逢甲大學

資訊工程學系碩士班

碩士論文

具時間限制之高效率序列樣式
探勘演算法

Efficient Algorithms for Mining Sequential Patterns with Time Constraints

指導教授：林明言

研究生：張家汶

中華民國九十五年七月

致謝

在七百多個日子後，我很高興我能在這裡寫我碩論的致謝。碩士班的歷練，讓我自己又邁向另一個層次，但這絕非僅僅是我自己的努力。在此，我要由衷的感謝我的指導教授林明言老師，感謝老師在研究上的啟發與嚴格的要求;包容我粗心所犯的錯誤，耐心的給予訂正。由衷的感謝師母薛夙珍老師在研究期間給予寶貴的經驗，提升我研究的品質。老師細心的教導不僅使我的研究更加充實、完善，更讓我體會到做研究的態度。也誠摯的感謝口試委員李素瑛教授與楊東麟教授的建議與鼓勵，使我更有信心去面對未來的挑戰。

當然，必須一提的是當初就讀朝陽碩班的的小琬學姊與嘉鴻學長，因為有妳們為榜樣，才能使我更快速的進入狀況、彌補所缺，提升自己的能力。也少不了一同努力切磋的偉忠學長、聖坤同學、明宏學弟、重義學弟以及實驗室的大家。然而，讓我暫時忘卻沉重研究壓力的是身旁的朋友，感謝不時傾聽我抱怨、訴說辛苦的阿妮與凱婷；一同成長、歡笑、互相幫助的宇翔；到台中後認識的阿鋒、黃婷以及大家。因為有妳們的照顧與陪伴，使我的碩士班充滿了精采的回憶。

言語不能言謝的是我的家人。父母的辛苦、支持才讓我有機會念完碩士班，家人給予我最溫馨的幸福。妳們一直是我不斷努力的動力。

中文摘要

探勘序列樣式是資料探勘領域中的一個重要研究議題。序列樣式探勘可以從一個序列資料庫中找出所有頻繁的子序列樣式。在探勘的過程中，除了使用者指定的最小頻繁門檻值外，若能提供其他條件則可找出更準確的結果。大部分的條件限制能夠從一般方法找出來的序列樣式加以過濾而取出滿足條件的樣式。由於與時間相關的條件限制影響了序列樣式支持度的計算;亦即一個序列樣式必須在資料序列中滿足時間的條件，其支持度的計數才會增加。所以指定時間限制的樣式探勘必須設計有考慮時間的新演算法。本論文提出一個在記憶體中建立時間索引的演算法，稱METISP，來找出有時間限制的序列樣式。演算法考慮的時間條件可以指定最小、最大或相等的時間間隔、可滑動的時間區間與總持續的時間範圍之時間限制。METISP 將資料庫載入記憶體並建立時間的索引以進行有效率的樣式探勘。利用索引集合與樣式增長（pattern-growth）的策略，METISP 不需要產生任何的候選樣式或是子資料庫即可找出符合要求的樣式。這些時間的索引減少搜尋的範圍並加速了項目支持度的計算。此外，本論文也提出一個找尋具時間限制封閉樣式之新演算法，稱CTSP。CTSP藉由一個整合時間限制的雙向封閉檢查的方法來探勘指定最大、最小間隔及可滑動的時間區間的封閉序列樣式。探勘出來的封閉的樣式在呈現給使用者更精簡的結果下仍保有完整的資訊。為評估效能，METISP與CTSP在許多不同的資料庫與時間限制下與知名的GSP與DELISP演算法做比較。眾多實驗的結果顯示了即使當使用者指定了較低的門檻值或是探勘大型的的資料庫，METISP與CTSP都能有較好的執行效率。

關鍵詞：序列樣式，時間限制，記憶體索引，探勘循序樣式，封閉序列樣式

Abstract

Sequential pattern mining is one of the important issues in the research of data mining. The mining is to find out all the frequent sub-sequences in a sequence database. In order to have more accurate results, constraints in addition to the support threshold need to be specified in the mining. Most time-independent constraints can be handled, without modifying the fundamental mining algorithm, by retrieving qualified patterns from the discovered ones. Time-constraints, however, cannot be managed by retrieving patterns because the support computation of patterns must validate the time attributes for every data sequence in the mining process. Therefore, a memory time-indexing approach, called METISP, is proposed in this thesis to discover sequential patterns with time constraints including minimum gap, maximum gap, exact gaps, sliding window, and duration. METISP scans the database into memory and constructs time-index sets for effective processing. Utilizing the index sets and the pattern-growth strategy, METISP efficiently mines the desired patterns without generating any candidate or sub-database. The index sets narrow down the search space to the sets of designated in-memory data sequences, and speed up the counting to the indicated ranges of potential items. In addition, a novel algorithm, called CTSP, is also proposed for mining closed sequential patterns with time constraints. The closed patterns preserve the complete information with more compact representations. We have evaluated METISP algorithm and CTSP algorithm with the well-known GSP algorithm and the DELISP algorithm for various datasets and constraints. The comprehensive experiments show that METISP and CTSP both have better efficiency, even with low support thresholds and very large databases.

Keywords: sequential patterns, time constraint, memory index, sequence mining, closed sequential pattern

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

The development and applications of computers and automation techniques enable huge number of data to be captured and stored. Discovering the previously unknown and potential useful knowledge from these data is becoming an important research topic. Many problems in data mining [12], such as sequential pattern mining [7, 8, 10, 13, 28, 32], association rule mining [1, 14], clustering, classification and so on thus have attracted much attention of researches. This thesis focuses on the study on mining sequential patterns with time constraints.

1.1 Background

The problem of mining sequential patterns is becoming an active research topic for its wide applications, including market basket analysis, traversal pattern finding, medical treatment diagnosis, natural disaster prediction, DNA sequence analysis, and so on. Give an example of web usage mining, those page-views clicked by the same visitor in a web site can be stored as a data sequence and represented as a form of sessions of items-sets. A discovered frequent pattern may be <(a)(c, e)>, which indicates many people would browse a.html in a session, and browse pages c.html and e.html in the consequent session. Another typical example is to find all the frequent sub-sequences in a retail database of customer purchasing sequences. The mining results disclose not only the frequent items bought together, but also the sequences of frequently appeared item-sets. Accordingly, the complexity of sequential pattern mining is more complex than association rule mining.

Improving the efficiency of sequence mining algorithms has been the focus of many studies while increasing the accuracy of mining results tends to be more desirable in practice. Though sequential patterns indicate the relationships among the frequently occurring item-set sequences, the huge number of patterns might still discourage users to proper actions because many useless patterns are included without precise constraints. Various constraints, such as item, length, super-pattern, duration, and gaps, can be specified to find the interesting patterns. Many constraints can be handled by a post-processing on the result of sequential pattern mining without constraints. For example, the result of mining with constraint “item = NOTEBOOK” can be obtained by retrieving the discovered patterns which contains the NOTEBOOK item. Nevertheless, time constraints affect the support computation of patterns so that they cannot be handled without adapting time attributes into the mining algorithms.

Time constraint is one of the most important constraints. The issue of mining sequential patterns with time constraints was first addressed in [3]. Three time constraints including minimum gap, maximum gap and sliding time-window are specified to enhance the semantics of sequence discovery. For example, without time constraints, one may find a pattern <(PC, PRINTER)(LCD-PROJECTOR)>, which means most customers could buy LCD-PROJECTOR after purchasing PC and PRINTER. However, the pattern could be insignificant if the time interval between the two transactions is too long. Such patterns could be filtered out if the maximum gap constraint is specified. Similarly, if the time interval between the two transactions is too short, the patterns may be useless for user to react in time. The minimum gap restricts the minimum time difference, and the maximum gap restricts the maximum time difference, between adjacent transactions and reinforces the discovery.

Essentially, a data sequence is defined to support a pattern if each element of the pattern is contained in an individual transaction of the data sequence. Sometimes users may not mind whether the items of an element are bought in the same transaction, or in adjoining transactions if the adjoining transactions occur within a specified time interval. For instance, given a sliding time-window of 4, a data sequence $<_{5}$(PC) $_{7}$(DVD-DRIVE) $_{10}$(DVD-R)> can support pattern <(PC, DVD-DRIVE)(DVD-R)> or <(PC)(DVD-DRIVE, DVD-R)> because the total time-span is within the sliding time-window. Sliding time-window constraint relaxes the definition of an element and broadens the applications of sequential patterns.

In addition to the three time constraints, duration and exact gap constraints are usually specified for finding actionable patterns. Duration indicates the maximum total time-span allowed for a pattern. Users would not obtain a pattern having transactions conducted over one year if the duration of 365 days is given. Exact gap can be used to find patterns, within which adjacent transactions occur exactly the specified time difference. We say that the sequential patterns discovered by specifying additional time constraints as time-constrained sequential patterns.

Not only time-constrained patterns [3, 19, 21, 22, 24], the issue of sequential pattern mining has been extended to various topics such as maximal sequential pattern mining [15, 20], closed sequential pattern mining [30, 31], top-k sequential pattern mining [27] and so on. A sequential pattern is maximal if it is not a subsequence of another frequent sequence. It is closed if no frequent super-sequence has the same support. For example, if <(a)>:3, <c>:3, <(e)>:2, <(a)(c)>:2, <(a)(e)>:2, (c, e):2, <(a)(c, e)>:2 are all mined sequential patterns with their respective supports. Accordingly, the closed patterns are <(a)>:3, <(c)>:3, <(a)(c, e)>:2, and the maximal pattern is <(a)(c, e)>:2. The problems for mining maximal sequential patterns and mining closed sequential patterns are to find all frequent maximal sequences and frequent closed sequences from databases, respectively. Lastly, the top-k sequential pattern mining is to discover the k most frequent sequential patterns without specifying a support threshold.

1.2 Research Motivation

For many applications, time constraint is particularly important for finding desired results so that users can make better decisions. For example, in a retail application, a discovered sequential pattern may be <(TV)(Audio, Jukebox)>, which means most customers could buy Audio and Jukebox after purchasing TV, and it can be applied to sales promotion, inventory control, etc.. However, if the time gap between two adjacent transactions in a pattern is long, such as over one year, it is useless for sales promotion. Moreover, if three days is indispensable for the supply of the stock and the time gap between two adjacent transactions in a pattern is too short, it will be useless to react in time.

Additionally, a consumer may forget to buy some goods in a transaction and buy them in the subsequent transaction within two days. A sliding window of two days may allow the two transactions to be handled (combined) as one. For instance, a data sequence <$_7$(TV)$_9$(Audio)$_{11}$(Jukebox)> can support this pattern <(TV)(Audio, Jukebox)>. Moreover, if the total time-span of over one year is useless, the duration between the first transaction and the last transaction should be specified to exclude invalid patterns. Besides, many consumers may have regular purchasing behavior like weekly shopping. Exact gap then can be specified to discover those patterns with occurring at fixed time interval. Therefore, time constraint is significant for more precise results.

Although there are many algorithms dealing with sequential pattern mining [2, 4 5], few handle the mining with the addition of time constraints [3, 19, 34]. The GSP algorithm [3] solves the mentioned constraints, except duration and exact gap, within the Apriori framework. GSP scans the database *k* times to discover patterns having *k* items. The *cSPADE* algorithm [34] extends the vertical mining algorithm *SPADE* [33] to deal with time constraints. The *cSPADE* algorithm checks minimum gap and maximum gap while doing temporal joins. Nevertheless, the huge sets of frequent 2-sequences must be preserved to generate the required classes for the maximum gap constraint. While it is possible for *cSPADE* to handle constraints like minimum/maximum gap by expanding the id-lists and augmenting the join-operations with temporal information, it does not appear feasible to incorporate sliding window. The sliding window constraint was not mentioned in *cSPADE*.

The PrefixSpan algorithm [25] is extended in [26] to push the prefix-monotone constraint into the pattern-growth framework. However, the sliding time-window constraint is neither monotonic nor anti-monotonic. Handling sliding time-window constraint is a non-trivial task of modifying the Prefix-growth algorithm since the prefix-monotonic feature is no longer valid. The DELISP algorithm [19], fully functionally equivalent to GSP on time constraint issues, solves the problem within pattern-growth framework. However, the size of the projected databases might accumulate to several times the size of the original database. As the main memory size increase, many small or medium sized databases could be completely loaded into memory. Motivated by such a scenario, the study aims to utilize memory for efficient mining of time-constrained patterns.

When mining is conducted in a dense database or the minimum support is very low, the length of discovered sequential patterns may be very long and the number of patterns may be large. On the basis of anti-monotone property, we know that given a frequent sequence, all of its subsequences would be frequent. Thus, it may be redundant to return all the frequent sequences to the user. In contrast to the complete set of sequential patterns, the maximal patterns give the smallest set of frequent patterns but some frequency counts are lost. The closed sequential patterns not only keep the complete information but also give a compact set of mining results. Recently, mining closed patterns rather than complete patterns has attracted many researches is more helpful to users.

Several algorithms [27, 30, 31] were introduced to deal with the problem of mining closed sequential patterns. The CloSpan [31] and BIDE [30] algorithms discover patterns using pattern-growth methodology. The CloSpan algorithm needs to keep potential closed patterns and verify them afterward. The BIDE algorithm was presented to directly output the closed patterns. However, to the best of our knowledge, no algorithm has been proposed to solving the problem of mining closed sequential patterns with time constraints. This difficult but useful issue also motivates the study of the thesis.

1.3 Research Objectives

In previous studies, most algorithms handle only some of the above time constraints, new algorithms which can handle all these constraints would be required. Mining with time constraints will inevitably demands more time than that without time constraints, improving the efficiency thus because an important issue.

Therefore, a new algorithm called METISP (MEmory Time-Indexing for Sequential Pattern mining) is proposed in this thesis for handling time constraints on sequential patterns. METISP deals with minimum/maximum/exact gap, sliding time-window and duration constraints by memory time-index sets within the pattern-growth framework [25]. The objective of METISP aims to efficiently mine the complete set of time-constrained sequential patterns, utilizing the memory.

Furthermore, we also formulate the problem of closed sequential patterns with time constraints. We propose a new algorithm called CTSP (Closed Time-constrained Sequential Pattern mining) for mining closed sequential patterns with minimum gap/maximum gap/sliding window constraints. The objective of CTSP is to discover the compact set of closed sequential pattern with memory indexing and improved closure checking.

1.4 Organization of this Thesis

The rest of the thesis is organized as follows. The related work is described in Chapter 2. Chapter 3 and Chapter 4 report the METISP algorithm for mining time-constrained sequential patterns and CTSP algorithm for mining closed time-constrained sequential patterns, respectively. Chapter 5 summaries the study results and addresses the future work.

Chapter 2 Related Work

In general, algorithms for mining sequential patterns with time constraints can be categorized into: Apriori-like approaches such as GSP [3] and pattern-growth approaches such as DELISP [19]. Table 2.1 shows a summary of the two categories. The Apriori-like approaches suffer from testing a large number of candidates and scanning multiple databases for long sequential patterns. Current pattern-growth approaches project the original database into many sub-databases recursively and grow patterns efficiently. In Section 2.1, we describe the GSP algorithm [3]. The SPADE [33] and cSPADE [34] algorithms are described in Section 2.2. The PrefixSpan [25] and prefix-growth [26] algorithms are addressed in Section 2.3. DELISP algorithm [19] is presented in Section 2.4. Finally, Sections 2.5, 2.6 and 2.7 report CloSpan [31], BIDE [30] and Par-CSP [11] algorithms for mining closed sequential patterns, respectively.

Table 2.1 A summary of algorithms with time constraints

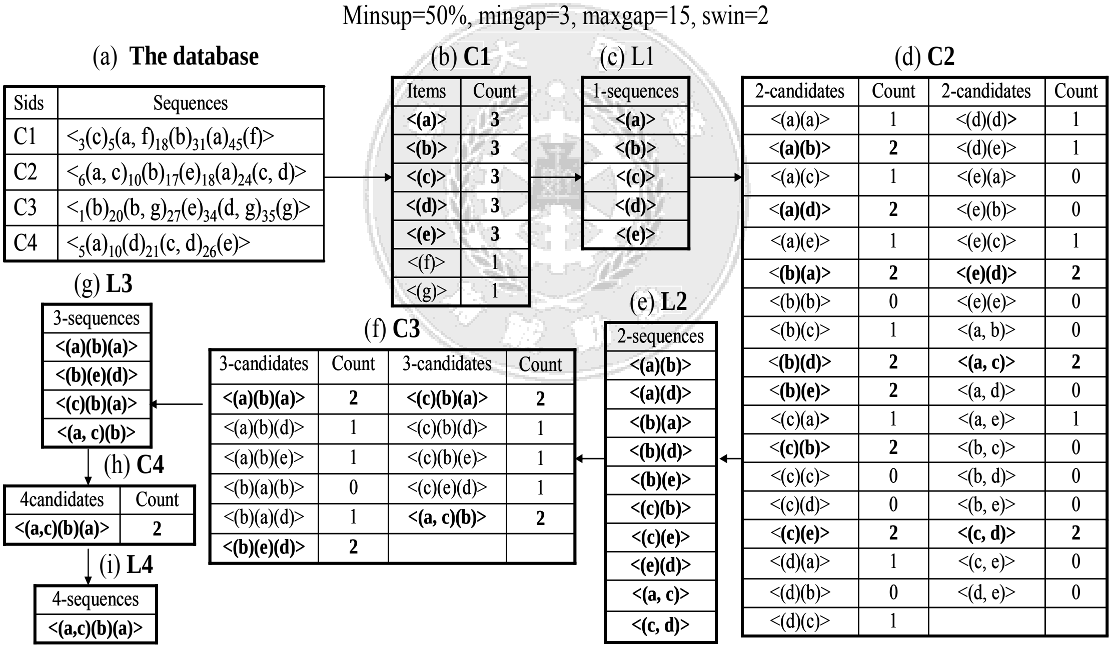
Framework	Algorithm	Min. gap	Max. gap	Exact gap	Sliding window	Duration	Year
Apriori-like approach	GSP	✓	✓		✓		1996
	PSP	✓	✓		✓		1998
	cSPADE	✓	✓	✓		✓	2000
	CCSM	✓	✓	✓			2004
Pattern-growth approach	prefix-growth	✓	✓	✓		✓	2002
	DELISP	✓	✓		✓		2002

2.1 GSP Algorithm

The problem of mining sequential pattern in customer database is first introduced in [2] by Agrawal and Srikant. The AprioriAll algorithm in [2] applies the idea of Apriori algorithm [1] which mines frequent itemsets in a transactions database to find sequential patterns. In the subsequent studies, the GSP (**G**eneralized **S**equential **P**atterns) algorithm [3] improves the ApririAll algorithm and generalizes the problem to involve time constraints and taxonomies. The major spirit is that GSP adopts a candidate-generation-and-test methodology to mine generalized sequential patterns. The concept of contiguous subsequences is introduced to assist candidate generations with time constraints including minimum gap, maximum gap, and sliding window in the framework. Any two frequent k-sequences found in the k step are used to join a new candidate in the (k+1) step when they share the common (k-1)-subsequence after one removes the first item and the other removes the last item.

GSP algorithm first scans the database once to find all frequent items. These frequent k-sequences are treated as the seed set. Then (k+1)-candidate sequences are generated by self-joins of the seed set. In the prune phase, these candidates can been further deleted if they have any infrequent contiguous (k-1)-subsequences. A hash-tree structure is then constructed to store all candidates for speeding up support counting. In order to check whether a candidate satisfies the time constraints, GSP switches forward phase and backward phase in each sequence. The level-wise mining is put into practice until no new candidates generated or no sequential patterns found. Therefore, the GSP algorithm needs to scan the database p times if the maximum length of the candidates is p.

Figure 2.1 shows an example of GSP algorithm. Given a database in Figure 2.1(a), the minsup=50%, mingap=3, maxgap=15 and swin=2. Figure 2.1(c) denotes all frequent items discovered. Figure 2.1(d) means the 2-candidates generated by self-joins of any two items in Figure 2.1(c). Then the supports of the 2-candidates are verified by checking the time constraints. In Figure 2.1(e), any two 2-sequences which share the same 1-subsequence when one removes the first item and the other removes last item will generate a new 3-candidate. For example, <(a)(b)> and <(b)(e)> join a candidate <(a)(b)(e)>, because they share the same 1-subsequence <(b)>. Although, <(a, c)(e)> is generated by <(a, c)> and <(c)(e)> but it is pruned because it contains a no-frequent contiguous 2-subsequence <(a)(e)>. Using the join rule, all of 3-candidates are found in Figure 2.1(f). GSP verifies the supports of all candidates and the remainder results are displayed in Figures 2.1(g), (h) and (i).



Minsup=50%, mingap=3, maxgap=15, swin=2

(a) **The database**

Sids	Sequences
C1	$<_3(c)_5(a, f)_{18}(b)_{31}(a)_{45}(f)>$
C2	$<_6(a, c)_{10}(b)_{17}(e)_{18}(a)_{24}(c, d)>$
C3	$<_1(b)_{20}(b, g)_{27}(e)_{34}(d, g)_{35}(g)>$
C4	$<_5(a)_{10}(d)_{21}(c, d)_{26}(e)>$

(b) **C1**

Items	Count
<(a)>	**3**
<(b)>	**3**
<(c)>	**3**
<(d)>	**3**
<(e)>	**3**
<(f)>	1
<(g)>	1

(c) L1

1-sequences
<(a)>
<(b)>
<(c)>
<(d)>
<(e)>

(d) **C2**

2-candidates	Count	2-candidates	Count
<(a)(a)>	1	<(d)(d)>	1
<(a)(b)>	**2**	<(d)(e)>	1
<(a)(c)>	1	<(e)(a)>	0
<(a)(d)>	**2**	<(e)(b)>	0
<(a)(e)>	1	<(e)(c)>	1
<(b)(a)>	**2**	**<(e)(d)>**	**2**
<(b)(b)>	0	<(e)(e)>	0
<(b)(c)>	1	<(a, b)>	0
<(b)(d)>	**2**	**<(a, c)>**	**2**
<(b)(e)>	**2**	<(a, d)>	0
<(c)(a)>	1	<(a, e)>	1
<(c)(b)>	**2**	<(b, c)>	0
<(c)(c)>	0	<(b, d)>	0
<(c)(d)>	0	<(b, e)>	0
<(c)(e)>	**2**	**<(c, d)>**	**2**
<(d)(a)>	1	<(c, e)>	0
<(d)(b)>	0	<(d, e)>	0
<(d)(c)>	1		

(e) **L2**

2-sequences
<(a)(b)>
<(a)(d)>
<(b)(a)>
<(b)(d)>
<(b)(e)>
<(c)(b)>
<(c)(e)>
<(e)(d)>
<(a, c)>
<(c, d)>

(f) **C3**

3-candidates	Count	3-candidates	Count
<(a)(b)(a)>	**2**	**<(c)(b)(a)>**	**2**
<(a)(b)(d)>	1	<(c)(b)(d)>	1
<(a)(b)(e)>	1	<(c)(b)(e)>	1
<(b)(a)(b)>	0	<(c)(e)(d)>	1
<(b)(a)(d)>	1	**<(a, c)(b)>**	**2**
<(b)(e)(d)>	**2**		

(g) **L3**

3-sequences
<(a)(b)(a)>
<(b)(e)(d)>
<(c)(b)(a)>
<(a, c)(b)>

(h) **C4**

4candidates	Count
<(a,c)(b)(a)>	**2**

(i) **L4**

4-sequences
<(a,c)(b)(a)>

Figure 2.1 An example of GSP algorithm

2.2 SPADE and cSPADE Algorithms

The SPADE (**S**equential **P**attern **D**iscovery using **E**quivalence classes) algorithm [33] uses a vertical id-list database format for mining sequential patterns. A lattice-partition approach based on prefix classes is applied to decompose the original problem to small sub-problems when the main memory is insufficient to fit all id-lists. SPADE performs temporal id-list joins of k-sequences to generate (k+1)-sequences. After sequence pruning, the frequency of a sequence can be obtained from counting the number of distinct *sid* values in its id-list. Differently, SPADE uses a vertical-to-horizontal database transformation and two-dimensional array to discover frequent 2-seuences because of the expensive cost in enumerating all combinations of 2-candidates. Beside, two search strategies, breadth-first and depth-first searches, are proposed for sequence enumerations. Hence, SPADE reduces largely the cost of scanning a database.

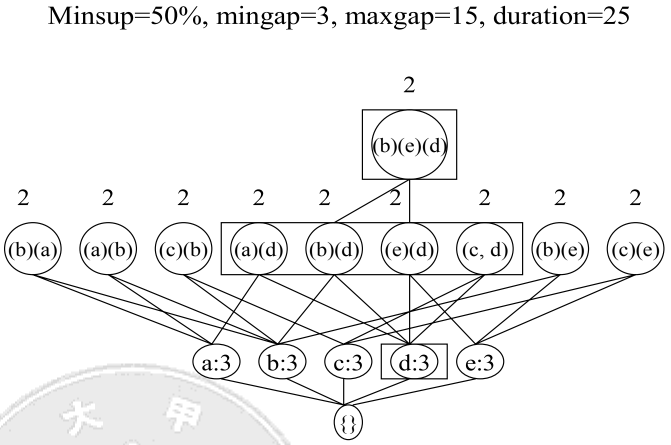
The cSPADE (**c**onstrained **S**equential **P**attern **D**iscovery using **E**quivalence classes) algorithm [34] is a modification of SPADE algorithm by integrating synthetic constraints, including minimum/maximum gap, window size, length, width constraints and others into mining process. In [34], the window size stands for duration time of patterns but not sliding window. If any two k-sequences share common (k-1)-suffix, they are in the same class and the temporal join can be done. Moreover, cSPADE examines time constraints at the same time. Minimum gap examination can be applied easily into temporal id-list joins without modifying much code of SPADE algorithm. However, if a user specifies a maximum gap which is not class preserving, cSPADE needs to retain all id-lists of 2-sequences to join others id-lists of different length sequences in order to ensure the discovered patterns are correct. To consider the window size, cSPADE adds a column additionally, *diff*, to original id-lists to indicate the total time-spans of a pattern.

Figure 2.2 shows an example of a vertical id-list database and its class lattice. Figure 2.3 give examples of temporal id-list joins. In Figure 2.2(b), the sequences <(b)(d)>, <(e)(d)> are in the same class and intersect a frequent 3-sequence <(b)(e)(d)>. In Figure 2.3(a), <(a)> appears before <(b)> in the same data sequence

C1 so that (Sid, Eid, Diff)=(C1, 5, 13) is recorded in the id-list of <(a)(b)>. The Eid 5 and diff 13 stand for the time (a) occurs and the total time-span of <(a)(b)>, respectively. It is similar in Figure 2.3(b), <(c)> and <d> occur at the same time 24 in data sequence C2 so that (Sid, Eid, Diff)=(C2, 24, 0) is recorded in the id-list of <(c, d)>.

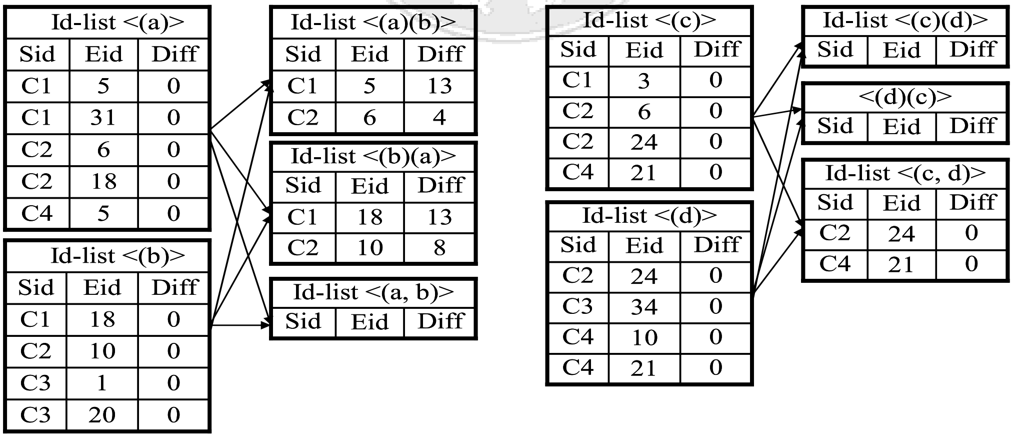
Items	Sid	Eid	Items	Sid	Eid
a	C1	5	d	C2	24
	C1	31		C3	34
	C2	6		C4	10
	C2	18		C4	21
	C4	5	e	C2	17
b	C1	18		C3	27
	C2	10		C4	26
	C3	1	f	C1	5
	C3	20		C1	45
c	C1	3	g	C3	20
	C2	6		C3	34
	C2	24		C3	35
	C4	21			

(a) A vertical id-list database



(b) A class lattice

Figure 2.2 An example of a vertical id-list database and its class lattice



Id-list <(a)>

Sid	Eid	Diff
C1	5	0
C1	31	0
C2	6	0
C2	18	0
C4	5	0

Id-list <(b)>

Sid	Eid	Diff
C1	18	0
C2	10	0
C3	1	0
C3	20	0

Id-list <(a)(b)>

Sid	Eid	Diff
C1	5	13
C2	6	4

Id-list <(b)(a)>

Sid	Eid	Diff
C1	18	13
C2	10	8

Id-list <(a, b)>

Sid	Eid	Diff

Id-list <(c)>

Sid	Eid	Diff
C1	3	0
C2	6	0
C2	24	0
C4	21	0

Id-list <(d)>

Sid	Eid	Diff
C2	24	0
C3	34	0
C4	10	0
C4	21	0

Id-list <(c)(d)>

Sid	Eid	Diff

<(d)(c)>

Sid	Eid	Diff

Id-list <(c, d)>

Sid	Eid	Diff
C2	24	0
C4	21	0

(a) Temporal joins of <(a)> and <(b)>　　(b) Temporal joins of <(c)> and <(d)>

Suffix joins on id-lists

Figure 2.3 Examples of temporal id-list joins

2.3 PrefixSpan and prefix-growth Algorithms

PrefixSpan (**Prefix**-projected **S**equential **pa**tter**n** mining) algorithm [25] is an effective pattern-growth methodology for mining long sequential patterns. Prefixspan projects original database into several small databases which are based on frequent prefixes. For each projected database, only the essential suffixes of sequences are retained. At the same time, PrefixSpan removes infrequent items and useless sequences. Instead of mining whole database, PrefixSpan only scans the sub-database to find all local frequent items to grow the prefix. PrefixSpan algorithm outputs different length of patterns recursively until no new patterns discovered. The bi-level projection scheme is proposed to reduce the size and the number of projected databases by building an S-matrix. When the size of main memory is enough to load a projected database into memory, a pseudo projection technique is used to reduce the cost of projecting databases.

In general, many common constraints can be classified several categories such as anti-monotonic, monotonic, succinct and others based on their inherent property. In [26], a new category, prefix-monotone, is proposed to contain those constraints belong to anti-monotonic or monotonic. Moreover, the prefix-growth algorithm is proposed to push prefix-monotone constraints, succinct and other constraints like regular expression into mining process without modifying the framework of pattern-growth too much. Among those constraints, time constraints, gap and duration constraints are indicated but the power sliding window is not involved in.

The prefix-growth algorithm first finds all frequent items and checks them by specifying constraints. Similar to PrefixSapn, after removing infrequent items and useless sequences, only those items which satisfy specifying constraints can be used as prefixes to project sub-databases. The algorithm further mines each sub-database to find local frequent items. Before forming a new growth pattern, prefix-growth must examine the new pattern using the specified constraints. If the pattern satisfies constraints, algorithm outputs it and calls subroutine to mine recursively. However, with time constraints, prefix-growth must check the time relation with time attributes of elements in sequences to decide the supports of patterns. The explicit method for handling time constraints is not reported in [26].

Figure 2.4 shows an example of prefix-growth algorithm. Total five frequent items are found after first scanning database. The infrequent items f and g are removed in later projected sub-databases. Take <a>-projected database for example, local frequent items (a), (b), (c), (d), and (e) are derived but only patterns <(a)(b)> and <(a)(d)> hold time constraints. Hence, <(a)(b)> and <(a)(d)> are output and treated as prefixes to project <(a)(b)> and <(a)(d)>-projected databases.

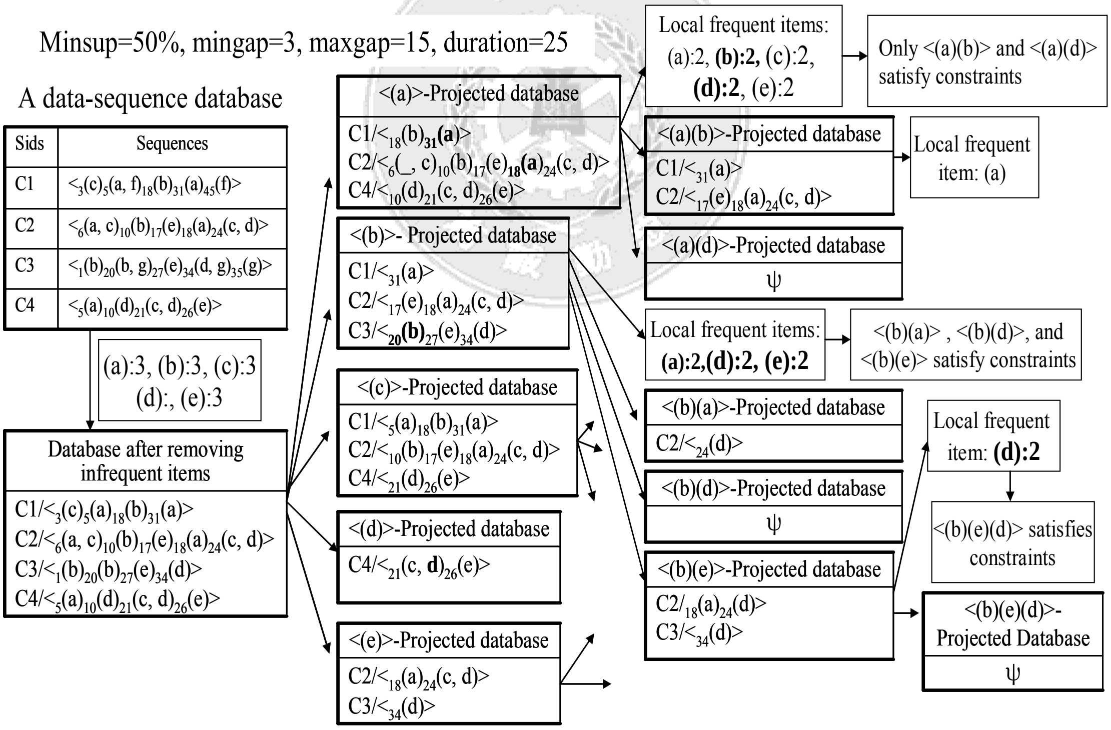

Minsup=50%, mingap=3, maxgap=15, duration=25
A data-sequence database
Sids
Sequences
C1
<$_3$(c)$_5$(a, f)$_{18}$(b)$_{31}$(a)$_{45}$(f)>
C2
<$_6$(a, c)$_{10}$(b)$_{17}$(e)$_{18}$(a)$_{24}$(c, d)>
C3
<$_1$(b)$_{20}$(b, g)$_{27}$(e)$_{34}$(d, g)$_{35}$(g)>
C4
<$_5$(a)$_{10}$(d)$_{21}$(c, d)$_{26}$(e)>
(a):3, (b):3, (c):3
(d):, (e):3
Database after removing infrequent items
C1/<$_3$(c)$_5$(a)$_{18}$(b)$_{31}$(a)>
C2/<$_6$(a, c)$_{10}$(b)$_{17}$(e)$_{18}$(a)$_{24}$(c, d)>
C3/<$_1$(b)$_{20}$(b)$_{27}$(e)$_{34}$(d)>
C4/<$_5$(a)$_{10}$(d)$_{21}$(c, d)$_{26}$(e)>
<(a)>-Projected database
C1/<$_{18}$(b)$_{31}$(a)>
C2/<$_6$(_, c)$_{10}$(b)$_{17}$(e)$_{18}$(a)$_{24}$(c, d)>
C4/<$_{10}$(d)$_{21}$(c, d)$_{26}$(e)>
<(b)>- Projected database
C1/<$_{31}$(a)>
C2/<$_{17}$(e)$_{18}$(a)$_{24}$(c, d)>
C3/<$_{20}$(b)$_{27}$(e)$_{34}$(d)>
<(c)>-Projected database
C1/<$_5$(a)$_{18}$(b)$_{31}$(a)>
C2/<$_{10}$(b)$_{17}$(e)$_{18}$(a)$_{24}$(c, d)>
C4/<$_{21}$(d)$_{26}$(e)>
<(d)>-Projected database
C4/<$_{21}$(c, d)$_{26}$(e)>
<(e)>-Projected database
C2/<$_{18}$(a)$_{24}$(c, d)>
C3/<$_{34}$(d)>
Local frequent items:
(a):2, (b):2, (c):2,
(d):2, (e):2
Only <(a)(b)> and <(a)(d)> satisfy constraints
<(a)(b)>-Projected database
C1/<$_{31}$(a)>
C2/<$_{17}$(e)$_{18}$(a)$_{24}$(c, d)>
Local frequent item: (a)
<(a)(d)>-Projected database
ψ
Local frequent items:
(a):2,(d):2, (e):2
<(b)(a)> , <(b)(d)>, and <(b)(e)> satisfy constraints
<(b)(a)>-Projected database
C2/<$_{24}$(d)>
<(b)(d)>-Projected database
ψ
<(b)(e)>-Projected database
C2/$_{18}$(a)$_{24}$(d)>
C3/<$_{34}$(d)>
Local frequent item: (d):2
<(b)(e)(d)> satisfies constraints
<(b)(e)(d)>-
Projected Database
ψ

Figure 2.4 An example of prefix-growth algorithm

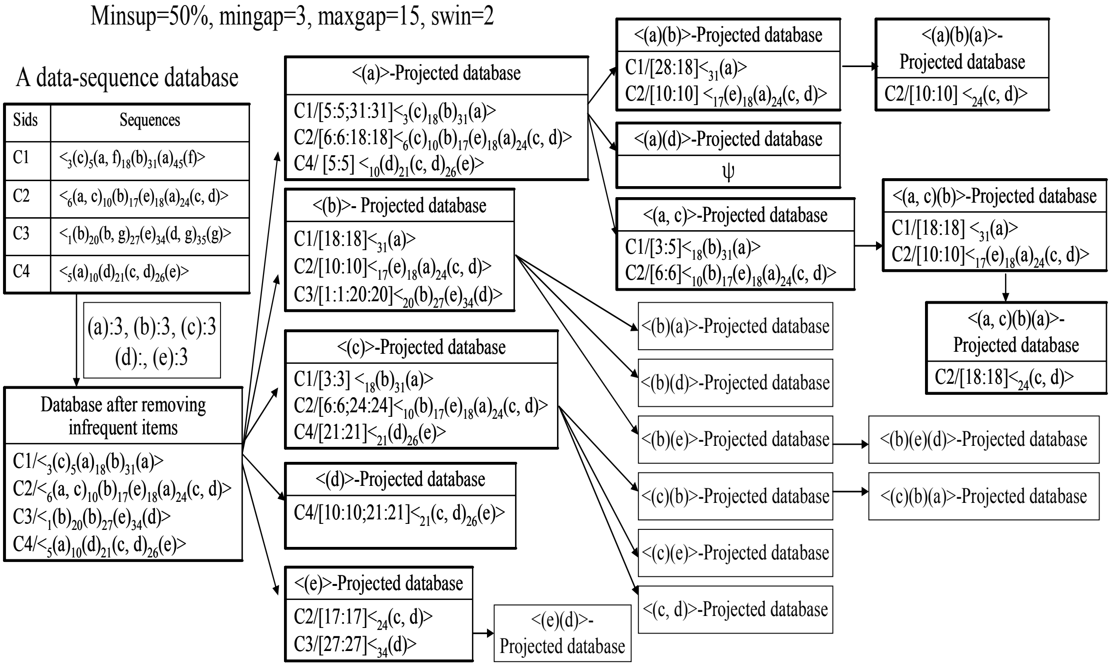
2.4 DELISP Algorithm

DELISP (**DELI**mited **S**equential **P**attern mining) algorithm [19] is similar to the Prefixspan algorithm but DELISP handles sliding window and minimum/maximum gap constraints. DELISP projects databases by windowed and bounded techniques and grows patterns effectively by delimited growth. Time constraints limit the search space and decrease number of the items which need to be counted.

For each new pattern discovered, DELISP scans each data sequence containing its prefix and records its timestamps involving start-time and end-time in each data sequence containing it to form its tag lists. Here, each start-time and end-time in a tag list represents the start and end time of the elements which contains the last element of the pattern in a data sequence. Then, bounded-projection and windowed-projection use the tag lists to filter out some elements or items which are impossible to contribute the supports of stems of type-1 and type-2 patterns. DELISP also removes infrequent items afterward, when the sub-databases are projected.

For each projected database, DELISP separately counts the supports of potential type-1 stems and type-2 stems by applying the delimited-growth technique. Hence, DELISP can find all stems to grow type1 and type-2 patterns quickly. Using the same steps, DELISP discovers all time-constrained sequential patterns recursively.

Figure 2.5 gives an example of DELISP algorithm. Take <(a)>-projected database for instance, item a appears at time 5 and time 31 in data sequence C1 and the tag list is marked as [5:5, 31:31]. The item c occurs before item a in C1, but it satisfies sliding window constraint using the tag list so that it is projected into sub-databases. Applying the delimited-growth technique, a type-1 stem (b) and a type-2 stem (c) are found to from patterns <(a)(b)> and <(a, c)>. Utilizing the same steps, all desired patterns can be found later.



Sids	Sequences
C1	$<_{3}(c)_{5}(a, f)_{18}(b)_{31}(a)_{45}(f)>$
C2	$<_{6}(a, c)_{10}(b)_{17}(e)_{18}(a)_{24}(c, d)>$
C3	$<_{1}(b)_{20}(b, g)_{27}(e)_{34}(d, g)_{35}(g)>$
C4	$<_{5}(a)_{10}(d)_{21}(c, d)_{26}(e)>$

Minsup=50%, mingap=3, maxgap=15, swin=2
A data-sequence database
(a):3, (b):3, (c):3
(d):, (e):3
Database after removing infrequent items
C1/$<_{3}(c)_{5}(a)_{18}(b)_{31}(a)>$
C2/$<_{6}(a, c)_{10}(b)_{17}(e)_{18}(a)_{24}(c, d)>$
C3/$<_{1}(b)_{20}(b)_{27}(e)_{34}(d)>$
C4/$<_{5}(a)_{10}(d)_{21}(c, d)_{26}(e)>$
<(a)>-Projected database
C1/[5:5;31:31]$<_{3}(c)_{18}(b)_{31}(a)>$
C2/[6:6:18:18]$<_{6}(c)_{10}(b)_{17}(e)_{18}(a)_{24}(c, d)>$
C4/ [5:5] $<_{10}(d)_{21}(c, d)_{26}(e)>$
<(b)>- Projected database
C1/[18:18]$<_{31}(a)>$
C2/[10:10]$<_{17}(e)_{18}(a)_{24}(c, d)>$
C3/[1:1:20:20]$<_{20}(b)_{27}(e)_{34}(d)>$
<(c)>-Projected database
C1/[3:3] $<_{18}(b)_{31}(a)>$
C2/[6:6;24:24]$<_{10}(b)_{17}(e)_{18}(a)_{24}(c, d)>$
C4/[21:21]$<_{21}(d)_{26}(e)>$
<(d)>-Projected database
C4/[10:10;21:21]$<_{21}(c, d)_{26}(e)>$
<(e)>-Projected database
C2/[17:17]$<_{24}(c, d)>$
C3/[27:27]$<_{34}(d)>$
<(e)(d)>-Projected database
<(a)(b)>-Projected database
C1/[28:18]$<_{31}(a)>$
C2/[10:10] $<_{17}(e)_{18}(a)_{24}(c, d)>$
<(a)(b)(a)>-Projected database
C2/[10:10] $<_{24}(c, d)>$
<(a)(d)>-Projected database
ψ
<(a, c)>-Projected database
C1/[3:5]$<_{18}(b)_{31}(a)>$
C2/[6:6]$<_{10}(b)_{17}(e)_{18}(a)_{24}(c, d)>$
<(a, c)(b)>-Projected database
C1/[18:18] $<_{31}(a)>$
C2/[10:10]$<_{17}(e)_{18}(a)_{24}(c, d)>$
<(a, c)(b)(a)>-Projected database
C2/[18:18]$<_{24}(c, d)>$
<(b)(a)>-Projected database
<(b)(d)>-Projected database
<(b)(e)>-Projected database
<(b)(e)(d)>-Projected database
<(c)(b)>-Projected database
<(c)(b)(a)>-Projected database
<(c)(e)>-Projected database
<(c, d)>-Projected database

Figure 2.5 An example of DELISP algorithm

2.5 CloSpan Algorithm

In [31], CloSpan algorithm (**Clo**sed **S**equential **pa**tter**n** mining) first is proposed to mine closed sequential patterns. Physical projection and pseudo projection are implemented to discover sequential patterns using itemset extension or sequence extension. CloSpan generates super-set of all closed sequential patterns and store them in a candidate tree. CloSpan maintains potentially closed patterns in memory and conducts post-pruning by checking them with new discovered patterns. The novel ideas of partial order and equivalence of projected databases are proposed to detect early termination condition for pruning search space by checking the constructed hash table and the candidate tree.

2.6 BIDE Algorithm

In [30], BIDE (**BI**-**D**irection **E**xtension based closed frequent sequence mining) algorithm has shown the better performance than CloSpan. BIDE scans database once to discover frequent 1-sequnces and build their pseudo-projection databases. The local frequent items are found in the projected databases for each prefix to grow the prefix. BIDE dose not maintain any candidates in memory by utilizing the Bi-direction extension strategy for checking closure of each prefix. Forward-extension event checking checks the support of the prefix with its all local frequent items and Backward-extension event checking checks whether there exits a backward extension item having the same support with the prefix. A prefix can be a closed pattern if there is neither forward extension item nor backward extension item. Additionally, BIDE uses backscan search pruning to eliminate unpromising patterns and stop to mine their projected databases.

2.7 Par-CSP Algorithm

In [11], a parallel algorithm Par-CSP (**Par**allel **C**losed **S**equential **P**attern mining) which is based BIDE algorithm has been developed to efficiently mine closed sequential patterns for large databases on a distribution memory system. Each processor counts the local supports of all items in their partly datasets and their true supports can been obtained by a global add reduction. Further, each processor builds pseudo-projections for each frequent item and broadcast to others processors. The identification of the projected database for each frequent item is store in a queue which is hold by a master processor. The databases in the queue are assigned to each processors sequentially, and the BIDE algorithm [30] is applied to mine closed sequential patterns until the queue is empty. The selective sampling is used to find some extra-large databases which are very time-consuming and partition them to several small sub-databases. Therefore, Par-CSP can mine all patterns efficiently even if the database is very large.

Chapter 3 METISP: MEmory Time-Indexing Sequential Pattern mining

We have developed a new algorithm, called METISP, for time-constrained sequential pattern mining. METISP uses the pattern-growth strategy similar to PrefixSpan [25] and DELISP [19] algorithms. METISP algorithm is the first algorithm which can solve all time constraints including minimum/maximum/exact gap, sliding window and duration constraints at the same time. The novel algorithm utilizes the technique of memory indexing to mark the timestamps. Hence, the inherent properties of time constraints can be applied to reduce the search space and improve the efficiency of mining time-constrained sequential patterns greatly. For extra-large database, the algorithm applies the idea of partition-and-verification for mining sequential patterns with time constraints.

The problem is formulated in Section 3.1. Section 3.2 introduces the terminology used in this chapter. The details of METISP algorithm is described in Section 3.3. An example of mining using METISP is illustrated in Section 3.4. Sections 3.5 and 3.6 describe the technique of partition-and-verification for dealing with extra-large databases and the differences between METISP algorithm and prefix-growth algorithm respectively. The experimental evaluations are reported in Sections 3.7 and the performance results are shown in Section 3.8. Finally, we give a summary in Section 3.9. For convenience, we refer to the sequence database as *DB* and a data sequence as *ds* in the following context.

3.1 Problem Statement

Let $\Psi = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ be a set of literals, called *items*. An *itemset* $I = (\beta_1, \beta_2, \ldots, \beta_q)$ is a nonempty set of q items such that $I \subseteq \Psi$. A *sequence s*, denoted by

$<e_1e_2\ldots e_w>$, is an ordered list of *w elements* where each *element* e_i is an itemset. Without loss of generality, we assume the items in an element are in lexicographic order. The *length* of a sequence *s*, written as $|s|$, is the total number of items in all the elements in *s*. Sequence *s* is a *k-sequence* if $|s| = k$. The sequence database *DB* contains $|DB|$ data sequences. A *data sequence ds* has a unique identifier *sid* and is represented by $<_{t_1}e_1{'}\ _{t_2}e_2{'}\ \ldots\ _{t_n}e_n{'}>$, where element $e_i{'}$ occurred at time t_i, $t_1 < t_2 < \ldots < t_n$.

A user gives a parameter *minsup* (minimum support) and four time-constraints *maxgap* (maximum gap), *mingap* (minimum gap), *swin* (sliding window) and *duration* to discover the set of all time-constrained sequential patterns. A sequence *s* is a *time-constrained sequential pattern* if $s.sup \geq minsup$, where *s.sup* is the *support* of the sequence *s* and *minsup* is the user specified minimum support threshold. The *support* of *s* is the number of data sequences *containing s* divided by $|DB|$. A data sequence $ds = <_{t_1}e_1{'}\ _{t_2}e_2{'}\ldots\ _{t_n}e_n{'}>$ *contains* a sequence $s = <e_1e_2\ldots e_w>$ if there exist integers $l_1, u_1, l_2, u_2, \ldots, l_w, u_w$ and $1 \leq l_1 \leq u_1 < l_2 \leq u_2 < \ldots < l_w \leq u_w \leq n$ such that the five conditions hold: (1) $e_i \subseteq (e_{l_i}{'} \cup \ldots \cup e_{u_i}{'})$, $1 \leq i \leq w$ (2) $t_{u_i} - t_{l_i} \leq swin$, $1 \leq i \leq w$ (3) $t_{u_i} - t_{l_{i-1}} \leq maxgap$, $2 \leq i \leq w$ (4) $t_{l_i} - t_{u_{i-1}} \geq mingap$, $2 \leq i \leq w$ (5) $t_{u_w} - t_{l_1} \leq duration$. Assume that t_j, *mingap*, *maxgap*, *swin*, and *duration* are all positive integers, $maxgap \geq mingap \geq 1$. Common sequential pattern mining without time constraints is a special case by setting $mingap = 1$, $maxgap = \infty$, $swin = 0$ and $duration = \infty$. When *mingap* is the same as *maxgap*, the time gap is called *exact gap*.

For example, given a sequence *DB* of four data sequences with their sids in Table 3.1, the rightmost column shows the time-constrained sequential patterns with constraints *mingap*=3, *maxgap*=15, *swin*=2 and *duration*=25. In the *DB*, data sequence C4 $<_5(a)_{10}(d)_{21}(c, d)_{26}(e)>$ has four itemsets occurring at time 5, 10, 21 and

26, respectively. The third itemset of C4 has two items *c* and *d*. Sequences <(a, c)(b)> and <(b)(e)(d)> are both 3-sequences. The sequence <(a, c)(b)> is contained in data sequences C1 $<_3(c)_5(a, f)_{18}(b)_{31}(a)_{45}(f)>$ because element (a, c) can be contained in the transaction combining $_3(c)$ and $_5(a, f)$ for 5−3≦2 (swin). Meanwhile, the other constraints can be satisfied for 18−5≧3 (mingap), 18−3≦15(maxgap) and total time span 18−3≦25 (duration). Similarly, <(a, c)(b)> is contained in C2. The support of <(a, c)(b)> is 2/4 and it is a time-constrained sequential pattern for *minsup* = 50%. Considering sequence <(a)(b)(a)>, it can be contained in C1 for the other constraints but it fails the *duration* constraint (31−5>25). Given *mingap* = *maxgap* = 7, that is, exact gap = 7, C3 $<_1(b)_{20}(b, g)_{27}(e)_{34}(d, g)_{35}(g)>$ contains pattern <(b)(e)(d)> but it does not contain pattern <(b)(d)> because 34−20≠7 (exact gap).

Table 3.1 Sequence database (DB) and the time-constrained sequential patterns

Sid	Sequence	Time-constrained sequential pattern (*minsup*=50%, *mingap*=3, *maxgap*=15, *swin*=2, *duration*=25)
C1	$<_3(c)_5(a, f)_{18}(b)_{31}(a)_{45}(f)>$	<(a)>:3, <(a)(b)>:2, <(a)(d)>:2, <(a, c)>:2, <(a,c)(b)>:2, <(b)>:3, <(b)(a)>:2, <(b)(d)>:2, <(b)(e)>:2, (b)(e)(d)>:2, <(c)>:3, <(c)(b)>:2, <(c)(e)>:2, <(c, d)>:2, <(d)>:3, <(e)>:3, <(e)(d)>:2
C2	$<_6(a, c)_{10}(b)_{17}(e)_{18}(a)_{24}(c, d)>$	
C3	$<_1(b)_{20}(b, g)_{27}(e)_{34}(d, g)_{35}(g)>$	
C4	$<_5(a)_{10}(d)_{21}(c, d)_{26}(e)>$	

3.2 Terminology Used in the Proposed Algorithm

Definition 1 (frequent item) An item *x* is called a *frequent item* in *DB* if <(*x*)>*.sup* ≥ *minsup*.

Definition 2: (type-1 pattern, type-2 pattern, stem, prefix) Given a frequent pattern *P* and a frequent item *x* in *DB*, *P'* is a *type-1 pattern* if it can be formed by adding an itemset of the single item *x* after the last element of *P*, and a *type-2 pattern* if formed by appending *x* to the last element of *P*. The item *x* is called the *stem* of the new frequent pattern *P'*. The *prefix pattern* (abbreviated as *prefix*) of *P'* is *P*.

For example, <(a)>, <(a)(b)>, <(a)(d)> and <(a, c)> in Table 3.1 are the discovered time-constrained sequential patterns. <(a)(b)> and <(a)(d)> are type-1 patterns by adding (b) and (d) after (a) respectively and <(a, c)> is a type-2 pattern by appending (c) to (a). Here, (b), (d) and (c) are the stems and <(a)> is their prefix.

Definition 3: (initial time, last-start time, last-end time, time index) Let the first element of a frequent pattern *P* be *FE* and last element be *LE*. If *ds* contains *P* by having $FE \subseteq e_{\delta} \cup e_{\delta+1} \cup \ldots \cup e_{\varepsilon}$ and $LE \subseteq e_{\gamma} \cup e_{\gamma+1} \cup \ldots \cup e_{\omega}$, where $e_{\delta}, \ldots, e_{\omega}$ are element in *ds*, the occurring time t_{δ}, t_{γ}, and t_{ω} for itemsets e_{δ}, e_{γ} and e_{ω}, are named *initial time* (abbreviated as it), *last-start time* (abbreviated as lst) and *last-end time* (abbreviated as let) of *P* in *ds*. The it:lst:let is marked as $e_{\delta} : e_{\gamma} : e_{\omega}$. Every occurrence of timestamp it:lst:let is collected as [it$_1$:lst$_1$:let$_1$, it$_2$:lst$_2$:let$_2$,…, it$_k$:lst$_k$:let$_k$], it$_i$≤lst$_i$≤let$_i$ for 1≤i≤k. Such a timestamp lists is called the *time index* of *P* in *ds*.

For example, given a sequence database as shown in Table 3.1, minimum support=50%, mingap=5, maxgap=16, swin=0, and duration=30. <(a)(d)> is a sequential pattern contained by C2 and C4. For C2 <$_{6}$(a, c)$_{10}$(b)$_{17}$(e)$_{18}$(a)$_{24}$(c, d)>, the time stamp is marked as 18:24:24. The initial time represents (a) of <(a)(d)> occurs at time 18. The last-start time and last-end time represents (d) of <(a)(d)> occurs at time 24. Similarly, for C4 <$_{5}$(a)$_{10}$(d)$_{21}$(c, d)$_{26}$(e)>, <(a)(d)> appears two times and the time index of <(a)(d)> is collected as [5:10:10, 5:21:21].

Definition 4: (valid time periods) Given a time index of *P* in *ds*, the time periods of the itemsets in *ds* to be used for finding a potential stem *x* of pattern *P'*, where *P* is the prefix of *P'*, are called *valid time periods* (abbreviated as VTPs).

Lemma 1: Given a time index of *P* in *ds* [it_1:lst_1:let_1, it_2:lst_2:let_2,…, it_k:lst_k:let_k], the valid time period to form a type-1 pattern must satisfy either one of the following conditions, as illustrated in Figure 3.1:

$$let_i+mingap \leq VTP \leq \min\{lst_i+maxgap,\ it_i+duration\},\ \exists\, i, 1 \leq i \leq k$$

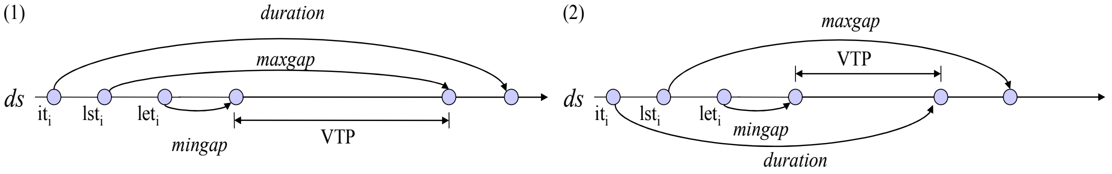


Figure 3.1 Valid time periods of a potential type-1 pattern (Type-1 VTPs)

For example, in the rightmost column of Table 3.1, the time-constrained sequential pattern <(b)(e)> is contained in C2 and C3. For C2 <$_6$(a, c)$_{10}$(b)$_{17}$(e)$_{18}$(a)$_{24}$(c, d)>, the time index is marked as [10:17:17]. For the type-1 patterns, the VTP can be obtained as 17+3(mingap) to 17+15(maxgap) because the duration limitation of 10+25 (duraion) is larger than the maxgap limitation of 17 +15(maxgap). Similarly, for C3, a stem (d) can be found within those time periods and <(b)(e)(d)> is a newly discovered pattern.

Lemma 2: Given a time index of *P* in *ds* [it_1:lst_1:let_1, it_2:lst_2:let_2,…, it_k:lst_k:let_k], the valid time period to form a type-2 pattern must satisfy either one of the following conditions, as illustrated in Figure 3.2:

$$let_i-swin \leq VTP \leq \min\{lst_i+swin,\ it_i+duration\},\ \exists\, i, 1 \leq i \leq k$$

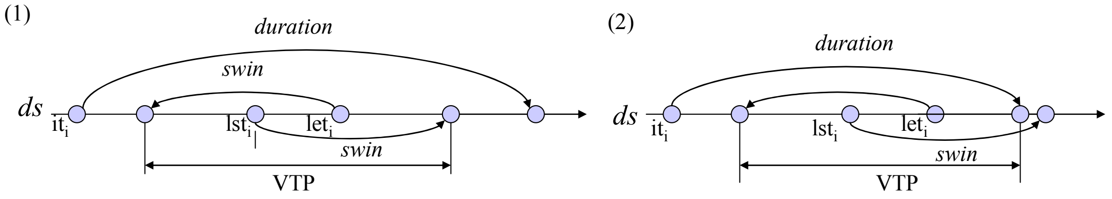


Figure 3.2 Valid time periods of a potential type-2 pattern (Type-2 VTPs)

Figure 3.2 shows the two conditions in Lemma 2. Note that the potential stem found using Lemma 2 can be pruned if it is lexicographically smaller than the items of the last element in *P*. For any type-2 pattern, the valid time periods are checked on not violating the minimum/maximum gap constraints between the last two elements. In case *P* has only one element and the VTP to be used is earlier than st_i, then *duration* constraint must be checked additionally.

For instance, the VTP of the type-2 pattern <(b)(e)> can be obtained as 17−2(swin) to 17+2(swin) for C2 $<_6(a, c)_{10}(b)_{17}(e)_{18}(a)_{24}(c, d)>$. The (a) appears at time 18, the support count therefore cannot be increased since the lexicographical order of (a) is small than (e).

3.3 METISP Algorithm: A Detailed Description

Figure 3.3 outlines the METISP Algorithm. METISP mines patters within the pattern-growth framework similar to the pseudo projection version of PrefixSpan algorithm [25], while it can handle constraints minimum/maximum gaps, duration and sliding time-window. Assume that the DB can fit into the main memory, METISP first loads DB into memory (as MDB) and scans MDB once to find all frequent items. With respect to each frequent item, METISP then constructs a time index-set for the 1-sequence item and recursively forms time-constrained sequential patterns of longer length. The time index-set is a set of (data-sequence pointer, time index) pairs. Only those data sequences containing that item would be included. The time index indicates the list of it:lst:let triplets as described in Definition 3.

In Figure 3.4 Mine (P, P-TIdx) mines type-1 patterns and type-2 patterns having prefix P by effectively locating VTPs using Lemma 1 and Lemma 2, respectively. The P-TIdx is the time index-set for P. METISP thus never search the data sequences irrelevant to P. Moreover, the VTPs ensure that METISP locates and counts the effective stems which can form valid patterns, rather than the whole set of items in the data sequence. Likewise, for a newly formed type-1 pattern or type-2 pattern P’, its time index-set P’-TIdx is constructed and Mine(P’, P’-TIdx) is invoked. By pushing time attributes deeply into the mining process, METISP efficiently discovers the desired patterns.

```
Algorithm: METISP
Input: DB (a sequence database), minsup (minimum support), mingap (minimum gap),
maxgap (maximum gap), swin (sliding time-window), duration (duration)
Output: the set of all time-constrained sequential patterns
1. load DB into memory (as MDB) and scan MDB once to find all frequent items.
2. for each frequent item x,
(1) form the sequential pattern P = <(x)> and output P.
(2) scan MDB once to construct P-Tidx, time index set of x.
(3) call Mine(P, P-Tidx)
```

Figure 3.3 Algorithm METISP

Theorem 1: All the sequential patterns can be divided into type-1 patterns and type-2 patterns. With the time constraints, a stem and a prefix must satisfy all time restrictions in data sequences to form a new type-1 or type-2 pattern. METISP mines all patterns by type-1 and type-2 pattern growth recursively. For each prefix, METISP counts the supports of items in all VTPs which have satisfied all time constraints as that all stems are found in the VTPs of type-1 and type-2 patterns to form type-1 and type-2 patterns, respectively. Hence, METISP mines completely time-constrained sequential patterns finally.

```
----------------------------------------------------------------------------------------------------
Subroutine: Mine (P, P-TIdx)
Parameter: P = prefix pattern, P-Tidx = time index set
1. for each data sequence ds in the P-DB, // P-DB: sequences indicated in P-TIdx
  (1) use the corresponding time index to collect the type-1 VTPs satisfying:
          $let_i + mingap \le VTP \le \min\{lst_i + maxgap,\ it_i + duration\},\ \exists\, i, 1 \le i \le k$
  (2) for each item in the VTPs of type-1 pattern, add one to its support count.
  (3) use the corresponding time index to collect the type-2 VTPs satisfying:
          $let_i - swin \le VTP \le \min\{lst_i + swin,\ it_i + duration\},\ \exists\, i, 1 \le i \le k$
  (4) for each item in the VTPs of type-2 pattern, add one to its support count
2. for each item x' found in the VTPs of type-1 pattern and its support is greater than
     or equal to minsup,
  (1) form the type-1 pattern P' by extending stem x' and output P'.
  (2) scan the VTPs of each ds in P-DB to construct P'-Tidx, time index set of x'.
   // in this, the VTPs must satisfy the formulas indicated in Lemma 1
  (3) call Mine(P', P'-TIdx);
3. for each item x' found in the VTPs of type-2 pattern and its support is greater than
     or equal to minsup,
  (1) form the type-2 pattern P' by appending stem x' and output P'.
  (2) scan the VTPs of each ds in P-DB to construct P'-Tidx, time index set of x'.
   // in this, the VTPs must satisfy the formulas indicated in Lemma 2
  (3) call Mine(P', P'-TIdx);
----------------------------------------------------------------------------------------------------
```

Figure 3.4 Subroutine Mine() of METISP algorithm

3.4 An Illustrating Example of Mining Patterns Using METISP

Example: Given a DB in Table 3.1, minsup=50%, *mingap*=3, *maxgap*=15, *swin*=2 and *duration*=25, METISP mines sequential patterns by the following steps.

Step 1: Load DB into memory and find all frequent items.

METISP first reads the DB into memory and scans the in-memory DB (abbreviated as MDB) once to find frequent items. <(a)>:3, <(b)>:3, <(c)>:3, <(d)>:3,<(e)>:3 are found, <(a)>:3 shows its support count of 3. For each frequent item, METISP uses step 2 to construct the Time Index Set.

Step 2: Construct the Time Index Set of frequent items

For each frequent item, METISP scans MDB once to construct the time index set which contains the time indexes and the sequence pointers where the item appears. Take <(a)> for example, METISP scans MDB and constructs the time index set <(a)>-Tidx, as shown in Figure 3.5(a). Three pointers appear in <(a)>-Tidx since <(a)> appears in C1, C2, and C4. For C1 <$_3$(c)$_5$(a, f)$_{18}$(b)$_{31}$(a)$_{45}$(f)>, element (a) occurs at time 5 and 31 so the time index is marked as [5:5:5, 31:31:31]. The indexes for C2 and C4 are processed accordingly. The <(a)>-Tidx is used in subroutine Mine, which searches potential type-1 and type-2 stems.

Step 3: Find stems in VTPs from the time index set and grow (discover) patterns (type-1 and type-2 stems of <(a)>)

For each sequence containing the prefix P, METISP uses the time index to find out the valid time periods of P for type-1 patterns and type-2 patterns and count the supports of all items within the valid time periods respectively. METISP finds all the stems quickly and growths type-1 patterns and type-2 patterns.

Subroutine Mine computes the supports of potential stems here. With respect to prefix <(a)>, the type-1 VTPs for C1 is [8:20, 34:46] from [lst_1+mingap:let_1+maxgap, lst_2+mingap:let_2+maxgap], where $\text{lst}_1 = \text{let}_1 = 5$ and $\text{lst}_2 = \text{let}_2 = 31$. The VTPs of C2

[9:21, 21:33] and C4 [8:20] are obtained similarly. Likewise, the type-2 VTPs for C1 is [3:7, 29:33] from [lst_1+swin:let_1-swin, lst_2+swin:let_2-swin], for C2 is [4:8, 16:20], and for C4 is [3:7]. Now, the supports of items *b* and *d* pass the threshold to be new type-1 stems and items *c* has enough support to form a new type-2 stem. Thus, <(a)(b)> (and later <(a)(d)>) is outputted to form two new type-1 patterns and <(a, c)> is outputted to form a new type-2 pattern. METISP further uses step 4 to construct the time index set of <(a)(b)> and grow pattern in subroutine Mine. Then <(a)(b)>, <(a)(d)> and <(a, c)> are processed in turn using the same steps.

Step 4: Construct the time index set of the sequential pattern and discover all sequential patterns recursively.

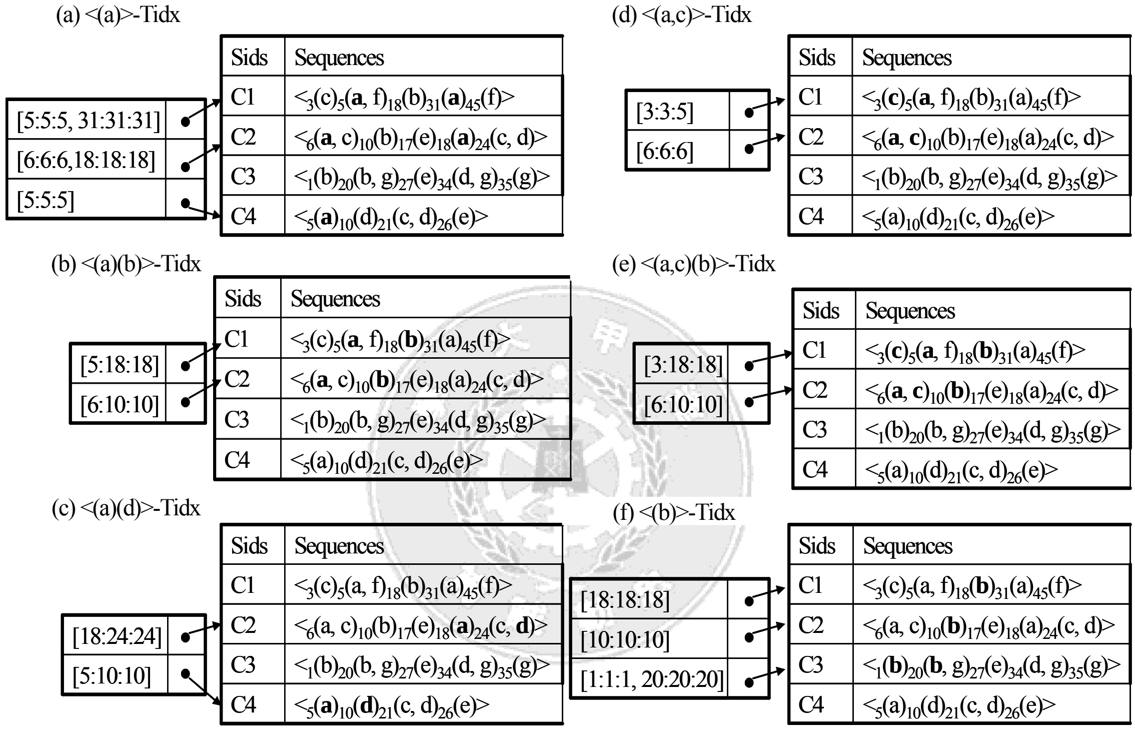
For each sequential pattern P’ with prefix P and stem x, METISP scans each data sequence containing P and records all the initial time, last-start time and last-end time of P’ within the valid time periods of P. METISP also labels sequence pointers *s_ptrs* containing P’ simultaneously. Thus, <(a)(b)>-Tidx is constructed, and Mine is called recursively. The <(a)(b)>-Tidx is shown in Figure 3.5(b). Considering type-1 VTPs of <(a)(b)>, which are [21:33] for C1 and [13:25] for C2, no more stems can be found. No pattern of prefix <(a)(b)> can be formed. Similarly, type-2 VTPs of <(a)(b)> is computed to find potential type-2 stems.

With respect to prefix <(a)(b)>, the type-2 VTPs for C1 is [16:20, 8:12] from [let_1−swin:lst_1+swin, let_2-swin:lst_2+swin], where $\mathrm{lst}_1 = \mathrm{let}_1 = 18$ and $\mathrm{lst}_2 = \mathrm{let}_2 = 10$. No pattern of prefix <(a)(b)> can be formed, the process returns to work on <(a)(d)>. Unfortunately, no stems can be found within both type-1 and type-2 VTPs of <(a)(d)>. The process returns to work on the type-1 and type-2 stems of <(a, c)>.

With respect to prefix <(a, c)>, <(a, c)>-Tidx is constructed. Note that in Figure 3.5(d), the initial time reflects the *swin* effect so that the time index for C1 is [3:3:5]. Subroutine Mine is then called recursively. Referring to <(a, c)>-Tidx, the type-1

VTPs for C1 is [8:18] from [5+mingap:3+maxgap], for C2 is [6+mingap:6+maxgap].

Subroutine Mine finds item *b,* outputs <(a, c)(b)>. For the type-2 VTPs of <(a, c)>, Mine cannot form any type-2 pattern and only <(a, c)(b)>-Tidx is constructed finally. However, no more pattern is formed and the mining of prefix <(a)> now stops. METISP then applies steps 2 to 4 on <(b)>, <(c)>, <(d)>, and <(e)>. The complete set of time-constrained patterns is listed in the rightmost column in Table 3.1.



(a) <(a)>-Tidx

[5:5:5, 31:31:31]
[6:6:6,18:18:18]
[5:5:5]

Sids	Sequences
C1	$<_{3}(c)_{5}(\mathbf{a}, f)_{18}(b)_{31}(\mathbf{a})_{45}(f)>$
C2	$<_{6}(\mathbf{a}, c)_{10}(b)_{17}(e)_{18}(\mathbf{a})_{24}(c, d)>$
C3	$<_{1}(b)_{20}(b, g)_{27}(e)_{34}(d, g)_{35}(g)>$
C4	$<_{5}(\mathbf{a})_{10}(d)_{21}(c, d)_{26}(e)>$

(b) <(a)(b)>-Tidx

[5:18:18]
[6:10:10]

Sids	Sequences
C1	$<_{3}(c)_{5}(\mathbf{a}, f)_{18}(\mathbf{b})_{31}(a)_{45}(f)>$
C2	$<_{6}(\mathbf{a}, c)_{10}(\mathbf{b})_{17}(e)_{18}(a)_{24}(c, d)>$
C3	$<_{1}(b)_{20}(b, g)_{27}(e)_{34}(d, g)_{35}(g)>$
C4	$<_{5}(a)_{10}(d)_{21}(c, d)_{26}(e)>$

(c) <(a)(d)>-Tidx

[18:24:24]
[5:10:10]

Sids	Sequences
C1	$<_{3}(c)_{5}(a, f)_{18}(b)_{31}(a)_{45}(f)>$
C2	$<_{6}(a, c)_{10}(b)_{17}(e)_{18}(\mathbf{a})_{24}(c, \mathbf{d})>$
C3	$<_{1}(b)_{20}(b, g)_{27}(e)_{34}(d, g)_{35}(g)>$
C4	$<_{5}(\mathbf{a})_{10}(\mathbf{d})_{21}(c, d)_{26}(e)>$

(d) <(a,c)>-Tidx

[3:3:5]
[6:6:6]

Sids	Sequences
C1	$<_{3}(\mathbf{c})_{5}(\mathbf{a}, f)_{18}(b)_{31}(a)_{45}(f)>$
C2	$<_{6}(\mathbf{a}, \mathbf{c})_{10}(b)_{17}(e)_{18}(a)_{24}(c, d)>$
C3	$<_{1}(b)_{20}(b, g)_{27}(e)_{34}(d, g)_{35}(g)>$
C4	$<_{5}(a)_{10}(d)_{21}(c, d)_{26}(e)>$

(e) <(a,c)(b)>-Tidx

[3:18:18]
[6:10:10]

Sids	Sequences
C1	$<_{3}(\mathbf{c})_{5}(\mathbf{a}, f)_{18}(\mathbf{b})_{31}(a)_{45}(f)>$
C2	$<_{6}(\mathbf{a}, \mathbf{c})_{10}(\mathbf{b})_{17}(e)_{18}(a)_{24}(c, d)>$
C3	$<_{1}(b)_{20}(b, g)_{27}(e)_{34}(d, g)_{35}(g)>$
C4	$<_{5}(a)_{10}(d)_{21}(c, d)_{26}(e)>$

(f) <(b)>-Tidx

[18:18:18]
[10:10:10]
[1:1:1, 20:20:20]

Sids	Sequences
C1	$<_{3}(c)_{5}(a, f)_{18}(\mathbf{b})_{31}(a)_{45}(f)>$
C2	$<_{6}(a, c)_{10}(\mathbf{b})_{17}(e)_{18}(a)_{24}(c, d)>$
C3	$<_{1}(\mathbf{b})_{20}(\mathbf{b}, g)_{27}(e)_{34}(d, g)_{35}(g)>$
C4	$<_{5}(a)_{10}(d)_{21}(c, d)_{26}(e)>$

Figure 3.5 Some time index sets of sequential patterns

3.5 Dealing with Extra-large Databases

With the ever-increasing installed memory size, many databases can fit into main memory completely and METISP is able to mine the time-constrained sequential patterns. However, some databases still might be too large to be in the main memory. Thus, we propose a flexible partition-and-verification technique to overcome the limitation, as shown in Figure 3.6.

The technique of partition-and-verification contains two steps: partitioned mining and candidate verification. In the first step, the extra-large database is partitioned into several partitions, in each partition the data sequences are loaded one by once as many as possibly until the memory is not enough. Then METISP algorithm is applied to each partition to discover all potentially time-constrained sequential patterns. Those candidate patterns are collected from all time-constrained sequential patterns in each partition with the same *minsup* and time constraints.

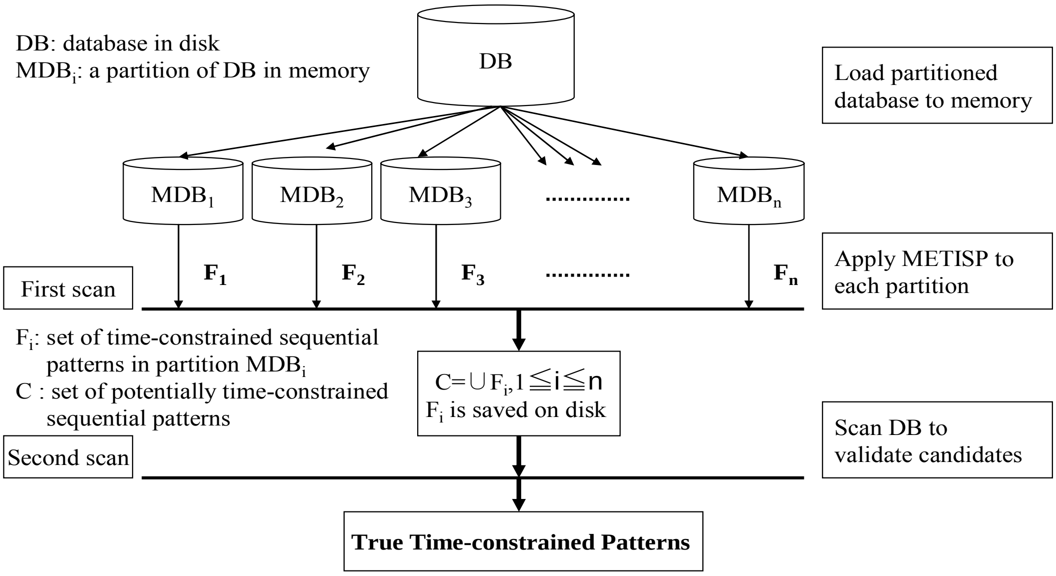

DB: database in disk
MDB$_i$: a partition of DB in memory
DB
Load partitioned database to memory
MDB$_1$
MDB$_2$
MDB$_3$
....
MDB$_n$
First scan
F$_1$
F$_2$
F$_3$
....
F$_n$
Apply METISP to each partition
F$_i$: set of time-constrained sequential patterns in partition MDB$_i$
C : set of potentially time-constrained sequential patterns
C=∪F$_i$,1≦i≦n
F$_i$ is saved on disk
Scan DB to validate candidates
Second scan
True Time-constrained Patterns

Figure 3.6 An overview of database partitions and pattern verifications

In the second step, the supports of those candidate patterns can be validated. We simply compare each data sequences with each candidate pattern to count its support. In order to check whether the candidate patterns satisfy all time constraints, we use the similar idea from which METISP delimiting the search space to find type-1 or type-2 patterns, by recording timestamps of their prefixes only. Then, all true patterns can be discovered after scanning each data sequences in the extra-large database once so that METISP can overcome the limitation of database size, by scanning the database at most twice.

3.6 Difference between METISP Algorithm and prefix-growth Algorithm

We compare the METISP algorithm and the prefix-growth algorithm below.

(1)Database projections: As the main memory is sufficient to hold the entire database, METISP loads DB into main memory and construct time index sets to discover all time-constrained sequential patterns recursively. Otherwise, a partition technique is applied with the algorithm. At most two times of database scanning is required and no databases are projected. The framework of prefix-growth [26] is similar to the PrefixSpan algorithm [25]. When the memory is not enough, for keeping the database at the same time, prefix-growth needs to project many databases so that the I/O cost is relatively high. For each database projected, prefix-growth removes infrequent items and useless sequences which do not satisfy constraints. For a time-constrained sequential pattern of length k, prefix-growth must project databases k times in the worst case.

(2)Time index: The prefix-growth in [26] adopts optimized techniques of bi-level and pseudo-projection to reduce the time and cost of projected databases. However, in order to handle the complicated time constraints, it must know when the pattern is appeared in each containing sequences. In reality, a pattern may occur multiple times in the same sequence and the time relationships to be considered are too huge. Also, the detailed implementation of prefix-growth is not reported in [26].

METISP proposes the idea of memory time index to record all timestamps of the occurrences of a pattern. METISP algorithm uses variable and multiple timestamps recorded in time index set to manage all time constraints, including sliding window which is not considered by the prefix-growth algorithm. The sliding window constraint not only affects the counting method of time gaps including minimum gap and maximum gap but also destroys the original frame of database projection proposed in [26]. The memory time index set used in METISP is effective to handle those time constraints by using indexes of timestamp with the methodology of pattern-growth.

(3)Search space limitation: The stems reflect the search space differences between METISP and prefix-growth. METISP utilizes time index set to limit the search space for discovering stems. The item found in VTPs having support no less than the specified threshold will become a stem to form a new sequential pattern. In comparison, prefix-growth scans all projected database to find all local frequent items and needs to examine whether a new pattern satisfies all time constraints. Additionally, prefix-growth removes infrequent items and unless sequences to reduce search space when projecting databases.

3.7 Experimental Evaluations

Extensive experiments were performed to assess the performance of the METISP algorithm. Time-constrained mining algorithms including GSP [3] and DELISP [19] were compared with METISP, using the IBM synthetic data generation program [1]. The parameters used in the IBM synthetic data generation program are shown in Table 3.2. The parameters N_S and N_I used to generate the database in all the experiments are fixed as 5000 and 25000, respectively. All the experiments were performed on an AMD 2400+ PC with 1GB memory running the Windows XP.

The experimental results are described in two parts. In the first part, the results of experiments varying different constraints are discussed. These constraints include minsup and the five time constraints mingap, maxgap, swin, duration, and exact gap. We describe the result of dataset C10-T2.5-S4-I1.25 having 100000 data sequences (|DB| = 100k), and N=1000. The results of varying |C|, |T|, |S|, and |I| were consistent.

Additionally, the executions of scalability are also reported. In the experiments of database scalability, two scenarios are described. First, the database can be fit into main memory completely. The sizes of those databases vary from 100k to 1000k. Next, the main memory is not enough to fit whole database. In this, the sizes of database are varied from 1000k to 10000k.

Table 3.2 Parameters in the experiments

Parameter	Description
|DB|	Number of data sequences in database DB
|C|	Average number of transaction per data-sequence
|T|	Average number of items per transaction
|S|	Average size of potentially sequential patterns
|I|	Average size of potentially frequent itemsets
N_S	Number of maximal potentially sequential patterns
N_I	Number of maximal potentially frequent itemsets
N	Number of items

Second, the experiments focus on changing different parameter setting including |C|, |T|, |S|, |I| and N. METISP is compared with GSP and DELISP algorithms by with respect to different databases. The characteristics of databases are shown in Table 3.3.

Table 3.3 The setting of database parameters

Database	|C|	|T|	|S|	|I|	N	|DB|
DB1	10	2.5	4	1.25	10k	100k
DB2	**15**	2.5	4	1.25	10k	
DB3	15	**5**	4	1.25	10k	
DB4	15	2.5	**8**	1.25	10k	
DB5	15	2.5	4	**2.5**	10k	
DB6	15	2.5	4	1.25	**1k**	

3.8 Performance Results

The total execution times of the three algorithms on minimum support, *mingap*, *maxgap*, and *swin* were compared. Figure 3.7 shows that, on average, the total execution times of GSP [3] and DELISP [19] are about 5.6 and 3.9, respectively, times of METISP. We show the results of varying mingap in Figure 3.8, varying maxgap in Figure 3.9, and varying swin in Figure 3.10. No other constraints were set in the experiments so as to observe the effect of the single constraint. METISP outperforms GSP and DELISP for all these experiments. The minsup of the three experiments is all fixed as 0.0035 as to have longer patterns. In fact, METISP constantly mines the patterns about 55 seconds, for the distinct experiments with minisup 0.0075. Those results are not shown in this. When we increase the mingap (Figure 3.8) or decrease the maxgap value (Figure 3.9), the number of patterns will decrease since both changes restrict some data sequences to contain certain patterns. When swin is increased (Figure 3.10), more patterns are discovered, and more execution time is required with the increased swin.

The results of changing duration and exact-gap values are shown in Figures 3.11 and 3.12, respectively. We depict the results of METISP because GSP and DELISP cannot handle the constraints. We report the result of different minsup from 0.0035 to 0.001 by varying time constraints. When the increase of duration, the patterns discovered becomes more and the execution time does. The METISP algorithm also has good performance for different values of exact-gap constraint.

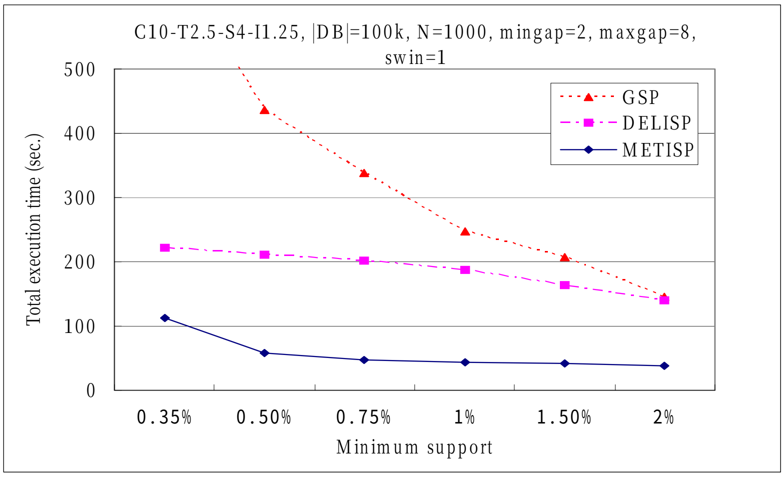


Figure 3.7 Effect of varying minimum support

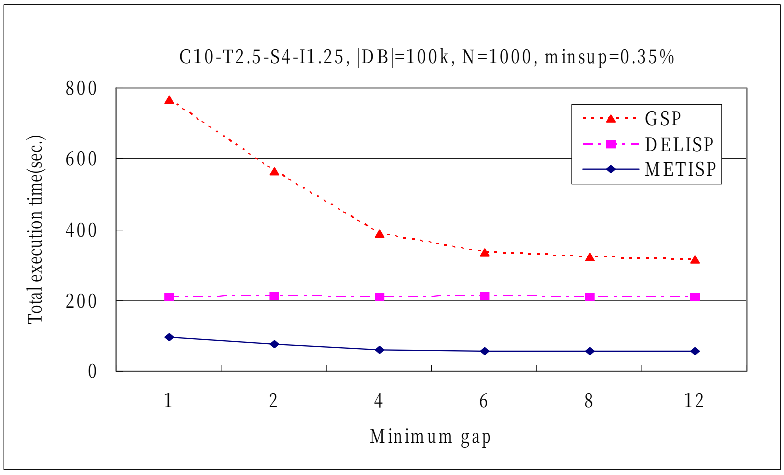


Figure 3.8 Effect of varying minimum gap

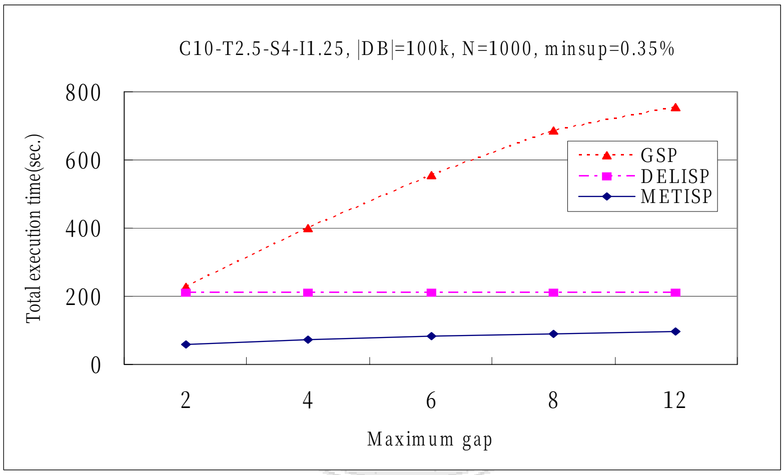


Figure 3.9 Effect of varying maximum gap

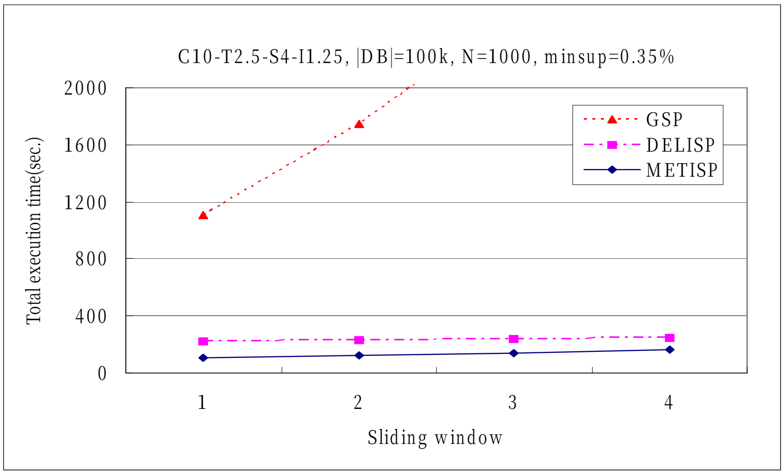


Figure 3.10 Effect of varying sliding window

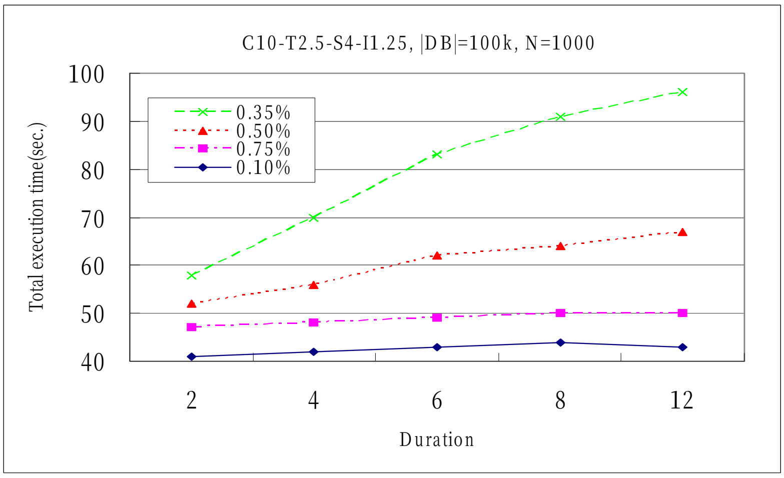


Figure 3.11 Effect of varying duration

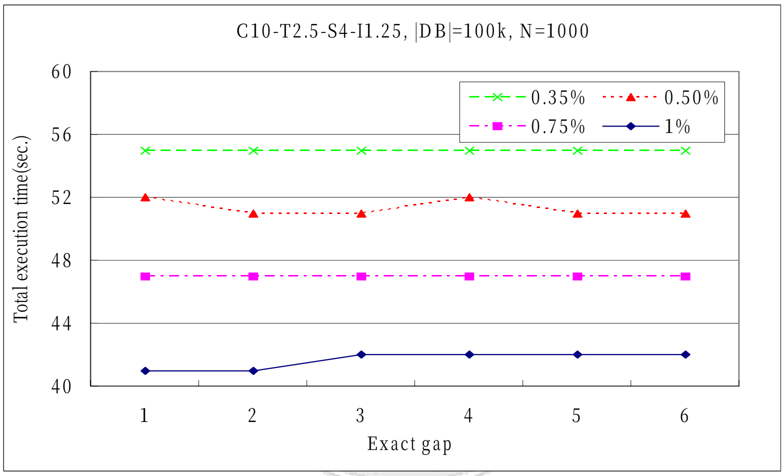


Figure 3.12 Effect of varying exact gap

Figures 3.13 and 3.14 show that the METISP can has good scalability whether the database can be load into memory wholly or not. The minimum is 0.0075 and the time constraints are set as 8(maxgap), 2(mingap), 1(swin) and no constraint(duration). In figure 3.13, for the database having huge number of data sequences, the database is partition into several parts and the algorithm needs scan database two times as to require more cost than that without database partition. Carefully, the execution time of all algorithms in this is relative to that of METISP which runs database with 1000k data sequences.

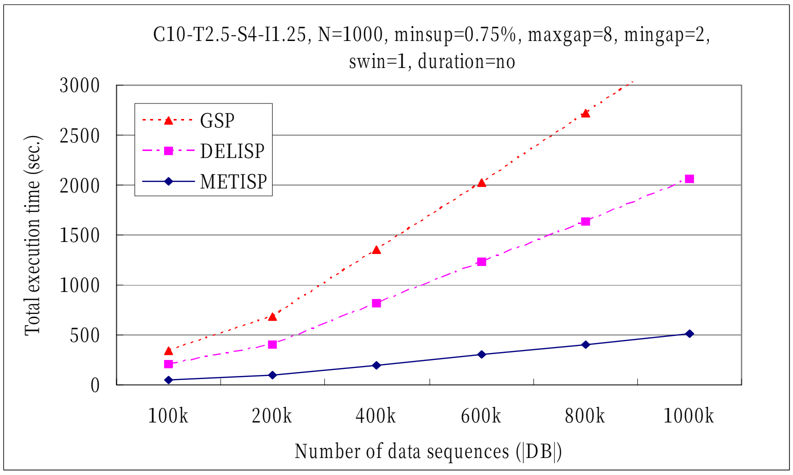


Figure 3.13 Linear scalability of the database size from 100k to 1000k

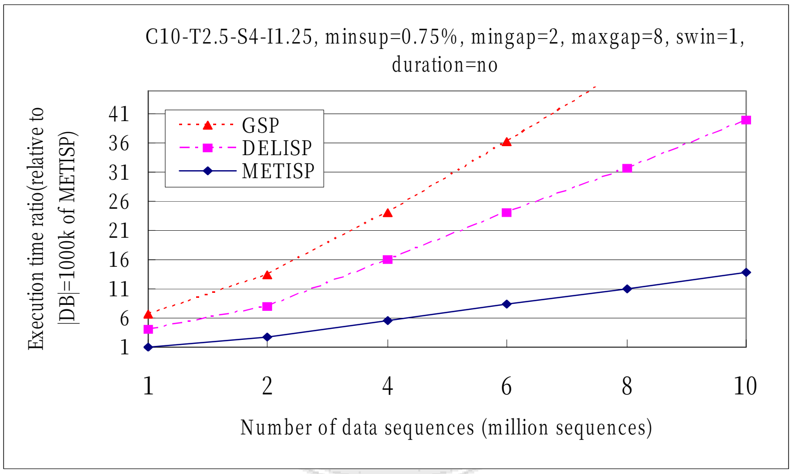


Figure 3.14 Linear scalability of the database size from 1000k to 10000k

Figures 3.15, 3.16, 3.17, 3.18, 3.19, and 3.20 show that the effect of METISP compared with GSP and DELISP using different database parameters. All experiments are run by specifying time constraint as 8 (maxgap), 2(mingap), 1(swin), and no constraint (duration) and varied by different minsup. We observe three algorithms executed in the figures 3.15 and 3.16 by changing single parameter |C|, in figures 3.16 and 3.17 by changing single parameter |T|, in Figures 3.16 and 3.18 by changing single parameter |S|, in figures 3.16 and 3.19 by changing single parameter |I|, in figures 3.16 and 3.20 by changing single parameter |N|. When the parameters change, METISP still have better performance than GSP and DELISP even with very low minsup threshold like 0.0035. The experimental results show that METISP is applicable and effective in various and multiple database.

The Figures 3.21 and 3.22 display the distribution of the length of the discovered patterns from different database. Table 3.4 gives a summary of the characteristics of the patterns. When minsup is set as 0.0035, the maximum length of mining patterns is 12 in DB3, and the maximum length of mining patterns is 9 in DB4. The results show that METISP can still have great performances even the length of mining patterns is very long.

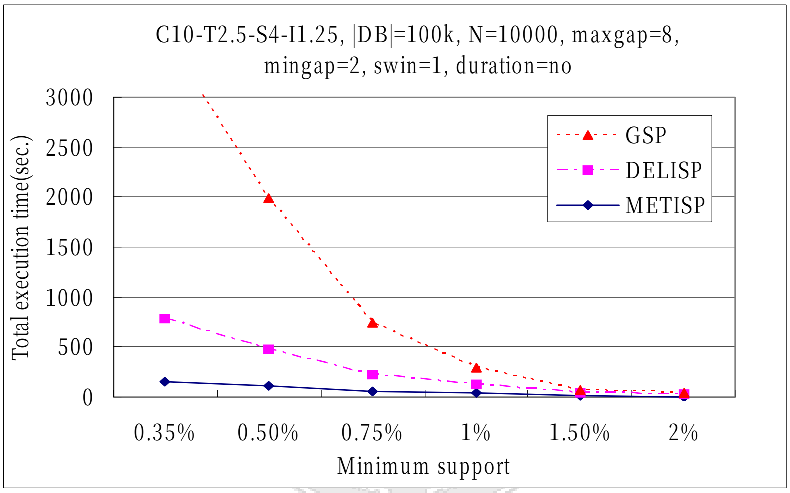


Figure 3.15 Effect of varying minimum support in DB1

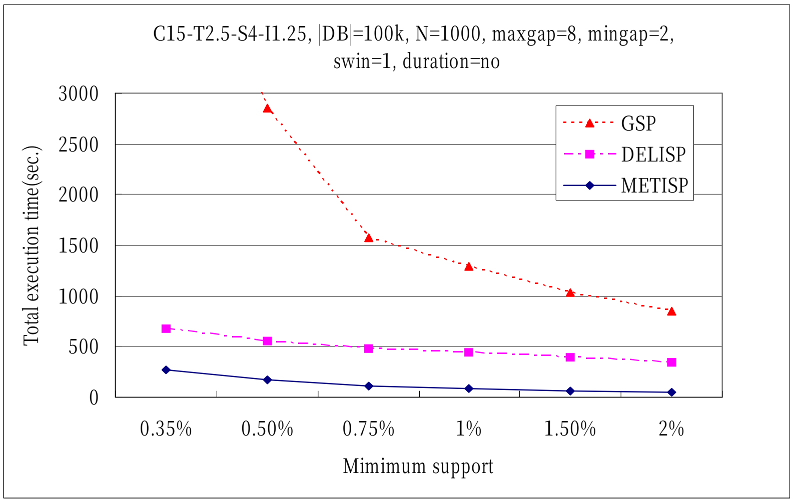


Figure 3.16 Effect of varying minimum support in DB2

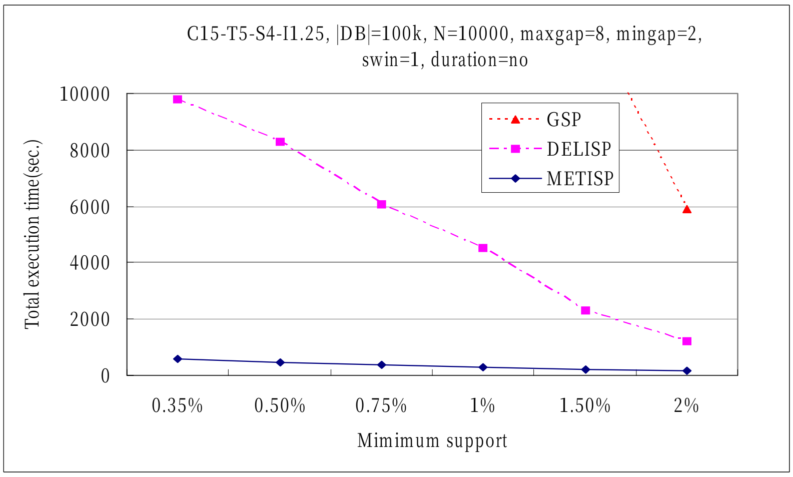


Figure 3.17 Effect of varying minimum support in DB3

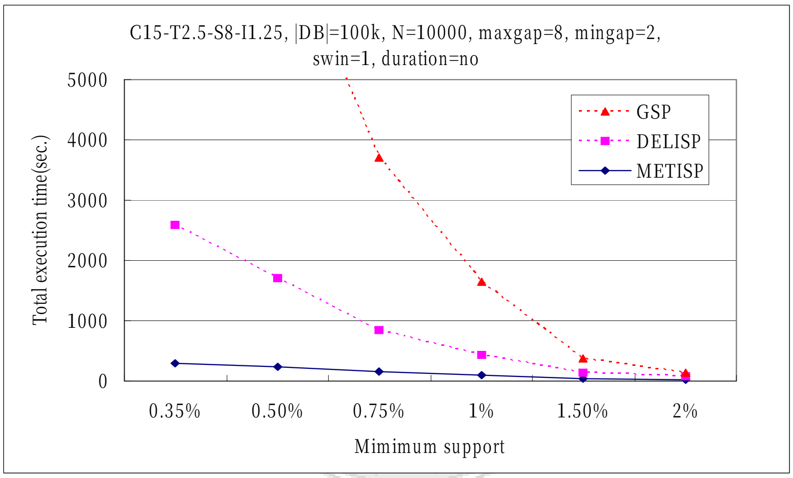


Figure 3.18 Effect of varying minimum support in DB4

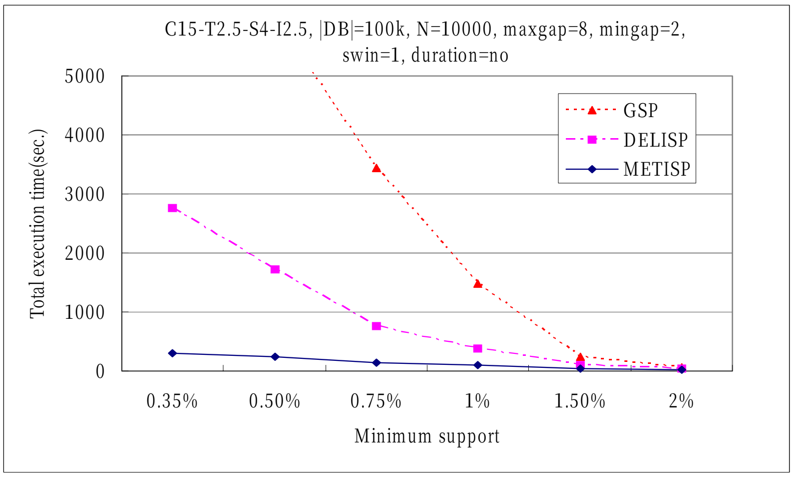


Figure 3.19 Effect of varying minimum support in DB5

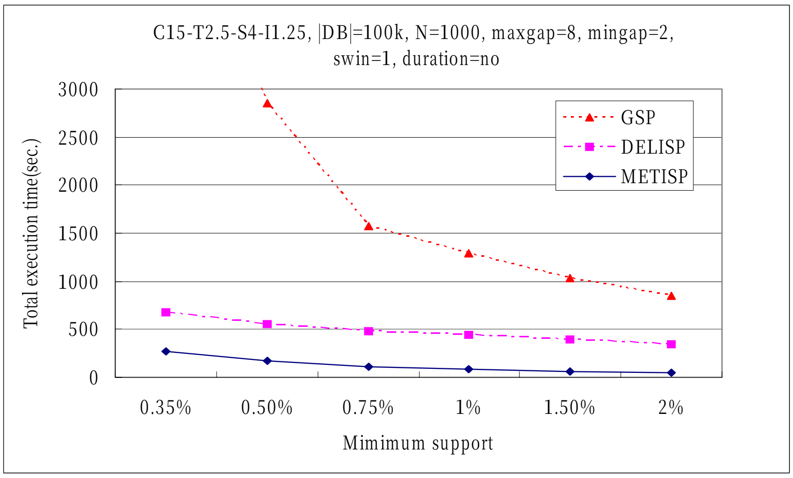


Figure 3.20 Effect of varying minimum support in DB6

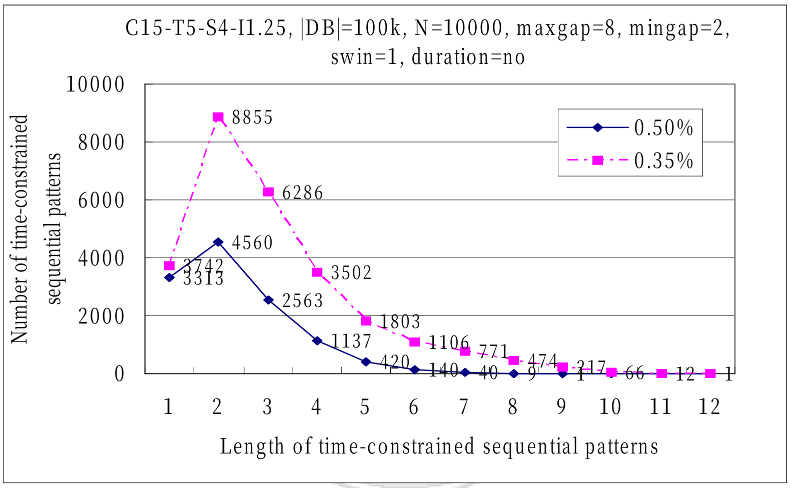


Figure 3.21 Length of time-constrained sequential patterns in DB3

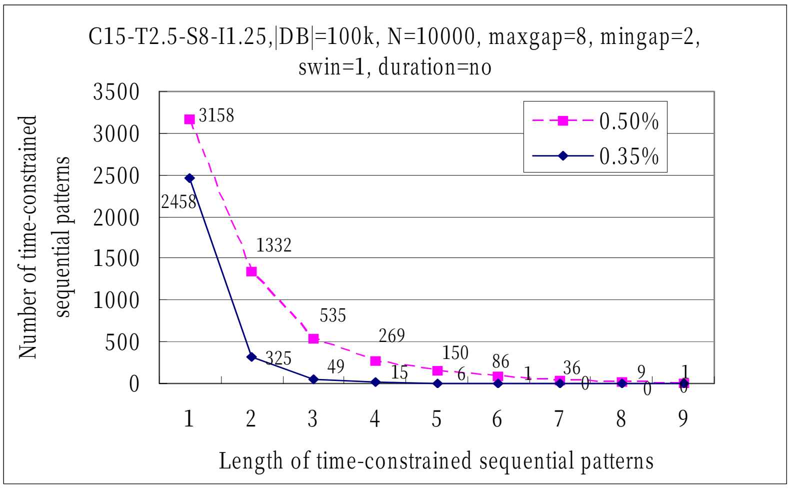


Figure 3.22 Length of time-constrained sequential patterns in DB4

Table 3.4 Characteristic of mining patterns and the execution time (METISP)

Database	Minsup	Max. length	Average length	Total number	Time(sec.)
DB3	0.35%	12	3.42	26835	578
	0.5%	9	2.29	12183	478
DB4	0.35%	9	1.98	5076	299
	0.5%	6	1.17	3345	232
DB5	0.35	8	2.01	8067	303
	0.5%	6	1.56	6440	231

3.9 Summary

In this thesis, we have presented METISP, an efficient memory time-indexing algorithm for mining sequential patterns with various time constraints, including minimum/maximum/exact gaps, sliding time-window, and duration. Using the pattern-growth approach together with the new memory time-index sets, METISP effectively shrinks the search space of potential patterns. The ever-increasing main memory enables the mining algorithm to push the time constraints deeply into the mining process. Considering that some extra-large databases still might not fit into the main memory, we also developed a partitioning approach utilizing the feature of METISP to overcome the limits of main memory for improved, time-constrained sequence mining. The comprehensive experiments also show that METISP is very efficient and outperforms both GSP and DELISP algorithms. Additionally, the experimental results show that METISP is applicable for various types of databases.

Chapter 4 CTSP: Closed Time-constrained Sequential Pattern mining

In this chapter, we propose a new algorithm, called CTSP, for closed time-constrained sequential pattern mining. CTSP uses the pattern-growth methodology [25, 18] to discover all time-constrained sequential patterns and check the closure by bi-direction verification strategy similar to BIDE [30]. The problem is formulated in Section 4.1. The Section 4.2 introduces the terminology used in CTSP. The Section 4.3 describes the overall of CTSP algorithm and Section 4.4 gives an example. The discussion of mining large databases is presented in Section 4.5. The experimental evaluations and results are reported in Sections 3.6 and in Section 3.7, respectively. Finally, Section 3.9 summarizes the chapter. For convenience, we refer to the sequence database as *DB* and a data sequence as *ds* in the following context.

4.1 Problem Statement

Let $\Psi = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ be a set of literals, called *items*. An *itemset* $I = (\beta_1, \beta_2, \ldots, \beta_q)$ is a nonempty set of q items such that $I \subseteq \Psi$. A *sequence s*, denoted by $<e_1e_2\ldots e_w>$, is an ordered list of w *elements* where each *element* e_i is an itemset. Without loss of generality, we assume the items in an element are in lexicographic order. The *length* of a sequence s, written as $|s|$, is the total number of items in all the elements in s. Sequence s is a *k-sequence* if $|s| = k$. The sequence database *DB* contains $|DB|$ data sequences. A *data sequence ds* has a unique identifier *sid* and is represented by $<{}_{t_1}e_1'\ {}_{t_2}e_2' \ldots {}_{t_n}e_n'>$, where element e_i' occurred at time t_i, $t_1 < t_2 < \ldots < t_n$.

A sequence s in the sequence database *DB* is a *time-constrained sequential pattern* (abbreviated as time-pattern) if $s.sup \geq minsup$, where $s.sup$ is the *support* of the sequence *s* and *minsup* is the user specified minimum support threshold. The *support* of sequence *s* is the number of data sequences *containing s* divided by |*DB*|. Note that the support calculation has to satisfy three time-constraints *maxgap* (maximum gap), *mingap* (minimum gap), and *swin* (sliding window). A data sequence $ds = <_{t_1}e_1{}'\ _{t_2}e_2{}'\ldots\ _{t_n}e_n{}'>$ *contains* a sequence $s = <e_1e_2\ldots e_w>$ if there exist integers l_1, u_1, l_2, u_2, …, l_w, u_w and $1 \leq l_1 \leq u_1 < l_2 \leq u_2 < \ldots < l_w \leq u_w \leq n$ such that the four conditions hold: (1) $e_i \subseteq (e_{l_i}{}' \cup \ldots \cup e_{u_i}{}')$, $1 \leq i \leq w$ (2) $t_{u_i} - t_{l_i} \leq swin$, $1 \leq i \leq w$ (3) $t_{u_i} - t_{l_{i-1}} \leq maxgap$, $2 \leq i \leq w$ (4) $t_{l_i} - t_{u_{i-1}} \geq mingap$, $2 \leq i \leq w$. Assume that t_j, *mingap*, *maxgap*, *swin* are all positive integers, *maxgap* ≥ *mingap* ≥ 1. When *mingap* is the same as *maxgap*, the time constraint is additionally called *exact gap*. Common sequential pattern mining without time constraints is a special case by setting *mingap* = 1, *maxgap* = ∞, *swin* = 0.

A sequence *s* is closed if no *contiguous supersequence* s_{sup} with the same support exists. Given a sequence $s = <e_1e_2\ldots e_w>$ and a subsequence s_{sub} of *s*, s_{sub} is a contiguous subsequence of *s* if s_{sub} can be obtained by any one of following ways: (1) dropping an item from either e_1 or e_w (2) dropping an item from element e_i ($1 \leq i \leq w$) which has least two items (3) s_{sub} is a contiguous subsequence of s_{sub}', and s_{sub}' is a contiguous subsequence of *s*. Additionally, *s* is called a *contiguous supersequence* of s_{sub}. A sequence *s* is a closed time-constrained sequential pattern, abbreviated as *closed time-pattern*, if it is a time-pattern and is closed. The mining aims to discover the set of all closed time-patterns.

Generally, all mined sequential patterns can be preserved as the form of closed patterns and can be deduced from the subsequences of closed patterns using the anti-monotone property. Unfortunately, not all subsequences of a closed time-pattern can hold all time constraints and appear in the mined results so that correct patterns can be derived afresh from closed ones. The new definition of closure ensures the accuracy and the completeness of all time-constrained sequential patterns. That means the all contiguous subsequences of a closed time-pattern are time-constrained sequential patterns.

Table 4.1 Sequence database (DB) and the closed time-patterns

Sid	Sequence	Closed time-constrained sequential pattern (*minsup*=50%, *mingap*=3, *maxgap*=15, *swin*=2)
C1	<$_3$(c)$_5$(a, f)$_{18}$(b)$_{31}$(a)$_{45}$(f)>	<(a)>:3, <(a, c)(b)>:2, <(b)>:3, <(b)(e)(d)>:2, <(c)>:3, <(c, d)>, <(d)>:3
C2	<$_6$(a, c)$_{10}$(b)$_{17}$(e)$_{24}$(c, d)>	
C3	<$_1$(b)$_{20}$(b, g)$_{27}$(e)$_{36}$(d)>	
C4	<$_5$(a)$_{10}$(d)$_{21}$(c, d)>	

Given a sequence database *DB* in Table 4.1, the rightmost column shows the closed time-constrained sequential patterns with constraints *mingap*=3, *maxgap*=15, *swin*=2 and *minsup*=50%. In Table 4.1, data sequence C4 <$_5$(a)$_{10}$(d)$_{21}$(c, d)> has three itemsets occurring at time 5, 10, 21, respectively. The third itemset of C4 has two items *c* and *d*. Sequences <(a, c)(b)> and <(b)(e)(d)> are both 3-sequences. The sequence <(a, c)(b)> is contained in data sequences C1 <$_3$(c)$_5$(a, f)$_{18}$(b)$_{31}$(a)$_{45}$(f)> because element (a, c) can be contained in the transaction combining $_3$(c) and $_5$(a, f) for 5-3≦2 (*swin*). Meanwhile, the mingap and maxgap constraints are satisfied for $18-5 \geq 3$ and $18-3 \leq 15$. Similarly, <(a, c)(b)> is contained in C2. The support of <(a, c)(b)> is 2/4 and it is a time-pattern for *minsup* = 50%. <(a, c)(b)> is also a closed time-pattern since it has no contiguous supersequence with the same support. <(c)(b)> is a time-pattern for <(c)(b)>.*sup* = 2/4 but not closed for its contiguous supersequence <(a,c)(b)> having the same support.

The notion of contiguous subsequence (supersequence) is introduced to guarantee the correctness and completeness of closed time-patterns. The subsequence of a (closed) time-pattern is not necessary a (closed) time-pattern if the contiguous relationship is not hold. For example, though <(b)(d)> is the subsequence of <(b)(e)(d)>, time-pattern <(b)(e)(d)> does not imply that <(b)(d)> is also a time-pattern since <(b)(d)> fails the maxgap constraint in C3. However, <(b)(e)(d)> of support 2/4 ensures that its contiguous subsequences, such as <(b)(e)> and <(e)(d)>, are also time-constrained sequential patterns with support 2/4. Such a definition enables the close time-patterns to have the same expressive power with more compact patterns.

4.2 Terminology Used in CTSP

Definition 1 (frequent item) An item x is called a frequent item in DB if <(x)>.*sup* ≥ *minsup*.

Definition 2: (type-1 pattern, type-2 pattern, stem, prefix) Given a frequent pattern *P* and a frequent item *x* in *DB*, *P'* is a type-1 pattern if it can be formed by adding an itemset of the single item *x* after the last element of *P*, and a type-2 pattern if formed by appending *x* to the last element of *P*. For type-2 pattern, the lexicographic order of *x* must be larger than all items in the element. The item *x* is called the stem of the new frequent pattern *P'*. The prefix pattern (abbreviated as prefix) of *P'* is *P*.

For example, <(a)(b)> is a type-1 pattern by adding (b) after <(a)>, and <(a, c)> is a type-2 pattern by appending (c) to <(a)>. Here, (b) and (c) are the stems and <(a)> is the prefix.

Definition 3: (last-start time, last-end time, time-index) Let the last element of a frequent pattern *P* be *LE*. If *ds* contains *P* by having $LE \subseteq e_{\gamma} \cup e_{\gamma+1} \cup \ldots \cup e_{\omega}$, where $e_{\gamma},\ldots, e_{\omega}$ are elements in *ds*, the occurring time t_{γ}, and t_{ω} for itemsets e_{γ} and e_{ω}, respectively, are named *last-start time* (abbreviated as **lst**) and *last-end time* (abbreviated as **let**) of *P* in *ds*. Every occurrence of timestamp lst:let is collected altogether as [lst_1:let_1, lst_2:let_2,…, lst_k:let_k], $lst_i \leq let_i$ for $1 \leq i \leq k$. Such a timestamp lists is called the *time index* of *P* in *ds*.

According to the definition, any contiguous supersequence P' of the frequent pattern P can be derived by appending an item to any element of P or adding an element of single item before first element or after last element of P. In addition to stems, items in the backward direction of a frequent pattern *P*, i.e. occurring before *P*, may be used to form a contiguous supersequence *P'*. For example, let *mingap*=3, *maxgap*=15, and *swin*=2, *P* = <(a)(b)> is contained in *s* = $<_2(e)_6(a, c)_{10}(b, d)_{18}(a)>$ with *time-index* [10:10]. Stem (a) (timestamp 18) may extend *P* to a type-1 pattern <(a)(b)(**a**)> and stem (d) (timestamp 10) may extend *P* to a type-2 pattern <(a)(b, **d**)> in the forward direction. Considering the (a) in *P*, item (e) (timestamp 2) and item (c) (timestamp 6) can be used to form contiguous supersequences <(a, **c**)(b)> and <(**e**)(a)(b)>, respectively. The two non-stem items are found in the backward direction of *P*. Thus, we have the following definition.

Definition 4: (extension item, extension period) Given a frequent pattern *P* contained in *ds*, a non-stem item in *ds* is called an *extension item* (abbreviated as **EI**) if it can be used in extending *P* to form a contiguous supersequence *P'* satisfying the time constraints. The time periods within which exists is called *extension period* (abbreviated as **EP**). The time period is referred to as *backward extension period* (abbreviated as **BEP**) and the item is called *backward extension item* (abbreviated as **BEI**). For convenience, we refer to the time period within which stems exist as *forward extension period* (abbreviated as **FEP**).

Lemma 1: Given a time-index of frequent pattern *P* in *ds* [lst_1:let_1, lst_2:let_2,…, lst_k:let_k], the FEP satisfies either one of the following conditions: (1) $\exists$ *i*, $1 \le i \le k$, let_i+mingap ≤ FEP ≤ lst_i+maxgap (2) $\exists$ *i*, $1 \le i \le k$, let_i−swin ≤ FEP ≤ lst_i+swin. Figure 4.1 illustrates Lemma 1.

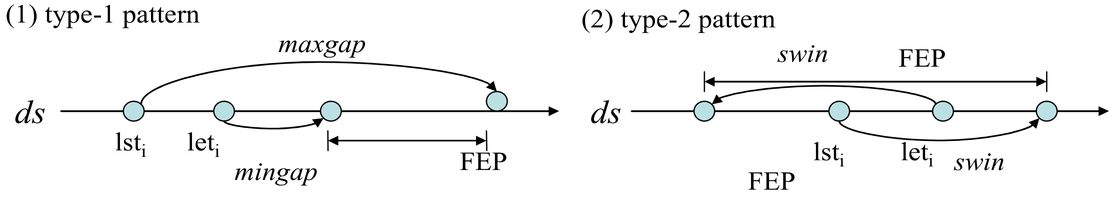


Figure 4.1 Forward extension periods for type-1 pattern and type-2 pattern

Definition 5: (timeline) Given a frequent patter P=< $e_1 \ldots e_r \ldots e_w$> contained in *d*s. If $e_1 \subseteq e_{st_1} \cup \ldots \cup e_{et_1}$, …, $e_r \subseteq e_{st_r} \cup \ldots \cup e_{et_r}$, ..., and $e_w \subseteq e_{st_w} \cup \ldots \cup e_{et_w}$, where e_{st_1}, ..., e_{st_w}, e_{et_1}, ..., e_{et_w} are elements in *ds*, the list of timestamps [st_1:et_1, …, st_r:et_r, …, st_w:et_w] where $st_r \leqq let_r$ for $1 \leqq r \leqq w$ is called the *timeline* of *P* in *ds*.

Note that the *ds* may have several timelines of *P*. For example, *P* = <(a)(d)(e)> can be found in *ds* = <$_5$(b)$_9$(a)$_{13}$(d)$_{14}$(c)$_{16}$(e)$_{19}$(e)> with timeline [9:9, 13:13, 16:16] and timeline [9:9, 13:13, 19:19]. The two timelines, implemented as linked lists, are shown in Figure 4.2.

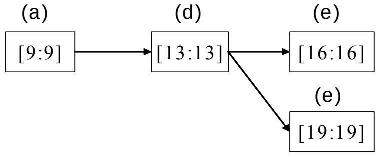
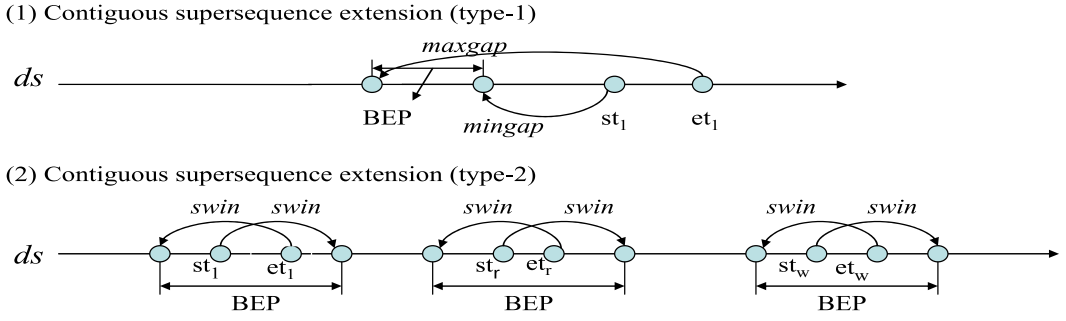


Figure 4 2 The timelines of <(a)(d)(e)> in <$_5$(b)$_9$(a)$_{13}$(d)$_{14}$(c)$_{16}$(e)$_{19}$(e)>

Lemma 2: Given a frequent pattern *P* = <$e_1 \ldots e_r \ldots e_w$> contained in *ds*. For each timeline [st_1:et_1,…,st_r;et_r,…,st_w:et_w] of *P* in *ds*, the BEP satisfies either one of the following conditions: (1) et_1-maxgap ≤ BEP ≤ st_1-mingap (2) ∃ *i*, $1 \le i \le w$, et_i−swin ≤ BEP ≤ st_i or et_i ≤ BEP ≤ st_i+swin. Figure 4.3 illustrates Lemma 2.


(1) Contiguous supersequence extension (type-1)

(2) Contiguous supersequence extension (type-2)

Figure 4.3 Backward extension periods of contiguous supersequences of P

Forming a type-2 pattern with the potential stems using Lemma 1 can be pruned in advance if the stem is lexicographically smaller than the items of the last element in *P*. Moreover, the time periods are checked on not violating the minimum/maximum gap constraints between adjacent elements when a type-2 pattern is formed. Similarly, forming contiguous supersequence type-2 extension using Lemma 2, the time periods are also checked on not violating the minimum/maximum gap constraints between the adjacent elements.

Lemma 3: Given a frequent pattern P=<e_1…e_r…e_w>, if a stem to be used in extending *P* to a contiguous supersequence *P'* is found in every data sequence containing *P*, then P' has the same support with P, and P is not a closed time-pattern.

Lemma 4: Given a frequent pattern P=<e_1...e_r...e_w>, if a BEI to be used in extending *P* to a contiguous super-sequence *P'* is found in every data sequence containing *P*, then P' has the same support with P, and P is not a closed time-pattern.

4.3 CTSP Algorithm: A Detailed Description

Figure 4.4 outlines the CTSP Algorithm, which mines patterns within the pattern-growth framework. The techniques used are similar to the pseudo projection version of PrefixSpan algorithm [25] and bi-direction closure checking in BIDE algorithm [30], while it can handle constraints minimum/maximum gaps and sliding time-window. Assume that the DB can fit into the main memory, CTSP first loads DB into memory (as MDB) and scans MDB once to find all frequent items. With respect to each frequent item, CTSP then constructs a time-index set for the 1-sequence item and recursively forms time-patterns of longer length. The time-index set is a set of (data-sequence pointer, time-index) pairs. Only those data sequences containing that item would be included. The time-index indicates the list of lst:let pairs as described in Definition 3.

```
Algorithm: CTSP
Input: DB (a sequence database), minsup (minimum support), mingap (minimum
gap), maxgap (maximum gap), swin (sliding time-window).
Output: the set of all closed time-constrained sequential patterns
1. load DB into memory (as MDB) and scan MDB once to find all frequent items.
2. for each frequent item x,
   (1) form the sequential pattern P = <(x)>
   (2) scan MDB once to construct P-Tidx, time index set of x.
   (3) call CMine(P, P-TIdx)
```

Figure 4.4 Algorithm CTSP

In Figure 4.5, CMine (P, P-Tidx) mines type-1 patterns and type-2 patterns having prefix P by effectively locating FEPs with Lemma 1. The P-Tidx is the time-index set for P. CTSP thus never search the data sequences irrelevant to P. Moreover, the FEPs ensure that CTSP locates and counts the effective stems which can form valid patterns, rather than the whole set of items in the data sequence. Furthermore, CTSP adopts forward and backward closure checking to examine the closure of prefix *P*. The supports of potential stems and BEIs are handled by forward closure checking and backward closure checking, respectively. The backward closure checking is done by backward(P, P-TIdx) shown in Figure 4.6. If neither stem nor BEI has the same support as *P*, then *P* is a desired closed time-pattern. Recursively, for a newly formed type-1 pattern or type-2 pattern P’, its time-index set P’-Tidx is constructed and CMine(P’, P’-Tidx) is invoked. By pushing time attributes deeply into the mining process, CTSP efficiently discovers the desired patterns.

```
Subroutine: CMine (P, P-Tidx)
Parameter: P = prefix with support count, P-Tidx=time-index set
1. for each data sequence ds in the P-DB, // P-DB: sequences indicated in P-Tidx
  (1) use Lemma 1 to collect the FEPs of type-1 and type-2 patterns, respectively.
  (2) for each item in the FEPs of type-1 and type-2 patterns, add one to its support
       count, respectively.
2. if any support count of a stem is equal to the support count of P, then P is not closed.
   Otherwise output P if Backward(P, P-Tidx) return “closed”.
3. for each item x’ found in the FEPs of type-1 pattern and $<(x)>.sup \geq minsup$,
  (1) form the type-1 pattern P’ by extending stem x’.
  (2) use Lemma 1 and the FEPs of each ds in P-DB to construct P’-Tidx, time-index
       set of x’.
  (3) call CMine(P’, P’-Tidx);
4. for each item x’ found in the FEPs of type-2 pattern and $<(x)>.sup \geq minsup$,,
  (1) form the type-2 pattern P’ by appending stem x’.
  (2) use Lemma 2 and the FEPs of each ds in P-DB to construct P’-Tidx, time-index
       set of x’.
  (3) call CMine(P’, P’-Tidx);
```

Figure 4.5 Subroutine CMine() of CTSP algorithm

```
Subroutine: Backward (P, P-Tidx)
Parameter: prefix P = <$e_1…e_r…e_w$>, P-Tidx = time-index set
1. for each element $e_j$ in P, 1≦r≦w
  (1) for each data sequence ds in the P-DB, // P-DB: sequences indicated in P-Tidx
      1.1 find the BEPs of $e_j$ in ds
      1.2 add one to the support count of the BEIs
  (2) if any support count of a BEI is equal to the support count of P, return "not
      closed"
2. return "closed"
```

Figure 4.6 Subroutine Backward() of CTSP algorithm

4.4 An Illustrating Example of Mining Patterns Using CTSP

Example 1: Given the *DB* in Table 4.1, *minsup*=50%, *mingap*=3, *maxgap*=15, *swin*=2, CTSP mines closed time-patterns by the following steps:

Step 1: Load DB into memory and find all frequent items

CTSP first reads the DB into memory and scans the in-memory DB (abbreviated as MDB) once to find frequent items. <(a)>:3, <(b)>:3, <(c)>:3, <(d)>:3, <(e)>:2 are found, and <(a)>:3 shows its support count is 3.

Step 2: Construct the time-index set of the sequential pattern

Take <(a)> for example, CTSP scans MDB and constructs the time-index set <(a)>-Tidx, as shown in Figure 5(a). Three pointers appear in <(a)>-Tidx for <(a)> appears in C1, C2, and C4. For C1 $<_{3}(c)_{5}(\mathbf{a}, f)_{18}(b)_{31}(\mathbf{a})_{45}(f)>$, element (a) occurs at time 5 and 31 so the time-index is marked as [**5:5**, **31:31**]. The indexes for C2 and C4 are processed accordingly. The <(a)>-Tidx is used in subroutine CMine, which searches potential type-1 and type-2 stems.

Step 3: Find stems in FEPs from the time-index set

Subroutine CMine computes the supports of potential stems here. With respect to prefix <(a)>, the type-1 FEPs for C1 is [8:20, 34:46] from [lst_1+mingap:let_1+maxgap, lst_2+mingap:let_2+maxgap], where $lst_1 = let_1 = 5$ and $lst_2 = let_2 = 31$. The FEPs of C2 [9:21] and C4 [8:20] are obtained correspondingly. Now, the support of item *b* passes the threshold to be a stem. Similarly, the type-2 FEPs for C1 is [3:7, 29:33], for C2 is [4:8], and for C4 is [3:7]. Items *c* then has enough support to be a stem.

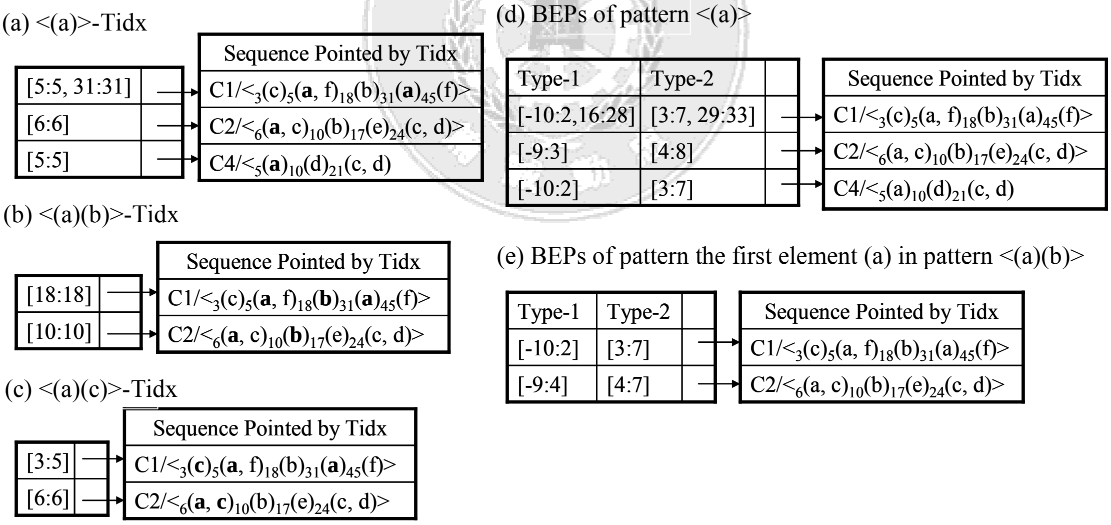
Step 4: Check the closure of sequential pattern

Prefix <(a)> has support 3 but the support of stems c (and b) is 2, so CTSP calls subroutine Backward to count the supports of potential BEIs. The BEPs corresponding to (a) are shown in Figure 5(d). No BEI is found to satisfy the time constrains. Consequently, <(a)>:3 is outputted as a closed time-pattern. Notably, the item c is not a BEI because its order is larger than item a, though c is discovered in the BEPs of element (a) for C1 and C2 using type-2 backward extension shown in Lemma 2.

Step 5: Recursively grow (discover) pattern

For each sequential pattern *P'* with prefix *P*, CTSP scans each data sequence containing *P* and records all the last-start time (*lst*) and last-end time (*let*) of P'. CTSP creates pointers pointing to data sequences containing P' simultaneously. Thus, <(a)(b)>-Tidx and <(a, c)>-Tidx are constructed, and CMine are recursively called. Figure 5(b) and 5(c) show <(a)(b)>-Tidx and <(a, c)>-Tidx, respectively. Considering type-1 FEPs of <(a)(b)>, which are [21:33] for C1 and [13:25] for C2, no stem can be found so that no pattern of prefix <(a)(b)> is formed. Similarly, the type-2 FEPs from [let_1−swin:lst_1+swin] for C1 and C2 are [16:20] and [8:12], respectively. No pattern of prefix <(a)(b)> can be formed. Subroutine Backward is then called to check the closure of <(a)(b)>.

The *timelines* of prefix <(a)(b)> are [5:5, 18:18] for C1 and [6:6, 10:10] for C2. By using the Lemma 2**,** the BEPs corresponding to type-2 extension of element (b) are [16:20] for C1 and [9:12] for C2. The time period for C1 is [9:12] but not [8:10], since the time period must not violate the minimum/maximum gap between element (a) and (b). No items are found to extend element (b). Similarly, the BEPs corresponding to type-2 extension and type-1 extension of element (a) can be found as shown in Figure 5(e). The item c appears in both BEPs of C1 and C2 (type-2 extension) so that <(a)(b)> is not a closed time-pattern. That is, <(a)(b)>'s contiguous supersequence <(a, c)(b)> has the same support. Since <(a)(b)> is processed, CMine then continues on the type-1 and type-2 stems of <(a, c)>.



(a) <(a)>-Tidx

	Sequence Pointed by Tidx
[5:5, 31:31]	C1/<$_{3}$(c)$_{5}$(**a**, f)$_{18}$(b)$_{31}$(**a**)$_{45}$(f)>
[6:6]	C2/<$_{6}$(**a**, c)$_{10}$(b)$_{17}$(e)$_{24}$(c, d)>
[5:5]	C4/<$_{5}$(**a**)$_{10}$(d)$_{21}$(c, d)

(b) <(a)(b)>-Tidx

	Sequence Pointed by Tidx
[18:18]	C1/<$_{3}$(c)$_{5}$(**a**, f)$_{18}$(**b**)$_{31}$(**a**)$_{45}$(f)>
[10:10]	C2/<$_{6}$(**a**, c)$_{10}$(**b**)$_{17}$(e)$_{24}$(c, d)>

(c) <(a)(c)>-Tidx

	Sequence Pointed by Tidx
[3:5]	C1/<$_{3}$(**c**)$_{5}$(**a**, f)$_{18}$(b)$_{31}$(**a**)$_{45}$(f)>
[6:6]	C2/<$_{6}$(**a**, **c**)$_{10}$(b)$_{17}$(e)$_{24}$(c, d)>

(d) BEPs of pattern <(a)>

Type-1	Type-2	Sequence Pointed by Tidx
[-10:2,16:28]	[3:7, 29:33]	C1/<$_{3}$(c)$_{5}$(a, f)$_{18}$(b)$_{31}$(a)$_{45}$(f)>
[-9:3]	[4:8]	C2/<$_{6}$(a, c)$_{10}$(b)$_{17}$(e)$_{24}$(c, d)>
[-10:2]	[3:7]	C4/<$_{5}$(a)$_{10}$(d)$_{21}$(c, d)

(e) BEPs of pattern the first element (a) in pattern <(a)(b)>

Type-1	Type-2	Sequence Pointed by Tidx
[-10:2]	[3:7]	C1/<$_{3}$(c)$_{5}$(a, f)$_{18}$(b)$_{31}$(a)$_{45}$(f)>
[-9:4]	[4:7]	C2/<$_{6}$(a, c)$_{10}$(b)$_{17}$(e)$_{24}$(c, d)>

Figure 4.7 Time index set and BEPs of patterns

<(a, c)> is not closed since stem *b* having the same support of 2 is found so that the process need not to call backward closure checking. The process continues to mine with prefix <(a, c)(b)> and output closed pattern <(a, c)(b)> after bi-dimensional closure checking. However, no more pattern is formed and the mining of prefix <(a)> now stops.

CTSP then applies steps 2 to 5 on <(b)>, <(c)>, <(d)>, and <(e)>. The complete set of closed time-patterns is listed in the rightmost column in Table 4.1.

4.5 Using CTSP for Mining Large Databases

CTSP uses proposed memory-indexing method so that algorithm must load the complete database into memory. For the database which does not fit into main memory, a projected database method based on all frequent 1-sequence can project the original database to small sub-database. Each sub-database is consisted of all complete sequences which contain the frequent 1-sequence. Therefore, each sub-database can be regarded as independent and apply the CPST algorithm to discover all closed patterns with the frequent 1-sequentce as prefix in each sub-database.

Moreover, in [11], a parallel algorithm Par-CSP which is based on BIDE algorithm have been developed to efficiently mine closed sequential patterns for large databases on a distribution memory system. CTSP algorithm extends the BIDE algorithm with time constraints also is suitable for the framework of Par-CSP algorithm. Therefore, CTSP can also overall the limitation of memory by using parallel method on a distribution memory system.

4.6 Experimental Evaluations

Extensive experiments were performed on the synthetic data and real data to assess the performance of the CTSP algorithm. Algorithms GSP and DELISP, which mine all time-constrained sequential patterns without closure checking, were used to compare with CTSP. The synthetic data is generated using the IBM synthetic dataset generation program [1]. All experiments are run performed on AMD 2800+ PC with 1GB memory running the Windows XP. Here, we describe the result of dataset

C10-T2.5-S4-I1.25 having 100000 data sequences (|DB| = 100k), with Ns=5000, N_I=25000 and N=10000. The detailed parameters are addressed in [2]. The results of varying |C|, |T|, |S|, and |I| were consistent.

The real database Gazelle is a clickstream data from Gazelle.com and can be derived from KDD Cup 2000. We pre-process the Gazelle data by using the CookieID as the customer id to identify diverse customer sequences. We use ProductID as the unique item code and utilize SessionID to distinguish whether the items are within the same itemsets or not. The request time is transformed into appropriate form for mining. The Table 4.2 gives the detail information of the Gazelle data. Additionally, the maximum sequence length and maximum session size are 628 and 267, respectively. The maximal session number in a data sequence is 140. More information can be found in [16].

Table 4.2 The parameters of the Gazelle database

Parameter	Value
Number of data sequences (|DB|)	29369
Number of total sessions (Number of total itemsets)	35722
Number of page-views (Number of total items)	85374
Number of different items (N)	386
Average length of data sequences	2.9
Average session number in each data sequence	1.21
Average item number in each session	2.38

4.7 Performance Results

The Figures 4.8, 4.9 and 4.10 show the results of varying mingap, maxgap and sliding window constraints on synthetic database C10T2.5S4I1.25N10 with 100k data sequences, respectively. The minsup is fixed to 0.5%. No other constraints were set in the experiments so as to observe the effect of the single constraint. The number of closed patterns decreases as mingap increases or maxgap decreases. The gap constraints restrict more patterns so that the running time is decreased. The *swin* relaxes the constraint and allows more patterns to appear so that total execution time is increased. The effect of varying minsup on the same database from 0.35% to 1% is shown in Figure 4.11. The result of scaling up databases, as in depicted in Figure 4.16, indicated that CTSP has good linear scalability.

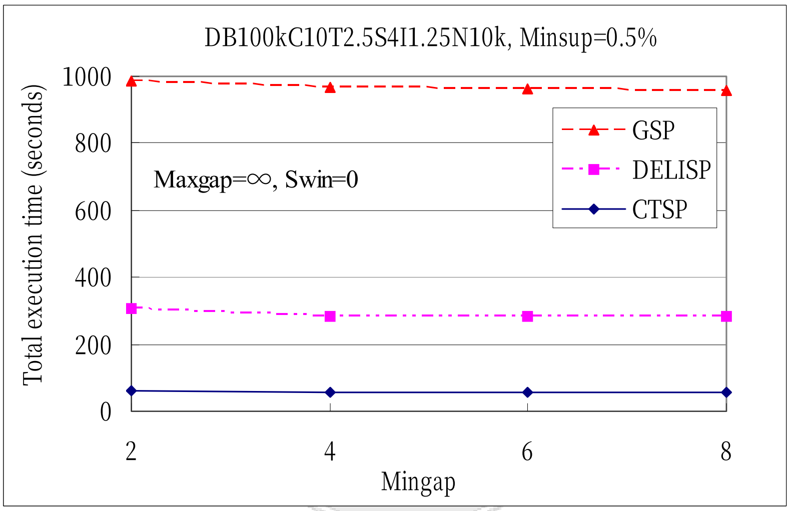


Figure 4.8 Effect of CTSP vs. GSP and DELISP by varying minimum gap

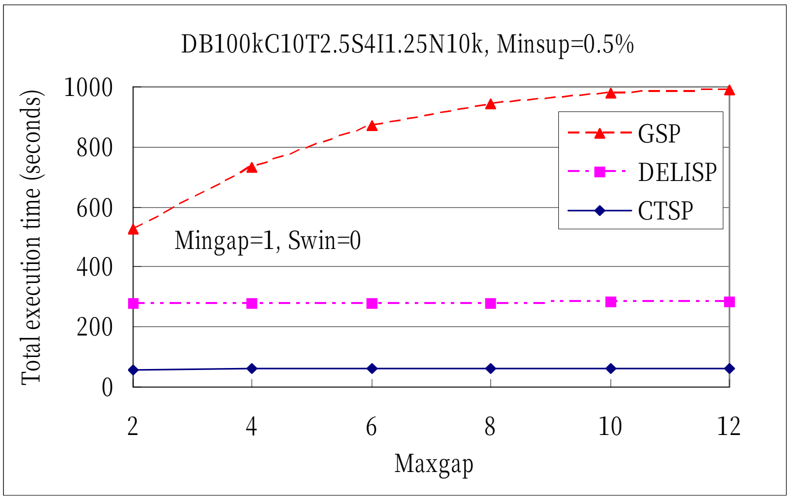


Figure 4.9 Effect of CTSP vs. GSP and DELISP by varying maximum gap

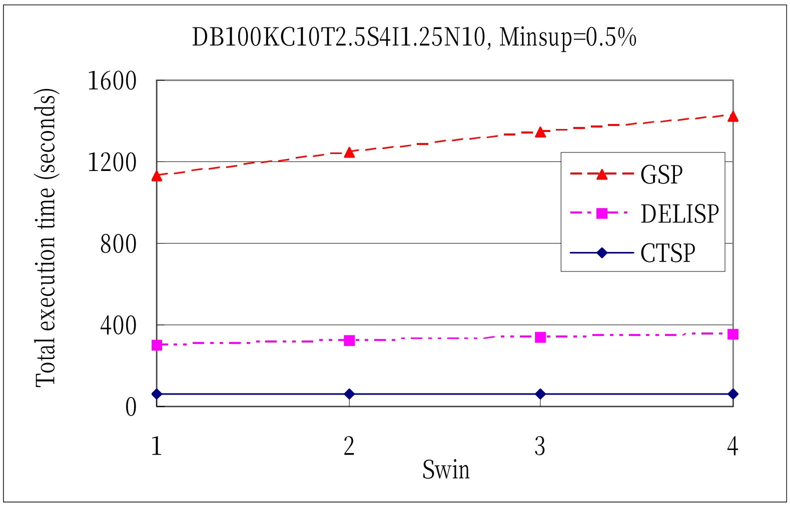


Figure 4.10 Effect of CTSP vs. GSP and DELISP by varying sliding window

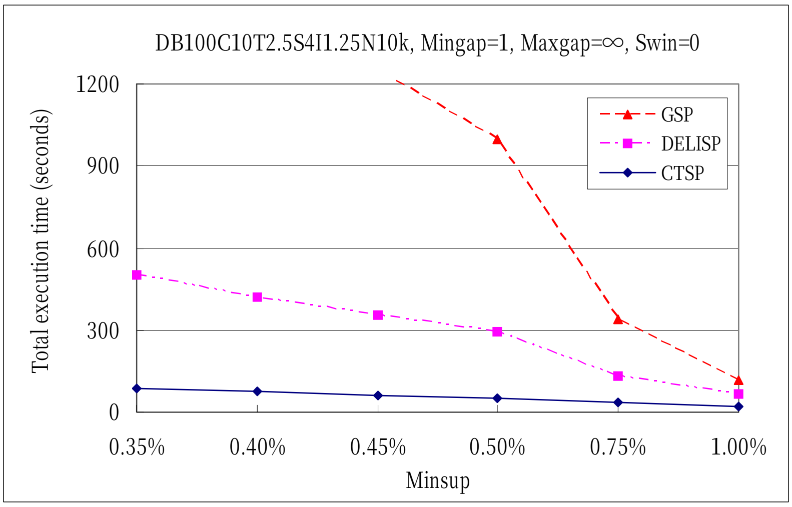


Figure 4.11 Effect of CTSP vs. GSP and DELISP by varying minimum support

Two experiments are done by varying minsup from 0.04% to 0.06% on the Gazelle database. In Figure 4.12, no constraints are set and the run time of CTSP is only 20 seconds even if the minsup is low to 0.04%. The Figure 4.13 indicates the distribution of the discovered closed time-patterns and the maximum length is up to 14. In Figure 4.14, the mingap, maxgap and sliding window are set as 3, 15 and 1 respectively. Relatively, Figure 4.15 shows the distribution of the discovered closed time-patterns. Moreover, Table 4.3 lists the number of total discovered time-patterns and time-patterns. The Gazelle is a dense database so that the number of discovered time-pattern is huge. In Table 4.3, the discovered closed time-patterns are meaningful to display compact results of the discovered time-patterns. The experiment results also show that CTSP has better performance than GSP and DELISP in both synthetic and real databases.

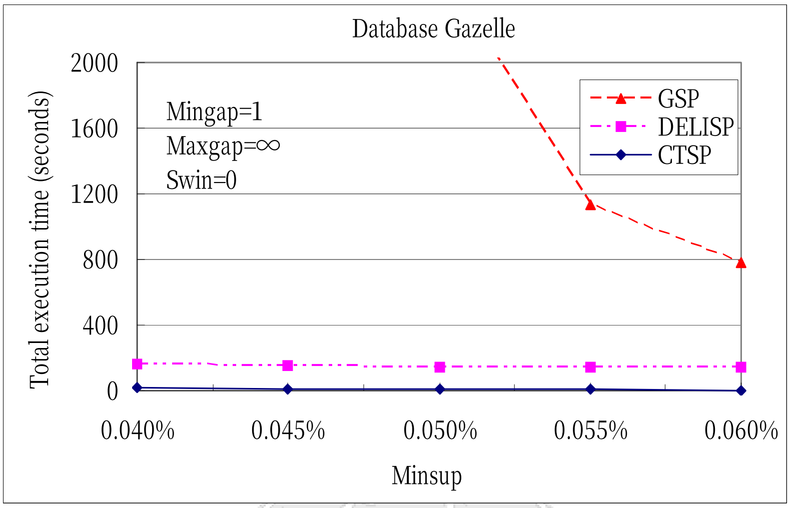


Figure 4.12 Effect of CTSP without time constraints in Gazelle database

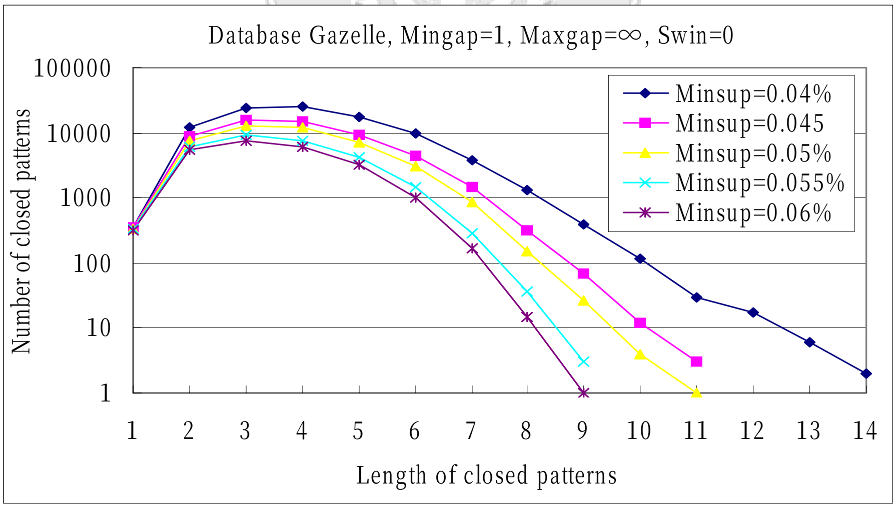


Figure 4.13 Distribution of the discovered patterns without time constraints

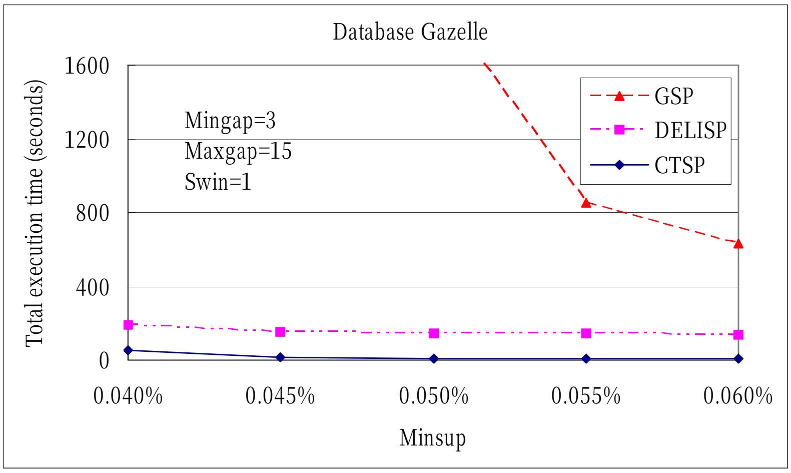


Figure 4.14 Effect of CTSP with time constraints in Gazelle database

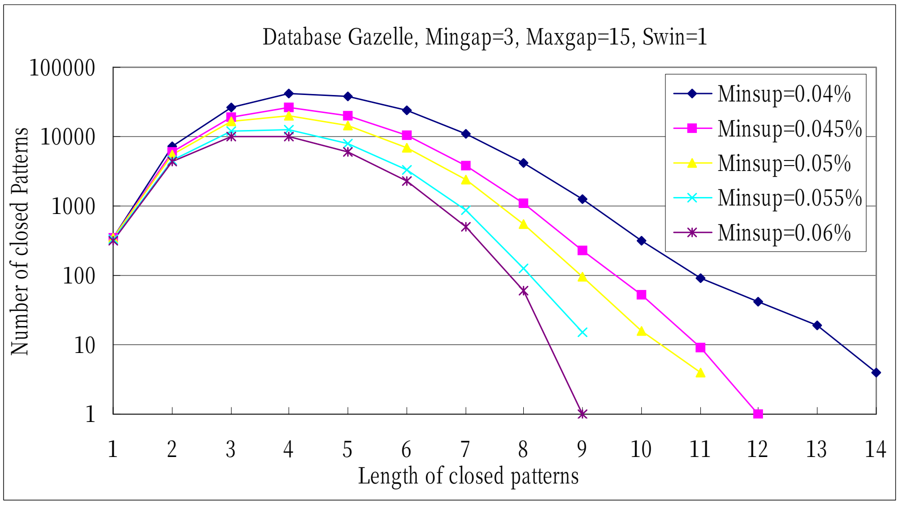


Figure 4.15 Distribution of the discovered patterns with time constraints

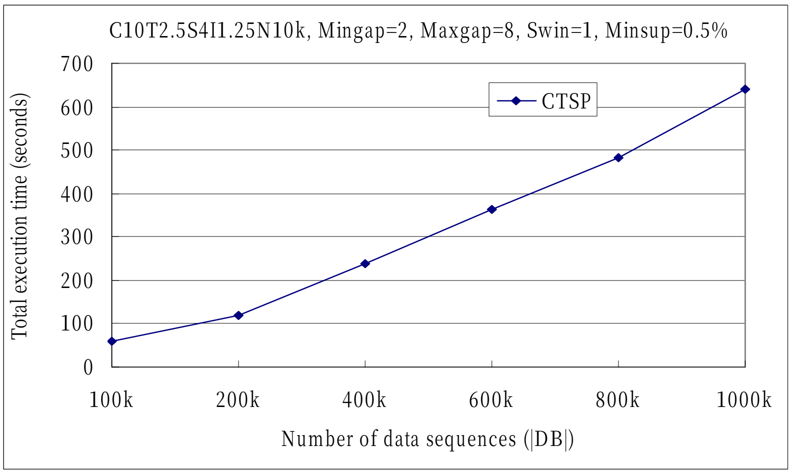


Figure 4.16 Linear scalability of the database size

Table 4.3 Number of total closed time-patterns vs. time-patterns

Minsup	0.04%	0.045%	0.05%	0.055%	0.06%
Time-pattern	515497	153212	101222	53164	40571
Closed Time-pattern	154724	87095	66960	41855	33898

4.8 Summary

In this thesis, we have presented a new definition and a new algorithm CTSP for mining closed time-patterns with minimum/maximum gap and sliding window constraints. CTSP utilizes the time constraints to shrinks the search-space effectively within the pattern-growth framework. The closure checking by handing time constraints is bi-directionally performed by. The experimental results show that CTSP has good performance than GSP and DELISP with gap constraints in both synthetic data and real data.

Chapter 5 Conclusions and Future Work

5.1 Contributions

In this thesis, we have proposed two novel algorithms METISP and CTSP for mining time-constrained sequential patterns and closed time-constrained sequential patterns. The contributions of METISP algorithm are indicated below:

(1)Handle composite time constraints at the same time: Increasing the accuracy of mining results is becoming an important topic. Time constraint is one of many influential constraints and must be considered within mining process. Hence, handling composite time constraints at the same time is difficult because time constraint estimations may affect each other. However, METISP can mine sequential patterns by specifying minimum/maximum/exact gap, sliding window and duration constraints which are not considered at the same time in previous algorithms.

(2)Improve the efficiency of algorithms for mining time-constrained sequential patterns: Although METISP handles more time constraints than previous algorithms. Nevertheless, METISP uses memory-indexing technique and utilizes the inherent properties of time constraints to avoid unnecessary searching space and decreases the execution time of mining patterns. To employ the main memory and CPU efficiently helps to discover desired patterns fast. The experimental results of varying different time constraints show METISP has better performance than GSP and DELISP algorithms.

(3)Successfully using partitioned technique for mining sequential patterns with time constraints: For the extra-large database, database projection and database partition are both proposed in previous algorithms for speeding up the mining. The database projection technique may generate many databases that amount to multiple size of original database. In comparison, the database partition technique can discover all

patterns by scanning original database, at most twice, without generating any sub-databases. The METSIP is the first algorithm which successfully utilizes partition-and-verification strategy to mine time-constrained sequential patterns on an extra-large database.

Next, we indicate the contributions of CTSP algorithm below:

(1)New definition of closure for mining closed time-patterns: Many algorithms have proposed to discover closed patterns but the definition of closed patterns in previous studies is not suitable for the time-constrained patterns. Hence, we propose a new definition of closure based on contiguous supersequences to ensure the accuracy and completeness of discovered closed time-constrained sequential patterns. Moreover, the new definition can be applied on the algorithms with minimum/maximum/exact gap, sliding window and duration constraints.

(2)Successful mining of closed time-patterns: To our knowledge, no algorithms are proposed for mining closed sequential patterns with time constraints because of high complicated problem. CTSP is the first algorithm proposed to mine closed time-constrained sequential patterns. The bi-directional closure checking with time constraints ensures the correctness of mining results. The experimental results also show that CTSP algorithm performs well than GSP algorithm in both synthetic and real databases.

5.2 Future Work

In comparison to traditional database mining, it is more challenging to mine patterns over data streams [29], since the characteristics of data streams bring many new restrictions. For example, there may be not enough time to deal with entire data which arrive quickly or there may be not enough memory to store the unbounded data which come through the Internet when the time goes by. For the reason, the past studies only deal with the problem of mining frequent itemsets over data streams. Obviously, it is more difficult for the problem of mining sequential patterns over data stream because the number combination of various temporal relationships is enormous. Recently, some researches [6, 9, 17, 23] have the ideas of stretching the field of mining itemsets to the field of mining sequential patterns over data streams. In our future work, we also expect to develop algorithms to solve the problem.

References

[1] R. Agrawal, and R. Srikant, “Fast Algorithms for Mining Association Rules in Large Databases,” *Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 487-499, 1994.

[2] R. Agrawal, and R. Srikant, “Mining Sequential Patterns,” *Proceedings of the 11th International Conference on Data Engineering*, pp. 3-14, Taipei, Taiwan,1995.

[3] R. Agrawal, and R. Srikant, “Mining Sequential Patterns: Generalizations and Performance Improvements,” *Proceedings of the 5th International Conference on Extending Database Technology*, pp. 3-17, Avignon, France, 1996.

[4] C. Antunes, and A.L. Oliveira, “Sequential Patterns Mining Algorithms: Trade-off between Speed and Memory,” *Proceedings Of Second Intfl Workshop on Mining Graphs, Trees and Sequences (MGTS 2004)*, Pisa, Italy, 2004.

[5] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu, “Sequential PAttern Mining using A Bitmap Representation,” *Proceedings of the 8th International Conference on Knowledge Discovery and Data Mining*, pp. 429-435, 2002.

[6] J. H. Chang, and W. S. Lee, “Efficient mining method for retrieving sequential patterns over online data streams,” *Journal of Information Science*, Volume 31, Issue 5, pp. 420-432, Oct. 2005.

[7] Y. L. Chen, M. C. Chiang, and M. T. Kao, “Discovering time-interval sequential patterns in sequence databases,” *Expert Systems with Applications*, Volume 25, Issue 3, pp. 343-354, Oct. 2003.

[8] Y. L. Chen, T. C. K. Huang, “Discovering fuzzy time-interval sequential patterns in sequence databases,” *IEEE Transactions on Systems, Man, and Cybernetics,* Part B, Volume 35, Number 5, pp. 959-972, Feb. 2005.

[9] G. Chen, X. Wu, and X. Zhu, “Sequential Pattern Mining in Multiple Streams,”

Proceedings of the 5th IEEE International Conference on Data Mining, pp. 585-588, Nov. 2005.

[10] D. Y. Chiu, Y. H. Wu, and A. L. P. Chen, "An Efficient Algorithm for Mining Frequent Sequences by a New Strategy without Support Counting," *Proceedings of the 20th International Conference on Data Engineering*, pp. 375-386, 2004.

[11] S. Cong, J. Han, and D. A. Padua, "Parallel mining of closed sequential patterns," *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* pp. 562-567, Chicago, Illinois, USA, Aug. 2005.

[12] F. M. Facca, and P. L. Lanzi, "Mining interesting knowledge from weblogs: a survey," *Data & Knowledge Engineering*, Volume 53, Number 3, pp. 225-241, Jun. 2005.

[13] M. N. Garofalakis, R. Rastogi, and K. Shim, "SPIRIT: Sequential Pattern Mining with Regular Expression Constraints," *Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 223-234, Edinburgh, Scotland, Sep. 1999.

[14] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," *Data Mining and Knowledge Discovery*, Volume 8, Issue 1, pp. 53-87, 2004.

[15] Ben. Kao, M. Zhang, C. L. Yip, D. W. Cheung, and U. M. Fayyad, "Efficient Algorithms for Mining and Incremental Update of Maximal Frequent Sequences," *Data Mining and Knowledge Discovery*, Volume 10, Number 2, pp. 87-116, Mar. 2005.

[16] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng, "KDD-Cup 2000 organizers' report: Peeling the onion," *SIGKDD Explorations*, pp. 86-98, 2000.

[17] H.-C. Kum, J. Pei, W. Wang, and D. Duncan, "ApproxMAP: Approximate

Mining of Consensus Sequential Patterns," *Proceedings of the Third SIAM International Conference on Data Mining*, San Francisco, CA, USA, May 2003.

[18] M. Y. Lin, and S. Y. Lee, "Fast Discovery of Sequential Patterns through Memory Indexing and Database Partitioning," *Journal of Information Science and Engineering*, Volume. 21, Number 1, pp. 109-128, Jan. 2005.

[19] M. Y. Lin, and S. Y. Lee, "Efficient Mining of Sequential Patterns with Time Constraints by Delimited Pattern-Growth," *Knowledge and Information Systems*, Volume 7, Issue 4, pp. 499-514, May 2005.

[20] C. Luo, and S. M. Chung, "A Scalable Algorithm for Mining Maximal Frequent Sequences Using Sampling," *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence*, pp. 156-165, Nov. 2004.

[21] F. Masseglia, F. Cathala, and P. Poncelet, "The PSP Approach for Mining Sequential Patterns," *Proceedings of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery*, pp. 176-184, Nantes, France, 1998.

[22] F. Masseglia, P. Poncelet, and M. Teisseire, "Pre-Processing Time Constraints for Efficiently Mining Generalized Sequential Patterns," *Proceedings of the 11th International Symposium on Temporal Representation and Reasoning,* pp. 87-95, France, 2004.

[23] A. Marascu, and F. Masseglia, "Mining Sequential Patterns from Temporal Streaming Data," *Proceedings of the first ECML/PKDD workshop on Mining Spatio-Temporal Data*, Porto, Portugal, Oct. 2005.

[24] S. Orlando, R. Perego, and C. Silvestri, "A new algorithm for gap constrained sequence mining," *Proceedings of the 2004 ACM Symposium on Applied Computing*, pp. 540-547, Nicosia, Cyprus, 2004.

[25] J. Pei, J. Han, B. Moryazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu, "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern

Growth," *Proceedings of the 17th International Conference on Data Engineering,* pp. 215-224, Heidelberg, Germany, Apr. 2001.

[26] J. Pei, J. Han, and W. Wang, "Mining sequential patterns with constraints in large databases," *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, pp. 18-25, 2002.

[27] T. Petre, Xifeng. Yan, and J. Han, "TSP: Mining top-k closed sequential patterns," *Knowledge and Information Systems*, Volume 7, Issue 4, pp. 438-457, May 2005.

[28] M. Seno, and G. Karypis, "SLPMiner: An Algorithm for Finding Frequent Sequential patterns Using Length-Decreasing Support Constraint," *Proceedings of the 2002 IEEE International Conference on Data Mining*, pp. 418-425, 2002.

[29] W. -G.. Teng, M.-S. Chen, and P. S. Yu, "A Regression-based Temporal Pattern Mining Scheme for Data Streams," *Proceedings of the 29th International Conference on Very Large Data Bases*, pp. 93-104**,** Berlin, Germany, Sep. 2003.

[30] J. Wang, and J. Han, "BIDE: Efficient Mining of Frequent Closed Sequences," *Proceedings of the 20th International Conference on Data Engineering*, pp. 79-90, Boston, Massachusetts, Mar. 2004.

[31] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Databases," *Proceedings of the Third SIAM International Conference on Data Mining*, San Francisco, CA, USA, May 2003.

[32] C. C. Yu, and Y. L. Chen, "Mining sequential patterns from multi-dimensional sequence data," *IEEE Trans on Knowledge and Data Engineering*, Volume 17, Issue 1, pp. 136-140, Jan. 2005.

[33] M. J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning Journal*, Volume 42, Issue 1-2, pp. 31-60, Jan. 2001.

[34] M. J. Zaki, "Sequence Mining in Categorical Domains: Incorporating

Constraints," *Proceedings of the 9th International Conference on Information and Knowledge Management*, pp. 422-429, Nov. 2000.