frequent 1 event slices
Sequence 1: $\langle(A^+)(B^+A^-)(B^-)(D^+)(E)(D^-)\rangle$
Sequence 2: $\langle(B)(D^+)(E)(D^-)\rangle$
Sequence 3: $\langle(A^+)(B^+A^-)(B^-)(@D^+)(E)(D^-)\rangle$
Sequence 4: $\langle(B)(A)(D^+)(E)(D^-)\rangle$

Table 5-1 Coincidence database.

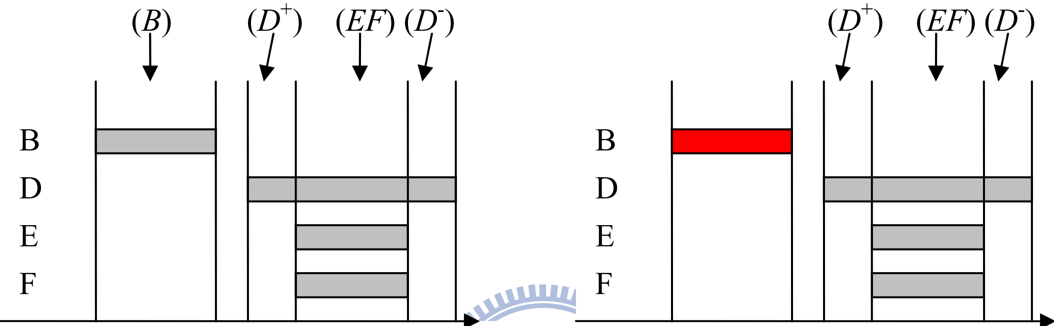
prefix	Projected database	Frequent coincidence patterns
$\langle(A)\rangle$	1:$\langle(B^-)(D^+)(E)(D^-)\rangle$ 3:$\langle(B^-)(@D^+)(E)(D^-)\rangle$ 4:$\langle(D^+)(E)(D^-)\rangle$	$\langle(A)(D)\rangle$ $\langle(A)(E)\rangle$ $\langle(A)(D^+)(E)(D^-)\rangle$
$\langle(A^+)\rangle$	1:$\langle(B^+\mathbf{A}^-)(B^-)(D^+)(E)(D^-)\rangle$ 3:$\langle(B^+\mathbf{A}^-)(B^-)(@D^+)(E)(D^-)\rangle$	$\langle(A^+)(A^-B^+)(B^-)\rangle$ $\langle(A^+)(A^-B^+)(B^-)(D)\rangle$ $\langle(A^+)(A^-B^+)(B^-)(E)\rangle$ $\langle(A^+)(A^-B^+)(B^-)(D^+)(E)(D^-)\rangle$
$\langle(B)\rangle$	1:$\langle(D^+)(E)(D^-)\rangle$ 2:$\langle(D^+)(E)(D^-)\rangle$ 3:$\langle(@D^+)(E)(D^-)\rangle$ 4:$\langle(A)(D^+)(E)(D^-)\rangle$	$\langle(B)(D)\rangle$ $\langle(B)(E)\rangle$ $\langle(B)(D^+)(E)(D^-)\rangle$
$\langle(B^+)\rangle$	1:$\langle(_A^-)(\mathbf{B}^-)(D^+)(E)(D^-)\rangle$[1] 3:$\langle(_A^-)(\mathbf{B}^-)(@D^+)(E)(D^-)\rangle$	$\varnothing$
$\langle(D)\rangle$	$\varnothing$	$\varnothing$
$\langle(D^+)\rangle$	1:$\langle(E)(\mathbf{D}^-)\rangle$ 2:$\langle(E)(\mathbf{D}^-)\rangle$ 3: $\langle(E)(\mathbf{D}^-)\rangle$ 4: $\langle(E)(\mathbf{D}^-)\rangle$	$\langle(D^+)(E)(D^-)\rangle$
$\langle(E)\rangle$	$\varnothing$	$\varnothing$
$\langle(E^+)\rangle$	$\varnothing$	$\varnothing$

Table 5-2 Projected databases and frequent temporal patterns

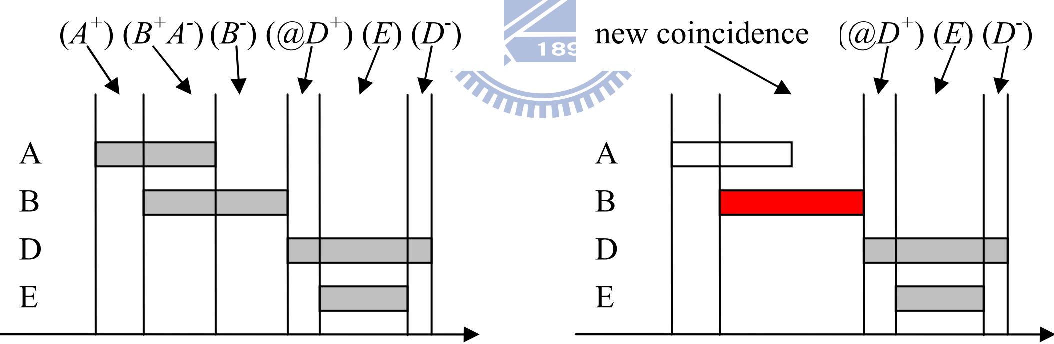
[1]. Note that, the first coincidence with "_" in projected Csequence indicates that the last coincidence in the prefix is the same as the first coincidence in the projected Csequence. The finish slice of italic and bold indicates which has corresponding start slice in the prefix.



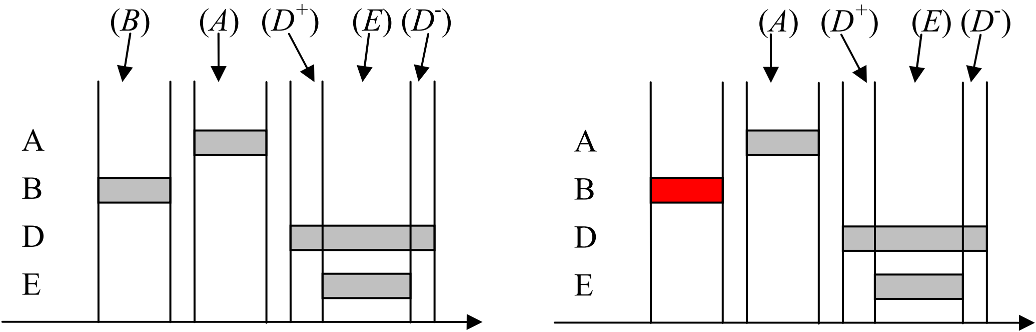
(a) Csequence of SID 1 and projected sequence w.r.t intact slice *B*.



(b) Csequence of SID and projected sequence w.r.t intact slice *B*.



(c) Csequence of SID 3 and projected sequence w.r.t intact slice *B*.



(d) Csequence of SID 4 and projected sequence w.r.t intact slice *B*.

Figure 5-7 Database of Csequences and projected databases w.r.t intact slice *B*

5.3 Phase III: Temporal Relation Discovery

The frequent coincidence representation can describe the relations among events in a pattern non-ambiguously. This representation utilizes the slice-coincidence architecture to facilitate the mining process of interval-based pattern discovery effectively and efficiently since the complex relationship among event slices is simple.

The objective of last phase is to present each coincidence pattern to a relation list which consists of temporal relations in a coincidence pattern. We use positions of coincidences to represent the relative positions of time points, named *pseudo time points*, of an event and compare the *pseudo time points* to determine the relation between any two events as the last column of Table 1-1.

The detailed algorithm of temporal relation discovery is shown in Fig. 5-8. An intact slice or a start slice in coincidence pattern implies the existence of an event and we set *pseudo time points* of each event by the following operations. For an intact slice of an event in coincidence c_i, we set both its' pseudo stp and pseudo ftp to i (Line 5-6, 11-12, Algorithm 4). For a start slice of an event in c_i and the corresponding finish slice in c_j, we set its' pseudo stp to i and pseudo ftp to j (Line 7-9, 13-15, Algorithm 4). The temporal relation of two events is determined by comparing the *pseudo time points* of them and a meet token "@" assists to distinguish the relation of two events in two consecutive coincidences being "*meet*" or "*before*" (Line 16-19 Algorithm 4).

Algorithm 4: Temporal_Relation_Discovery(*F*,*L*)
Input: a set of frequent coincidence patterns *F*, a set of frequent temporal patterns *L* Output: a set of frequent temporal patterns *L* 1: for each coincidence pattern *p* in *F* do 2: *Relation_list*←∅; 3: for each event slice *e1* of coincidence pattern *p* in coincidence c_i do 4: *src.start_time*=*src.finish_time*=*des.start_time*=*des.finish_time*=0; 5: if *e1.type*=intact slice then 6: *src.start_time* = *src.finish_time*= i; 7: else if *e1.type*=start slice then 8: get *e1'* is corresponding finish slice of *e* in coincidence c_k 9: *src.start_time*=i; *src.finish_time*=k; 10: for each event slice *e2* in coincidence c_j occurs after *e1* do 11: if *e2.type*=intact slice then 12: *des.start_time* = *des.finish_time*= j; 13: else if *e2.type*=start slice then 14: get *e2'* is corresponding finish slice of *e2* in coincidence c_l 15: *des.start_time*=j; *des.finish_time*=l; 16: if *src.finish_time*+1=*des.start_time* and the *src.finish_time*-th coincidence with "@" then 17: *relation*=meet; 18: else 19: *relation*=get_relation(*src*,*des*); 20: *Relation_list* ← *Relation_list* ∪ (*e1.event_type relation e2.event_type*); 21: L← L ∪ *Relation_list*; 22:output L;

Figure 5-8 The *Temporal_Relation_Discovery* algorithm.

In the example shown in Fig. 5-9(a), there is a frequent coincidence pattern $\langle(A^+)(BC^+)(A^-)(@C^-D)\rangle$. First, we set *pseudo time points* for each event, then (A: 1,3), (B: 2,2), (C: 2,4) and (D: 4,4) are obtained. Then, by comparing the *pseudo time points* with each event, the temporal relations (A *contain* B), (A *overlaps* C), (A *meet* D), (B *starts* C), (B *before* D), (C *finished-by* D) of the pattern are obtained as shown in Fig. 5-9(b). Note that, (X: i,j) denotes event X with stp at i and ftp at j and X.s, X.f denotes the stp and ftp of event X.

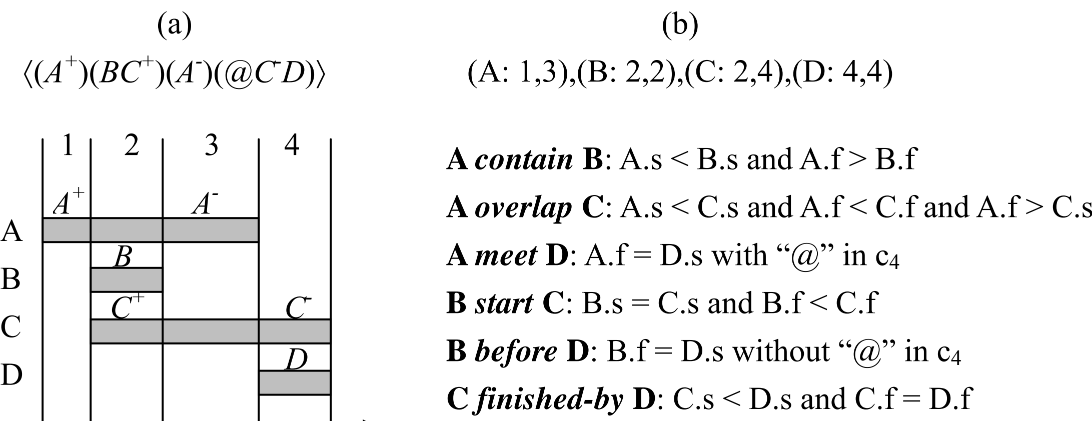


Figure 5-9 Illustration for discovering temporal relations. (a) illustrates the Csequence and (b) illustrates the relations determined by comparing the pseudo time points.

5.4 Handling large databases

The CTMiner algorithm only works if the database fits into memory. If the database is too large to fit into memory, the frequent temporal patterns are discovered by partition-and-validation technique. First, the database is partitioned so that each partition can be processed in memory by CTMiner. A temporal pattern is frequent in the database it has to be frequent in at least one partition. Thus, we can obtain the set of potential frequent patterns by collecting the discovered patterns after running CTMiner on those partitions. Then, we validate the frequency of each potential frequent pattern. The validation needs one more data scan to get all frequent temporal patterns.

Chapter 6

Experimental Result

To evaluate the performance of CTMiner, three temporal pattern mining algorithms, TPrefixSpan [11], H-DFS [10], and IEMiner [13] were implemented for comparison. All algorithms were implemented in C++ language and tested on a Pentium D 3.0 GHz with 4.0 GB of main memory running Windows XP system. The comprehensive performance study has been conducted on both synthetic and real datasets. In section 6.1.1, to compare the performance of CTMiner, TPrefixSpan, H-DFS and IEMiner, we perform experiments for the four algorithms on small (10K), medium (100K) and large (200K) synthetic datasets and varying the minimal support threshold. We discuss the memory usage of each algorithm in section 6.1.2. In section 6.1.3, to verify the scalability of CTMiner, we also test the CTMiner on different sizes of synthetic datasets. Besides the experiments on synthetic datasets, we also perform CTMiner on real dataset of library lending which is described in section 6.2. The detailed description of the parameters in the experiment is listed in Table 6-1.

Parameters	Description
$\|D\|$	Number of customers
$\|C\|$	Average number of transactions per customer
$\|T\|$	Average number of items per transaction
$\|S\|$	Average length of maximal potentially large sequences
$\|I\|$	Average size of itemsets in maximal potentially large sequences
Ns	Number of maximal potentially large sequences
Ni	Number of maximal potentially large itemsets
N	Number of items

Table 6-1 Parameters of synthetic data generator

6.1 Experiments on synthetic datasets

6.1.1 Runtime comparisons

Some parameters are fixed in the runtime experiments: $|C|=10$, $|T|=2.5$, $|S|=4$, $|I|=1.25$, $Ns=500$, $Ni=2,500$. That means that the average length of sequences is 25, i.e., $|C|\times|T|$, and the average length of temporal patterns is 6, i.e., $|S|\times|I|$, and parameters Ns and Ni are set to small values which indicates plenty temporal patterns will be generated. Three runtime experiments on different data sizes will be testified individually which are small, medium and large datasets. The first experiment testifies small dataset of 10K sequences in temporal database and 500 different event types and varying minimum support threshold from 4 % to 1 %.

Fig. 6-1 and Fig. 6-2 illustrate the running time and the number of temporal patterns of small dataset with respect to different minimum support threshold, respectively. Fig. 6-3 shows the distribution of the length of frequent temporal patterns. Obviously, when the minimum support threshold decreases, the running time required for all algorithms increases. However, the runtime for IEMiner, H-DFS and TPrefixSpan increase drastically compared to CTMiner. When minimum support is 1 %, the data set contains a large number of frequent temporal patterns (3,184). CTMiner takes 509 seconds, which is 5 times faster than TPrefixSpan (2,532 seconds), more than 8 times faster than IEMiner (4,337 seconds) and more than 13 times faster than H-DFS (6,676 seconds).

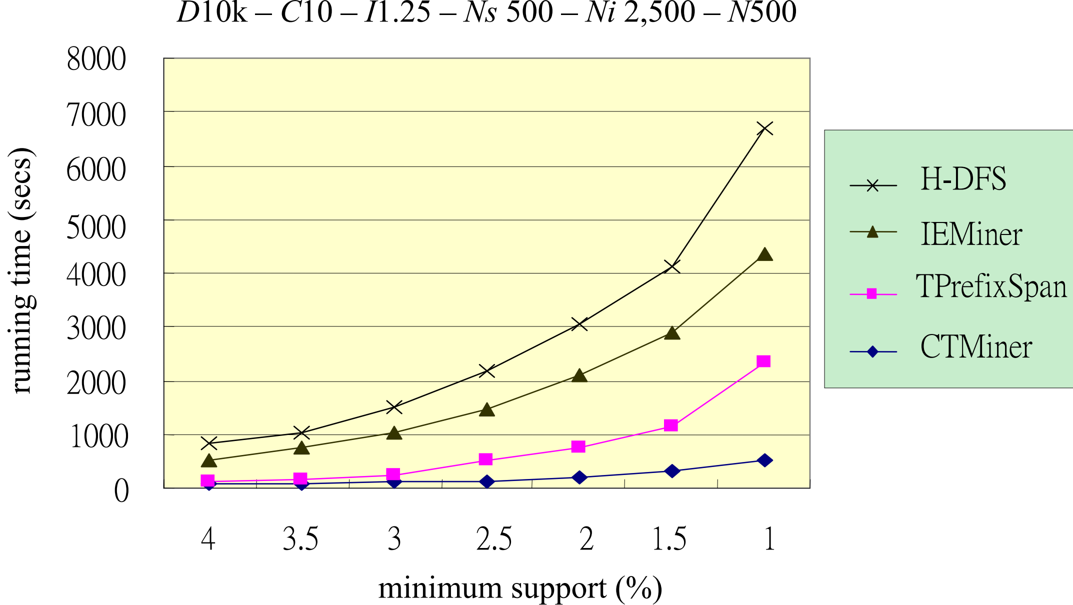


Figure 6-1 Performance of the four algorithms on data set with *D*10k – *C*10 – *I*1.25 – *Ns* 500 – *Ni* 2,500 – *N*10k

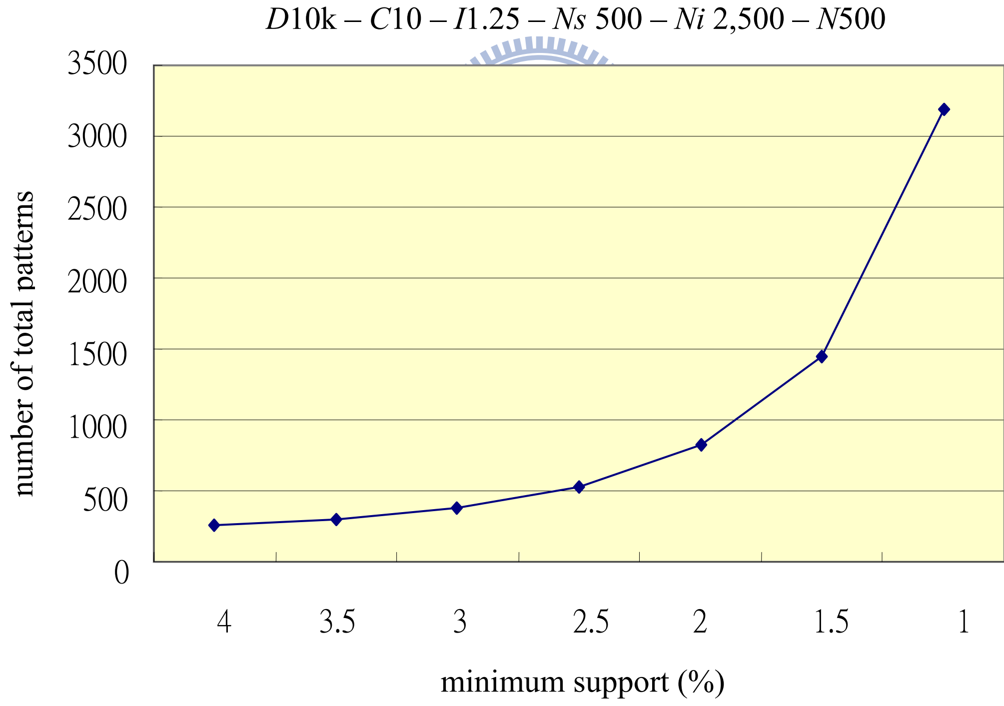


Figure 6-2 The number of generated frequent patterns on dataset *D*10k – *C*20 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*500.

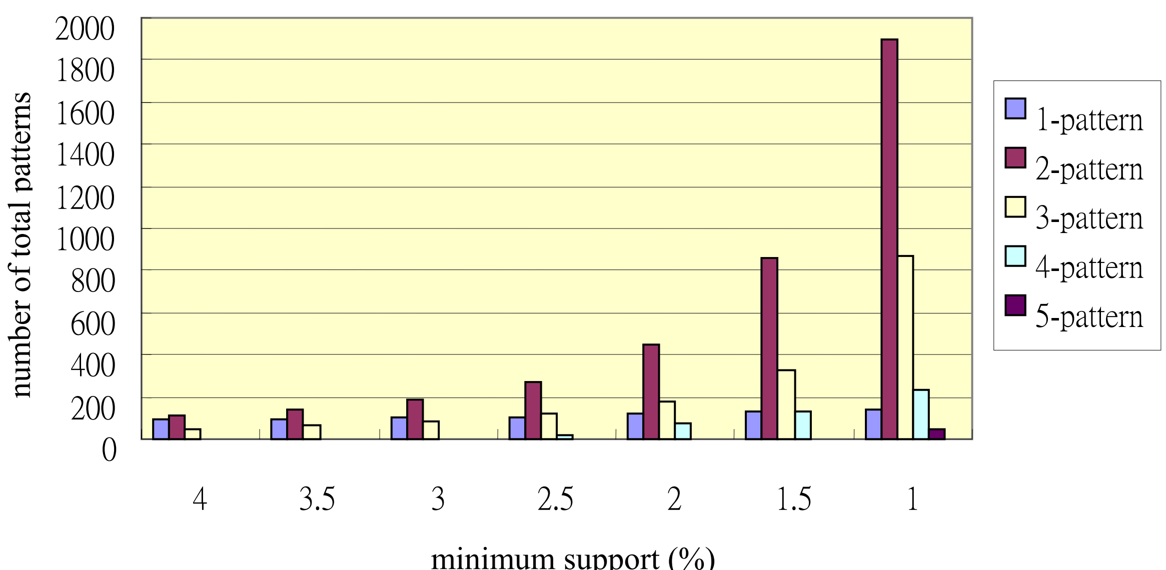


Figure 6-3 The distribution of frequent patterns of dataset with D10k – C20 – I2.5 – Ns 500 – Ni 2,500 – N500

The second experiment testifies medium dataset of 100K sequences in temporal database and 10K different event types and varying minimal support threshold from 1 % to 0.5%. Both the data size and event types of the medium dataset are 10 times larger than the small dataset. Fig 6-4 and Fig. 6-5 show the running time and number of temporal patterns of medium dataset with respect to different minimum support threshold, respectively. The Fig. 6-6 shows the distribution of the length of frequent temporal patterns. The data set contains a large number of frequent temporal patterns when minimum support is reduced to 0.5 % (2,880). CTMiner takes 4,695 seconds, which is 4 times faster than TPrefixSpan (18,789 seconds), more than 8 times faster than IEMiner (38,678 seconds) and more than 12 times faster than H-DFS (59,489 seconds).

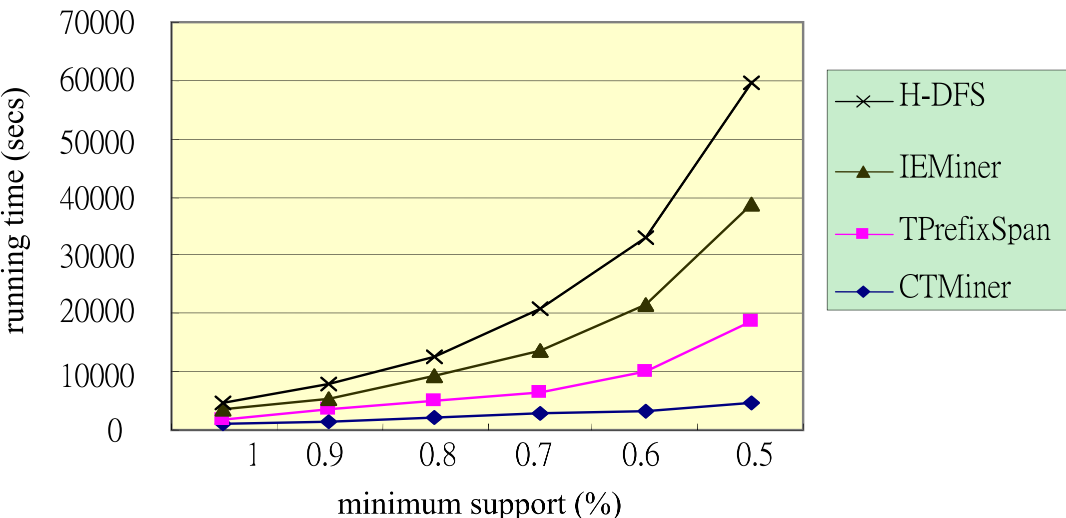


Figure 6-4 Performance of the four algorithms on data set with *D*100k – *C*10 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k

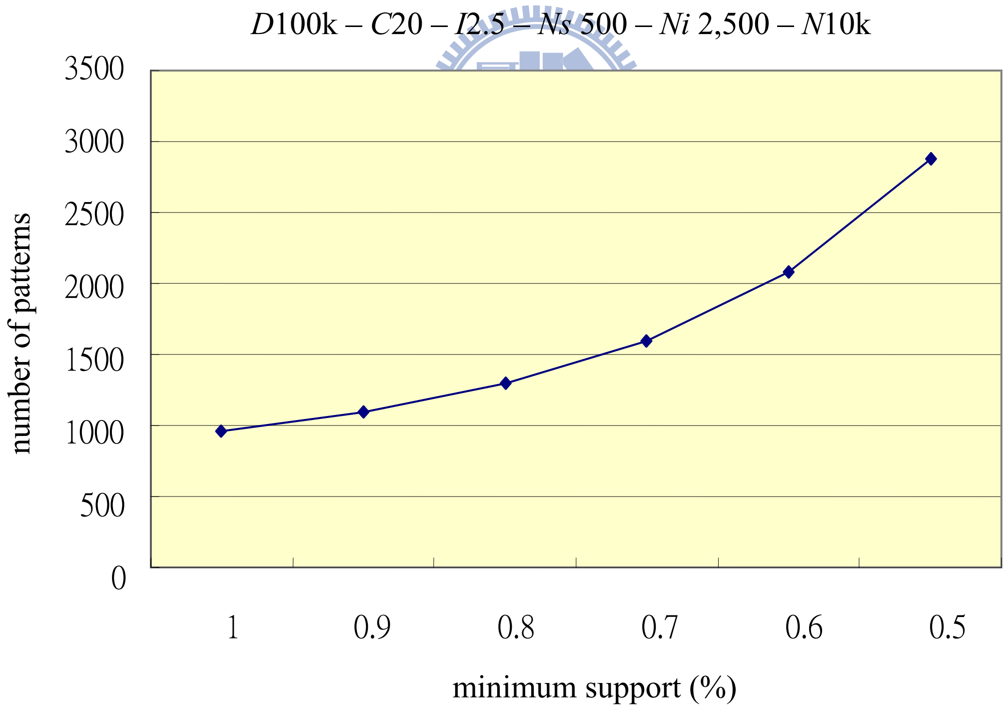


Figure 6-5 The number of generated frequent patterns on dataset with *D*100k – *C*20 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k.

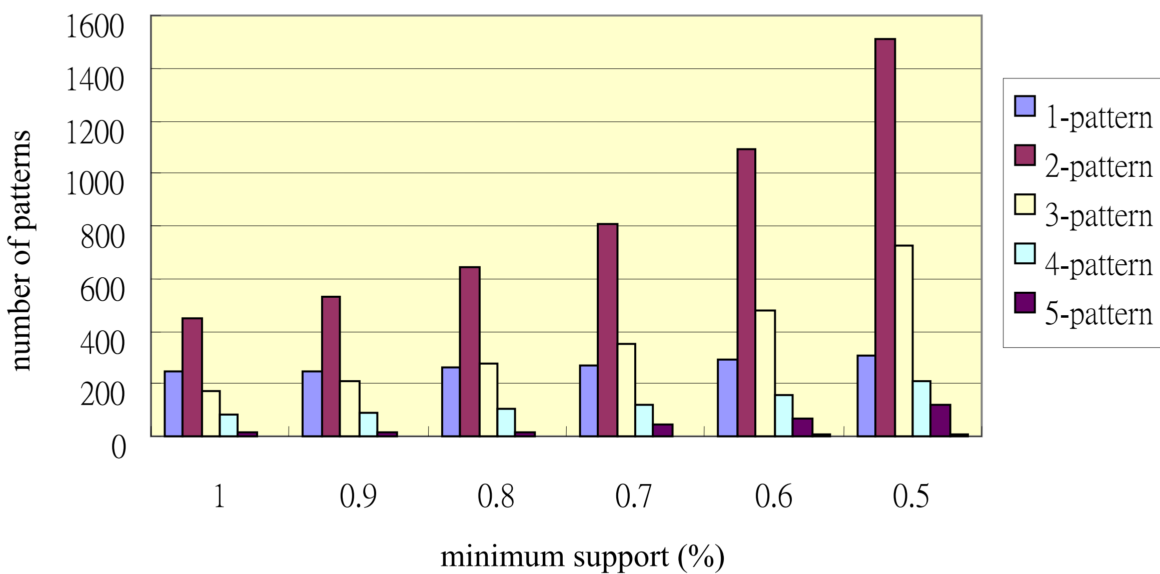


Figure 6-6 The pattern length distribution of frequent patterns on dataset with *D*100k – *C*20 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k.

The third experiment testifies large dataset of 200K sequences in temporal database and 10K different event types and varying minimum support threshold from 1 % to 0.5%. The data size of the large dataset is 2 times larger than the medium dataset. As shown in Fig. 6-7, when minimum support is reduced to 0.5 %, CTMiner takes 6,257 seconds, which is more than 4 times faster than TPrefixSpan (26,751 seconds), more than 9 times faster than IEMiner (56,972 seconds), while H-DFS never terminates in the experiment. Fig. 6-8 and Fig. 6-9 show the number of temporal patterns and the distribution of the length of frequent temporal patterns under different minimal supports. The experiments indicate that even with extremely low support and a large number of frequent patterns; CTMiner algorithm is still efficient and outperforms state-of-the-art algorithms.

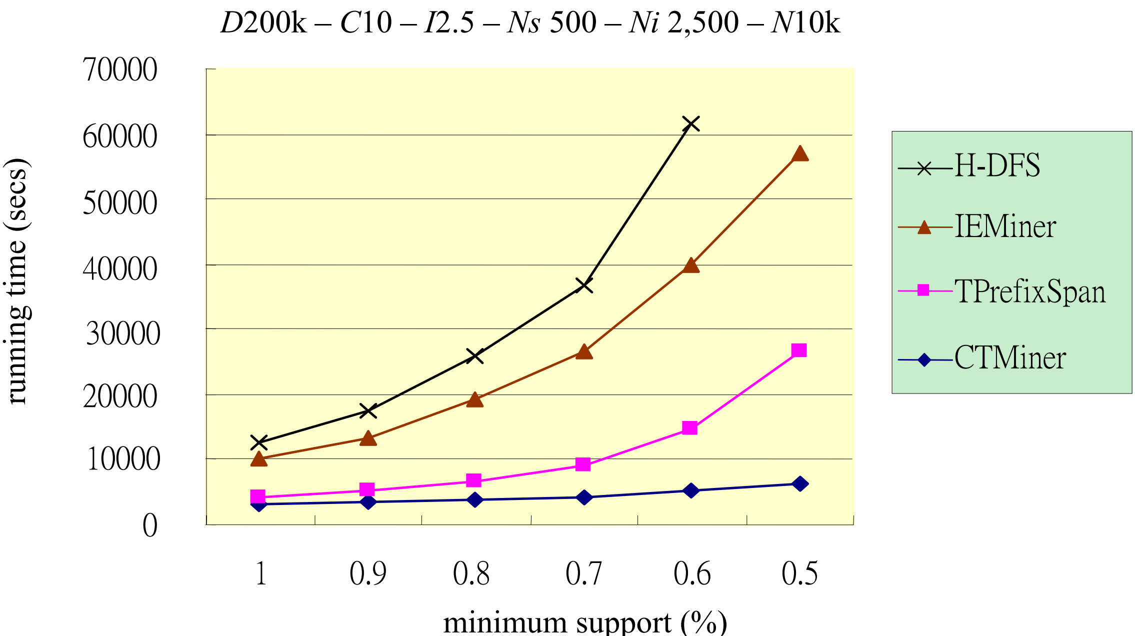


Figure 6-7 Performance of the four algorithms on data set with *D*200k – *C*10 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k

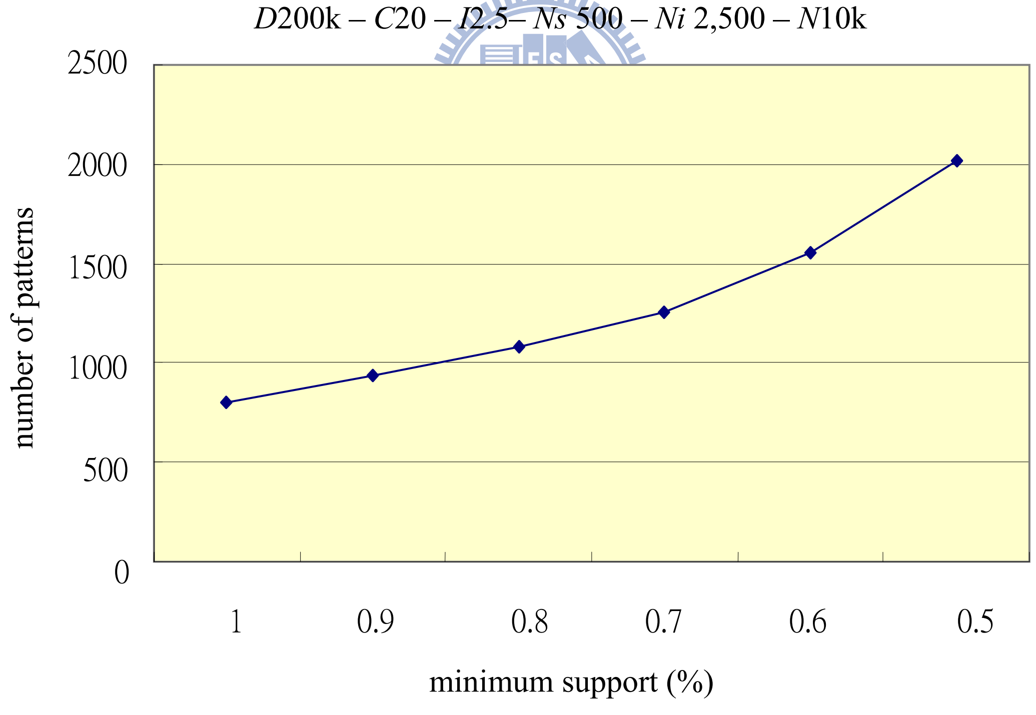


Figure 6-8 The number of generated frequent pattern on dataset with *D*200k – *C*20 – *I*2.5– *Ns* 500 – *Ni* 2,500 – *N*10k

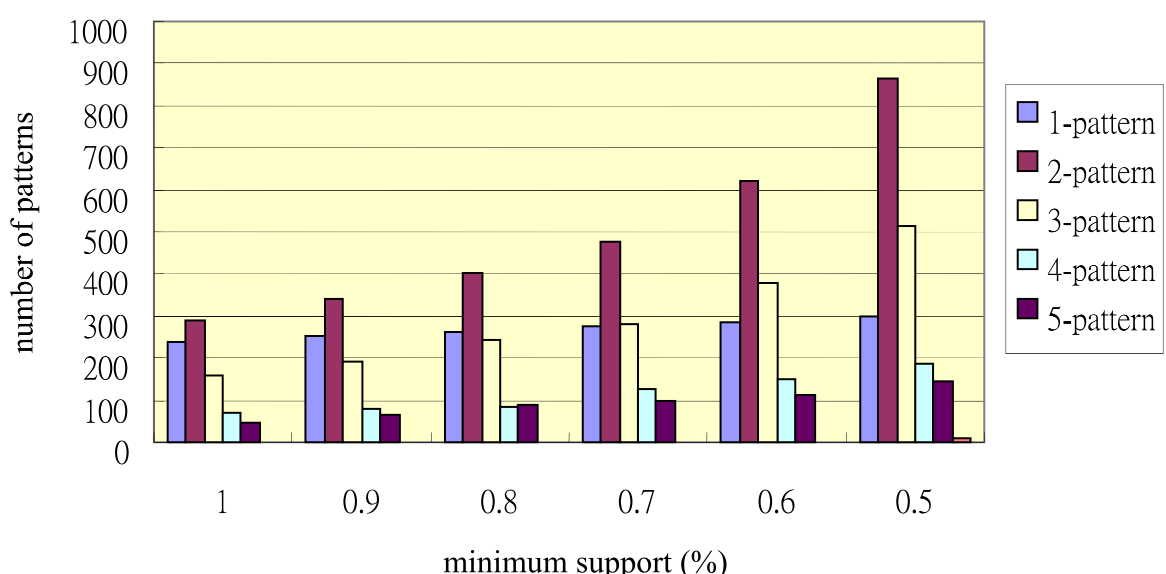


Figure 6-9 The pattern length distribution of frequent patterns on dataset with *D*200k – *C*20 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k.

6.1.2 Discussion of memory usage

The memory usage of the four algorithms on medium dataset of 100K is shown in Fig. 6-10. The H-DFS algorithm uses a huge amount of memory space from 712MB to 2.04GB because it stores a huge amount of related records of frequent 2-patterns and brute-and-force enumeration mining strategy. The TPrefixSpan algorithm also uses a lot of memory space from 524MB to 684 MB due to projected databases creation during the mining processing and without memory indexing technique. The proposed CTMiner algorithm uses the memory indexing to effectively reduce the memory space usage for storing projected databases which are generated by the proposed multi-projection scheme. The memory usage of CTMiner from 153MB to 330MB is also good. Last, IEMiner only temporally preserves candidate patterns and intermediate patterns i.e., prefixes of candidate patterns, which is generated by the support counting scheme and only needs one database scan. The IEMiner uses least amount of memory space from 61MB to 96MB but complexity representation causes running time of experiments unacceptable.

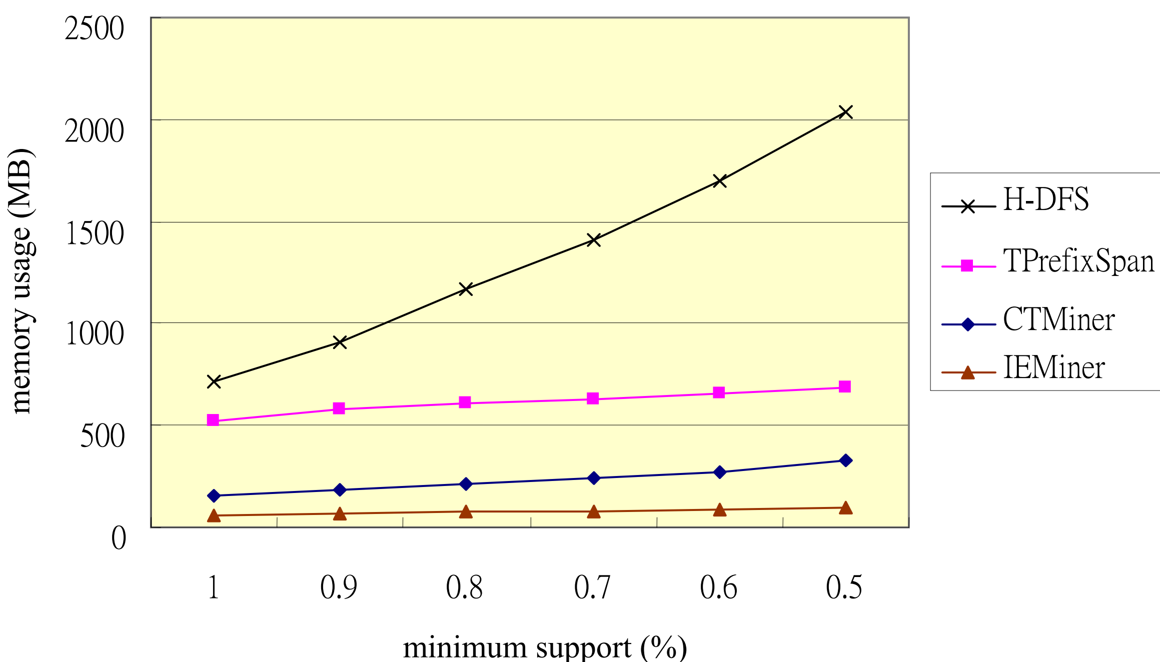


Figure 6-10 Memory usage comparison of the four algorithms on data set with *D*100k – *C*10 – *I*2.5 – *Ns* 500 – *Ni* 2,500 – *N*10k

6.1.3 Scalability

In this section, we study the scalability of the CTMiner algorithm. Fig. 6-11 and Fig. 6-12 show the results of scalability tests of the CTMiner algorithm, with the database size from 100K to 500K sequences and varying minimum support thresholds from 3% to 1%. The parameters of dataset are fixed as follows: $|C|=10$, $|T|=2.5$, $|S|=4$, $|I|=1.25$, $Ns=5{,}000$, $Ni=25{,}000$ and $N=10K$. The average length of the sequence is 25 and the number of event types in the database is 10K. 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 shown in Fig. 6-11 and Fig. 6-12. As can be seen, CTMiner is linearly scalable with different minimum support thresholds due to the compact and efficient coincidence representation and the proposed algorithm does not require candidate generation and test. When the number of frequent patterns is large, the runtime of CTMiner still increases

linearly with respect to different database sizes.

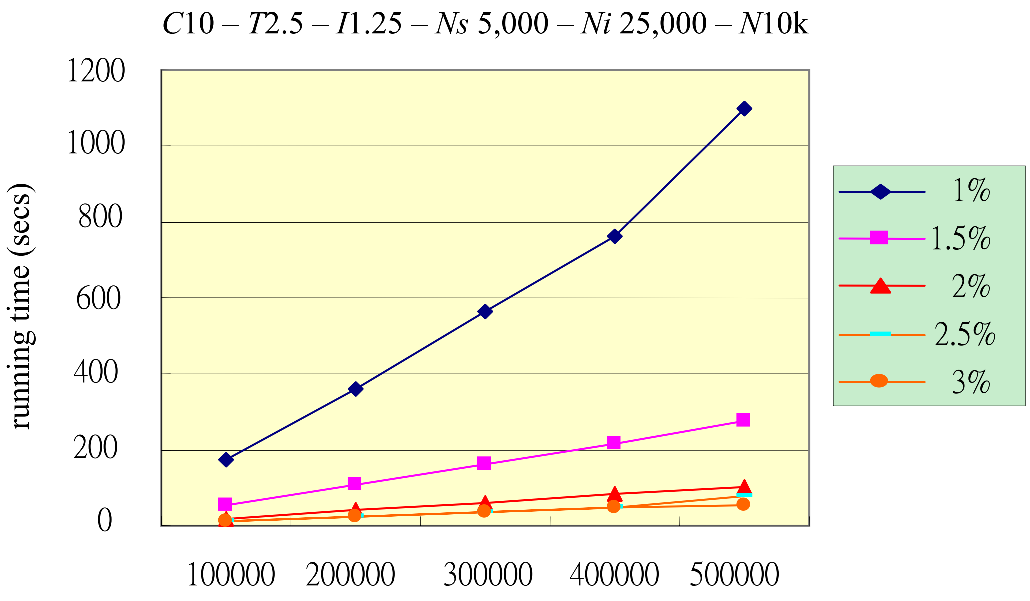


Figure 6-11 Scalability test of the CTMiner algorithm with different database size and minimum supports.

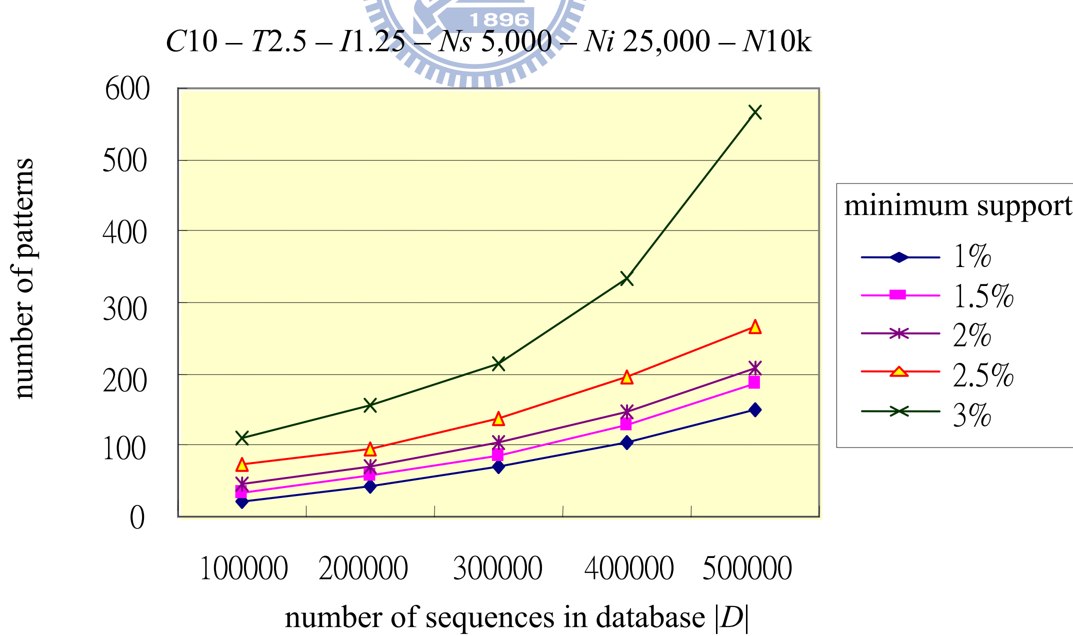


Figure 6-12 The number of generated frequent patterns with different database sizes and minimum supports.

6.2 Experiment on Real world dataset

In addition to using synthetic data sets, we have also performed experiments on real world dataset to compare the performance and validate the applicability of time interval-based pattern mining. The database used in the experiments collected 1,098,142 library records (including borrowing and returning) for last three years from the National Chiao Tung University Library. The experimental database includes 206,844 books, i.e., N=206,844, 28,339 readers and $|D|$=28,339. An event is constructed by a book ID and its associated borrowing and returning time. The size of the database is the number of sequences in the database i.e., total 28,339 readers. Fig. 6-13 indicates the running time of four temporal pattern mining algorithm with varying minimum support thresholds from 0.1 % to 0.05 % and the number of detected patterns under different thresholds is shown in Fig. 6-14. The distribution of the length of frequent temporal patterns is shown in Fig. 6-15. The experiments show that when the minimum support is greater than 0.1 %, most of generated frequent patterns are of length one or two. As the minimum support drops down to 0.05 %, there are 14,549 frequent patterns and the running time of CTMiner takes 4,771 seconds, which is about 2 times faster than TPrefixSpan (8,235 seconds), about 4 times faster than IEMiner (15,424 seconds) and H-DFS has never terminated.

We apply the CTMiner algorithm on books borrowing dataset to extract the readers' behavior. The experimental result shows that a lot of frequent patterns are related to a series of TV soaps or books. For instance, the frequent temporal pattern, "Friends of season 1 *overlaps* Friends of season 2" ^ "Friends of season 1 *before* Friends of season 3" ^ "Friends of season 2 *before* Friends of season 3", indicates users' behavior especially on borrowing a series of TV soaps. When a user wants to borrow a series of TV soaps, he always likely holds as many videos as he can.

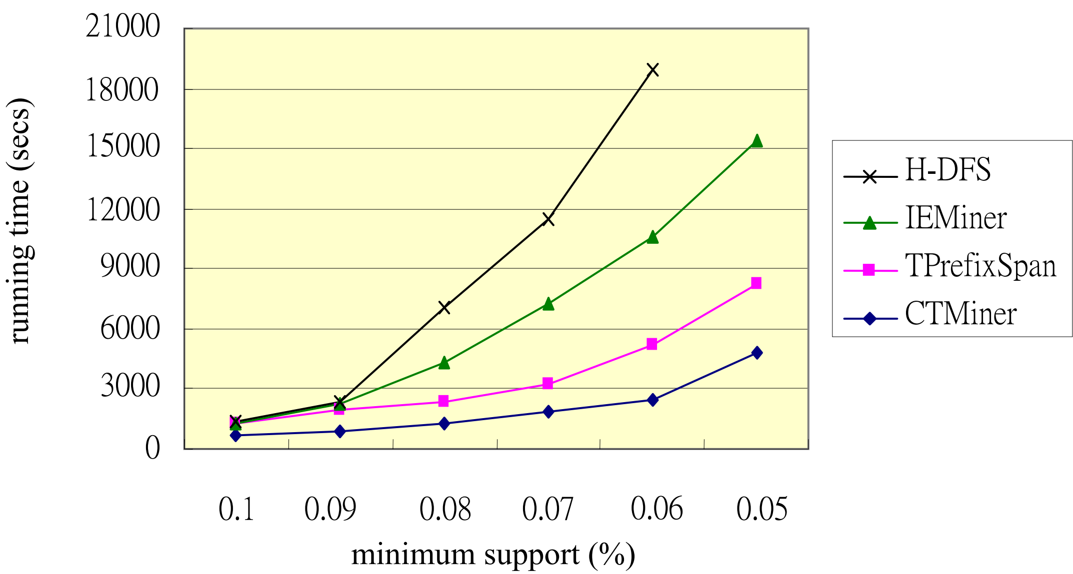


Figure 6-13 Experimental result of the CTMiner algorithm with varying minimal supports on real dataset.

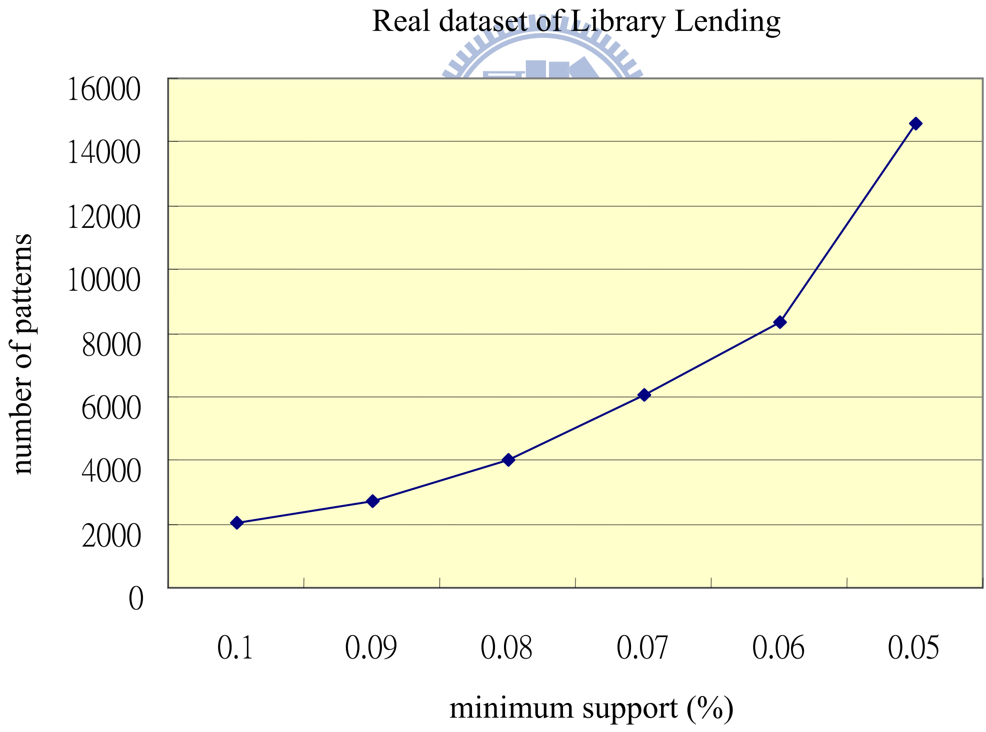


Figure 6-14 The number of generated frequent patterns with varying minimum support.

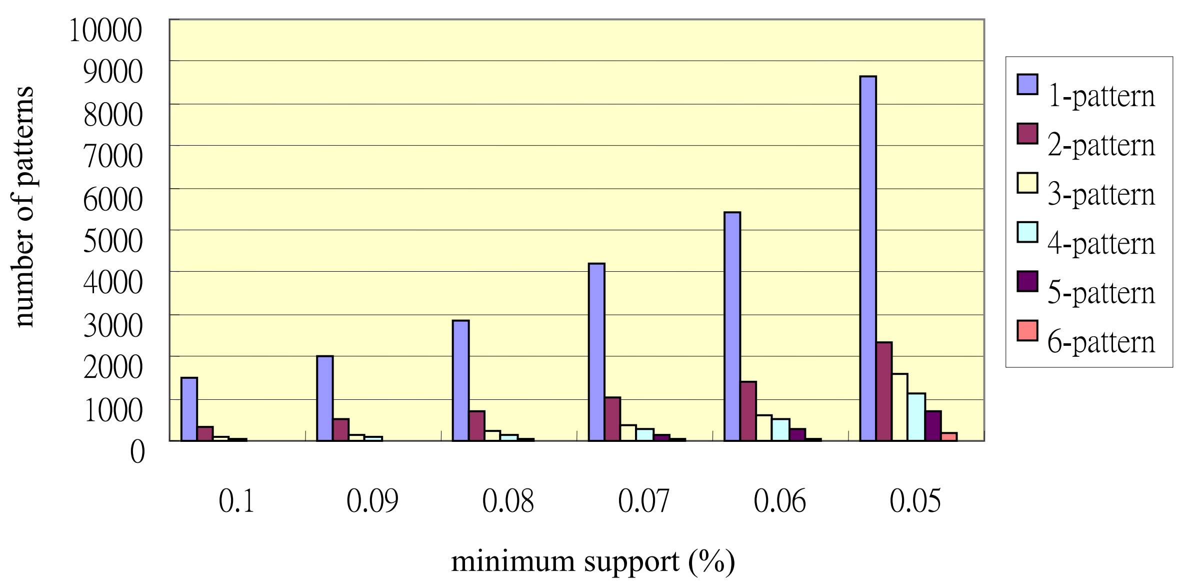


Figure 6-15 The pattern length distribution of pattern length of real dataset with varying minimum support.

The experimental results show that performing the CTMiner algorithm in synthetic data and real data is consistent with our expectation. The CTMiner algorithm exhibits the scalability and efficiency in the experiments.

Chapter 7

Conclusion and Future works

In this thesis, we observe and analyze the drawbacks of different proposed temporal representations based on the complex temporal relation among interval-based events. Then an unambiguous, scalable and efficient coincidence representation based on the concept of coincidence-slice is proposed to address the problem of the complex relationship among events which causes the inefficiency in temporal mining algorithms. Base on the coincidence representation, we also develop a pattern growth-based algorithm called CTMiner by borrowing the concept of PrefixSpan algorithm without candidate generation. According to the characteristics of coincidence representation, we also propose three pruning strategies to reduce the search space and avoid meaningless processing. Hence, the performance of our proposed algorithm CTMiner is improved. To comprehend a coincidence pattern, we discover all the relations in a pattern and present the relations by relation list representation. Experiments on synthetic datasets and real world datasets of library lending demonstrate the efficiency and scalability of our proposed algorithm.

Many extended researches based on interval-based events can be developed by using CTMiner algorithm such as mining partial orders of temporal pattern and closed patterns, maximal patterns, incremental mining and classification and so on. The notation of mining partial orders of temporal patterns has been introduced in [24] and the interesting approach has been recently proposed for closed sequential patterns in [25]. Many real life sequence databases grow incrementally. It is undesirable to mine sequential patterns from scratch each time when a small set of sequences grow, or when some new sequences are added into the database. The incremental mining algorithm has been proposed in [26]. However, these methods again assume that the events are instantaneous. The proposed algorithm provides the

opportunity to design more efficient algorithms in extend researches.

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] 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.

[3] R. Srikant and R. Agrawal. “Mining Sequential patterns: Generalizations and Performance Improvements,” Proceedings of 5th International Conference on Extended Database Technology (EDBT’96), 1996.

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

[5] J. Pei, J. Han, B. Mortazavi-Asl, H. Pito, Q. Chen, U. Dayal, and M.-C. Hsu, “PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth,” Proceedings of 17th International Conference on Data Engineering. (ICDE ’01), pp. 215-224, 2001.

[6] M.Y. Lin and S.Y. Lee. “Fast discovery of sequential patterns by memory indexing,” Proceedings of 4th International Conference on Data Warehousing and Knowledge Discovery (DaWaK’02), pp. 227-237, 2002.

[7] J.F Allen. “Maintaining Knowledge about Temporal Intervals,” Communications of ACM, vol.26, issue 11, pp.832-843, 1983.

[8] 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.

[9] F. Hoppner. “Discovery of Temporal Patterns: Learning Rules about the Qualitative Behaviour of Time Series,” European Conference on Principles of Data Mining and Knowledge Discovery (PKDD’01), vol. 2168, pp. 192-203, 2001

[10] P. Papapetrou, G. Kollios, S. Sclaroff, and D. Gunopulos, “Discovering frequent arrangements of temporal intervals,” International Conference on Data Mining (ICDM’05), pp. 647-661, 2005.

[11] S. Wu and Y. Chen. “Mining Nonambiguous Temporal Patterns for Interval-Based Events,” IEEE Transactions on Knowledge and Data Engineering (TKDE’07), vol.19, issue 6, pp. 742-758, 2007.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] M.-S. Chen, J. Han, and P.S. Yu, “Data Mining: An Overview from a Database Perspective,” IEEE Transactions on Knowledge and Data Engineering, vol. 8, no. 6, pp. 866-883, 1996.

[18] J. Han and M. Kamber, “Data Mining: Concepts and Techniques.” Academic Press, 2001.

[19] H. Mannila, H. Toivonen, and I. Verkamo, “Discovery of frequent episodes in event sequences,” ACM Special Interest Group on Knowledge Discovery and Data Mining, SIGKDD, 1995.

[20] T.B. Ho, T.D. Nguyen, S. Kawasaki, S.Q. Le, D.D. Nguyen, H. Yokoi, and K. Takabayashi. “Mining hepatitis data with temporal abstraction,” Proceedings of the 9th ACM SIGMOD international Conference on Knowledge Discovery and Data Mining (SIGKDD’03), pp. 369-377, 2003.

[21] C. Antunes and A. L. Oliveria, “Generalization of pattern-growth methods for sequential pattern mining with gap constraints,” Machine Learning and Data Mining in Pattern Recognition, vol. 2734, pp.239-251, 2003.

[22] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick. “Sequential pattern mining using a bitmap representation,” Proceedings of the 8th ACM SIGKDD international Conference on Knowledge Discovery and Data Mining (SIGKDD’02), pp. 429-435, 2002.

[23] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu, “FreeSpan:

Frequent Pattern-Projected Sequential Pattern Mining," Proceedings of the 6th ACM SIGKDD International Conference on Data Engineering. (ICDE'01), pp. 215-224, 2001.

[24] Mannila H and Toivonen H, "Discovery generalized episodes using minimal occurrences", Proceedings of ACM SIGMOD, pp. 146-151, 1996.

[25] Casas-Garriga G, "Summarizing sequential data with closed partial orders," Proceedings of SDM, 2004.

[26] H. Cheng, X. Yan and J. Han, "IncSpan: incremental mining of sequential patterns in large database," Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp.527-232, 2004.

[27] K. Gouda, M. J. Zaki, "Efficient Mining Maximal frequent itemsets," International Conference on Data Mining (ICDM'01), pp. 163, 2001.