An Efficient Algorithm for Mining Time Interval-based Patterns in Large Databases

Yi-Cheng Chen, Ji-Chiang Jiang, Wen-Chih Peng and Suh-Yin Lee
Department of Computer Science
National Chiao Tung University
Hsinchu, Taiwan 300
{ejen.cs95g, perrys0620.cs96g}@nctu.edu.tw wcpeng@cs.nctu.edu.tw sylee@csie.nctu.edu.tw

ABSTRACT
Most studies on sequential pattern mining are mainly focused on time point-based event data. Few research efforts have elaborated on mining patterns from time interval-based event data. However, in many real applications, event usually persists for an interval of time. Since the relationships among event time intervals are intrinsically complex, mining time interval-based patterns in large database is really a challenging problem. In this paper, a novel approach, named as **incision strategy** and a new representation, called **coincidence representation** are proposed to simplify the processing of complex relations among event intervals. Then, an efficient algorithm, **CTMiner** (**Coincidence Temporal Miner**) is developed to discover frequent time-interval based patterns. The algorithm also employs two pruning techniques to reduce the search space effectively. Furthermore, experimental results show that CTMiner is not only efficient and scalable but also outperforms state-of-the-art algorithms.

Categories and Subject Descriptors
H.2.8 [**Data Management**]: Database Applications - Data Mining

General Terms
Algorithms

Keywords
data mining, interval-based mining, sequential pattern, temporal pattern

1. INTRODUCTION
Recently, sequential pattern mining is an important research theme with broad applications, such as DNA tandem repeats, user Web click streams, customer buying behaviors and study of medical process, to name a few. Most studies mainly focus on exploring an approach to discover frequent time-point based patterns in large sets of data [1, 5, 6, 10, 11, 14].

CIKM'10, October 26–30, 2010, Toronto, Ontario, Canada.

However, in many real applications, some events, which intrinsically tend to persist for periods of time instead of instantaneous occurrences, cannot be treated as "time points". For example, in the medical field, some relationships can be mined from clinical records of patients to study the correlations between the symptoms and the diseases, or the influences between the diseases and other diseases. Traditional sequential mining approaches is hampered by the fact that they can only handle instantaneous events efficiently, not event intervals.

Mining time interval-based patterns (referring as **temporal patterns**) from such data is undoubtedly more complex and arduous, and requires a different approach from mining time point-based data. So far, little attention has been paid to the issue of mining time interval-based sequential pattern. This is partly due to the complex relations among event intervals. Since the feature of time interval is quite different from time point, the pairwise relationships between any two time interval-based events are intrinsically complex. The complex relation is really a crucial problem when we endeavor to design a temporal pattern mining algorithm with efficiency and effectiveness, since it may lead to generate more numbers of candidate sequences and workload for counting the support of candidate sequence.

To the best of our knowledge, all related researches in this domain are based on Allen's temporal logics [2], in which there are 13 temporal relations between any two event intervals, as shown in Fig. 1. These relationships can describe any relative position of two intervals based on the arrangements of the start and the finish end time points. However, all the Allen's logics are binary relation and may suffer several problems when describing relationships among more than three events. An appropriate representation is very crucial for facing this circumstance. Various representations have been proposed but most of them have restriction on either ambiguity or scalability.

Some recent works have investigated the mining of interval-based events. Kam et al. [4] designed an algorithm that uses the hierarchical representation to discover frequent temporal patterns. However, the hierarchical representation is ambiguous and many spurious patterns are found. Hoppner [3] defined the supporting level of a pattern as the total time in which the pattern can be observed within a sliding window. However, the algorithm needs to scan the database repeatedly, which would significantly lower its efficiency. Papapetrou et al. [8] proposed the Hybrid-DFS algorithm to mine frequent arrangements of temporal intervals. Authors transform an event sequence into a vertical representation using id-lists. The id-list of an event is merged with the id-list of

other events to generate temporal patterns. This approach does not scale well when the temporal pattern length increases.

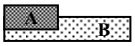
Temporal Relation	Inversed Relation	Pictorial Example
A before *B*	*B* after *A*	
A overlaps *B*	*B* overlapped-by *A*	
A contains *B*	*B* during *A*	
A starts *B*	*B* started-by *A*	
A finished-by *B*	*B* finishes *A*	
A meets *B*	*B* met-by *A*	
A equal *B*	*B* equal *A*	
A after *B*	*B* before *A*	
A overlapped-by *B*	*B* overlaps *A*	
A during *B*	*B* contains *A*	
A started-by *B*	*B* starts *A*	
A finishes *B*	*B* finished-by *A*	
A met-by *B*	*B* meets *A*	

Fig. 1: Allen's 13 relations between two intervals

Mochen [7] proposed a representation, TSKR which uses coincident concept to facilitate the processing of discovering temporal pattern. The pattern represented with TSKR is not easily understandable; it may reveal the relationship between pairwise event intervals in a pattern ambiguously. Winarko et al. [12] proposed an algorithm named ARMADA which is based on an efficient sequential pattern mining algorithm, MEMISP [5], to mine frequent temporal patterns. This approach only needs two database scans and does not require candidate generation or database projection, but memory usage is a bottleneck when database is very large.

Wu et al. [13] devised an algorithm called TPrefixSpan for mining temporal pattern from interval-based events. TPrefixSpan uses a nonambiguous representation, temporal sequence, to discover frequent temporal patterns. Although this algorithm only needs two scans of the database, it does not employ any pruning strategy to reduce the search space. Patel et al. [9] utilized additional counting information to achieve a lossless hierarchical representation, named Augmented Representation. An algorithm called IEMiner was designed to discover frequent temporal patterns from interval-based events. Although IEMiner uses some optimization strategies to reduce the search space and remove non-promising candidate sequences, it still has to scan database multiple times.

In this paper, a fundamentally different technique from previous work is proposed to discover temporal patterns. There are three contributions from our work. The first contribution is that we propose an **incision strategy**, to simplify the processing of complex relations when mining temporal patterns. The incision strategy segments all intervals to disjoint slices based on the global information in a pattern. Comparing with the complex relations between intervals, the relations among event slices are simple, i.e., only "*before*," "*after*" and "*equal*."

The second contribution is that we develop a new representation, **coincidence representation**, to express a pattern or sequence nonambiguously, based on the incision strategy. We group all event slices occurring simultaneously together to form a coincidence and concatenate all coincidences together to represent a pattern. As mentioned above, various existing representations may lead to different kinds of problem. An appropriate representation can facilitate processing and improve performance of algorithm.

The third contribution is that we design a new algorithm, **CTMiner,** which can effectively avoid the effort on candidate generation and testing for mining temporal patterns. We transform every interval sequence in database to coincidence format. Then, we borrow the idea from PrefixSpan [10] (Prefix-projected Sequential pattern mining), an efficient pattern growth-based algorithm in finding time pointed-based sequential patterns, to mine frequent temporal patterns. CTMiner recursively projects the temporal database into a set of smaller projected databases, and grows temporal patterns in each projected database by appending locally frequent slices. Furthermore, CTMiner employs the proposed optimization strategies to reduce the search space and avoids non-promising projection. The performance in both synthetic datasets and real datasets shows that CTMiner outperforms previous algorithms. Our experimental results also show that the proposed approach consumes a much smaller memory space.

The rest of the paper is organized as follows. Section 2 provides the detailed definitions. Section 3 and 4 introduce the incision strategy and the coincidence representation, respectively. Section 5 describes the CTMiner algorithm. Section 6 gives the experiments and performance study, and we conclude in Section 7.

2. PROBLEM DEFINITION

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

Definition 2 (Temporal database) Considering a database $D = \{r_1, r_2, \ldots, r_m\}$, each record r_i, where $1 \le i \le m$, consists of a sequence-id and an event interval (i.e. an event symbol, a starting time, and a finishing time, where starting time < finishing time). D is called a temporal database.

Actually, if all records in the database D with the same client-id are grouped together and ordered by nondecreasing start time, the database can be transformed into a collection of event sequences. As a result, the database D can be viewed as a collection of event sequences. For example, in Fig. 2, the temporal database consists of 17 event intervals, and 4 event sequences.

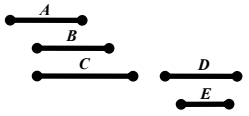
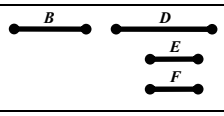
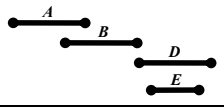
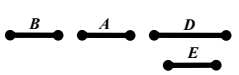
SID	event symbol	start time	finish time	event sequence
1	A	2	7	
1	B	5	10	
1	C	5	12	
1	D	16	22	
1	E	18	20	
2	B	1	5	
2	D	8	14	
2	E	10	13	
2	F	10	13	
3	A	6	12	
3	B	7	14	
3	D	14	20	
3	E	17	19	
4	B	8	16	
4	A	18	21	
4	D	24	28	
4	E	25	27	

Fig. 2: Database example

Definition 3 (Time set and time sequence) Given an event sequence $q = \langle (e_1, s_1, f_1), (e_2, s_2, f_2), \ldots, (e_n, s_n, f_n) \rangle$, The set $T = \{s_1, f_1, s_2, f_2, \ldots, s_i, f_i, \ldots, s_n, f_n\}$ is called a time set corresponding to sequence q where $1 \le i \le n$. If we order all the elements in T and eliminate redundant element, we can derive a sequence $Ts = \langle t_1, t_2, t_3, \ldots, t_k \rangle$ where $t_i \in T$, $t_i < t_{i+1}$. Ts is called a time sequence corresponding to sequence q.

Definition 4 (Event slice) A **start event slice** of event e_i, $1 \le i \le n$, is defined as an interval (e_i, s_i, t) where $t = \min\{ t' \mid t' \in Ts, s_i < t' < f_i \}$. A **finish event slice** of event e_i, $1 \le i \le n$, is defined as an interval (e_i, t, f_i) where $t = \max\{ t' \mid t' \in Ts, s_i < t' < f_i \}$. An **intermediate event slice** of event e_i, $1 \le i \le n$, is defined as an interval (e_i, t, t') where $t, t' \in Ts$, $t' \ne s_i$, $t' \ne f_i$, $s_i < t < t' < f_i$ and $t' = \min\{ t'' \mid t'' \in Ts, t < t'' < f_i \}$. An **intact event slice** of event e_i, $1 \le i \le n$, is defined as an interval (e_i, t, t') where $t = s_i$ and $t' = f_i$ and $\nexists\ t'' \in Ts$, such that $s_i < t'' < f_i$. The start, finish, intermediate, and intact event slice of event e_i are denoted as e_i^+, e_i^-, e_i^*, and e_i, respectively.

For example, in Fig. 2, sequence 2 has four event intervals, (B, 1, 5), (D, 8, 14), (E, 10, 13), and (F, 10, 13) and its corresponding time set = {1, 5, 8, 14, 10, 13, 10, 13} and time sequence = ⟨1, 5, 8, 10, 13, 14⟩. An event interval D has three event slices, start slice D^+ = (D, 8, 10), intermediate slice D^* = (D, 10, 13) and finish slice D^- = (D, 13, 14). The event interval B has only one intact slice B = (B, 1, 5). Obviously, an event interval can only have one start slice and one finish slice but can have many intermediate slices.

Definition 5 (Coincidence, coincidence sequence and Coincidence database)

Let set $L = \{ +, -, *, \varnothing \}$, a set of event sequences $Q = \{q_1, q_2, \ldots, q_i, \ldots\}$, $q_i = \langle (e_1, s_1, f_1), \ldots, (e_j, s_j, f_j), \ldots (e_n, s_n, f_n) \rangle$ where $(e_j, s_j, f_j) \in I$. A function $\mathbf{\Phi: N \times N \times Q \rightarrow (E \cup L)^+}$, $\mathbf{\Phi}$ **(** a, b, q_i **)** is defined as follow:

$$\forall (e_j, s_j, f_j) \quad \text{in } q_i, \quad \varepsilon_j = \begin{cases} e_j & \text{if } (s_j = a) \wedge (f_j = b) \\ e_j^+ & \text{if } (s_j = a) \wedge (f_j > b) \\ e_j^- & \text{if } (s_j < a) \wedge (f_j = b) \\ e_j^* & \text{if } (s_j < a) \wedge (f_j > b) \\ \phi & \text{otherwise,} \end{cases}$$

where $1 \le j \le n$, $\Phi(a, b, q_i) = \{ \varepsilon_1 \cup \varepsilon_2 \cup \cdots \cup \varepsilon_n \}$. Given an event sequence q, and two consecutive event times t_i and t_{i+1} in its corresponding time sequence $Ts = \langle t_1, t_2, t_3, \ldots, t_k \rangle$, a coincidence c_i is define as Φ (t_i, t_{i+1}, q), $1 \le i \le k-1$. A coincidence sequence Cs is denoted by $\langle c_1, c_2, \ldots, c_i, \ldots, c_{k-1} \rangle$. A coincidence database D_c is a set of tuples $\langle sid, Cs \rangle$ where sid is a sequence-id and Cs is a coincidence sequence.

3. INCISION STRATEGY

By our observation, the complex relation between each event interval is the major bottleneck for mining temporal patterns. We propose an incision strategy to address this critical issue.

There are six possible layouts between two consecutive event times, as shown in Fig. 3. Without loss of generality, we use "α" and "β" to represent two different event intervals. By traversing all end time points in a sequence, the incision strategy can segment all event intervals to event slices based on five kinds of outputs as Fig. 3(a) to 3(e). However, we cannot distinguish between two adjacent intervals and two disjoint intervals. A meet token "@" is used to address this drawback. As in Fig. 3(f), instead of segmenting any event interval, we only output the meet token "@" to assist the distinction of two adjacent event intervals.

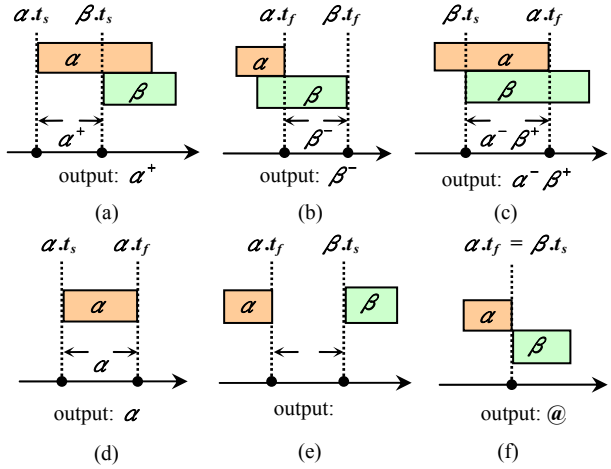


Fig. 3: Possible segmentations between two consecutive end time points

Now we give the detailed algorithm and discuss how to segment event intervals into disjoint event slices efficiently. The algorithm is shown in Fig. 4. Incision strategy first puts all the end time points of every event interval in sequence into a data structure *endtime_list* and sorts in non-decreasing order based on their times and types, i.e., start or finish (Lines 2-3, algorithm 1). If the times of two end time points are the same but the types are

different, the order is based on the type, i.e., finish type is smaller than start type. Then it merges the event symbol of end time points together if both time and type of end time points are identical (Line 4, algorithm). We can segment all event intervals to event slices by traversing all *endtime* records in *endtime_list* (Lines 5-12, algorithm 1). Notice that if the start slice e^+ and the finish slice e^- are in same *coincidence*, we can combine them to form an intact slice *e* since the interval *e* do not be incised (Line 14, algorithm 1).

Algorithm 1 incision_strategy (*Q*)

Input: An event sequence *Q*
Output: An incised sequence *Q*'
Variable: *endtime_list*, *last_endtime*, and *coincidence*

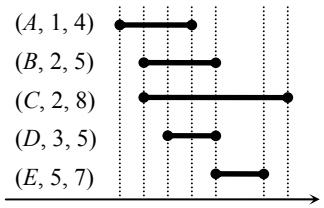
1: *endtime_list* ← ∅ , *last_endtime* ← ∅ , *coincidence* ← ∅ , *Q*' ← ∅ ;
2: add all the end time points of every event interval in *Q* into *endtime_list*;
3: sort every *endtime* in *endtime_list* by *endtime. time* in nondecreasing order;
4: merge all *endtime. symbols* together with identical *endtime. time* and *endtime. type*;
5: **for** each *endtime T* in *endtime_list* do
6: *coincidence* ← ∅ ;
7: **if** *last_endtime. time* = *T* . *time* **then**
8: *coincidence* ← *coincidence* ∪ "@"; // meet token
9: **else** // *last_endtime. time* ≠ *T* . *time*
10: **if** *last_endtime. type* = "*s*" **then**
11: *coincidence* ← *coincidence* ∪ every *symbol* in *last_endtime.symbol* add "+"; // start slice
12: **if** *T.type* = "*f*" **then**
13: *coincidence* ← *coincidence* ∪ every symbol in *T. symbol* add "–"; // finish slice
14: combine start slice and finish slice with same *symbol* in *coincidence*; // intact slice
15: *Q*' ← *Q*' ◇ ⟨ *coincidence* ⟩ ; // append *coincidence* to sequence
16: *last_endtime* ← *T*;
17: output *Q*';

Fig. 4: Algorithm of incision strategy

For example, considering an event sequence with 5 intervals shown in Fig. 5(a), we first put all 10 end time points into *endtime_list* and sort them in nondecreasing order based on their times and types. Then we merge the event symbol of end time points together if both time and type of end time points are the same. As in Fig. 5(b), since the $B.t_s$ is identical to the $C.t_s$, we can merge them together. But we can not combine $B.t_f$ and $D.t_f$ with $E.t_s$, since the type of end time points are not the same. We traverse all the sorted end time points in *endtime_list* one-by-one to generate the event slices.

Reducing memory usage and saving computation time are two important issues for algorithm design. Since we have utilized meet token to effectively distinguish two adjacent intervals, intermediate slices need not be incised. Given an example as Fig. 5(a), the event interval *C* can be segmented into five event slices, one start slice C^+, three intermediate slices C^*, and one finish slice C^-. The incision strategy can totally avoid the generation of intermediate slices. By trimming the intermediate slices, we can still express the relationship between any two intervals correctly, as shown in Fig. 5(c). This tactic reduces the memory usage and the computation cost effectively and efficiently, thereby improves the performance of our incision strategy.

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



(a)

endtime_list

symbol	*time*	*type*
A	1	*s*
BC	2	*s*
D	3	*s*
A	4	*f*
BD	5	*f*
E	5	*s*
E	7	*f*
C	8	*f*

(b)

$(A^+)(A^*B^+C^+)(A^-B^*C^*D^+)(B^-C^*\ D^-)$ @ $(C^*E)(C^-)$

⇩ intermediate slice trimming

$(A^+)(B^+C^+)(A^-D^+)(B^-D^-)$ @ $(E)(C^-)$

(c)

Fig. 5: An incision example

4. COINCIDENCE REPRESENTATION

All the Allen's relationships are binary relation and may suffer several problems when describing relationships among more than three events. An appropriate representation is very crucial for facing this circumstance. In this paper, a new format, coincidence representation is proposed to address the ambiguous and scalable problem. We group simultaneously occurring slices together to form the coincidences. Concatenation with all coincidences can describe an event sequence effectively and simplify the processing of complex pairwise relationships between all intervals efficiently. This is also the primary motivation of coincidence representation.

The coincidence representation of Allen's 13 relations between two event intervals is categorized as shown in Fig. 6. The new format is adopted to represent both event sequence and temporal pattern since it have several advantages, as follows:

- **Good scalability**: In the best case, all *k* intervals in a pattern are equal, thus the memory space for describing a *k*-interval pattern is *k*. In the worst case, all *k* intervals overlap one-by-one, thus we require 2*k* memory space to express a *k*-intervals pattern.
- **Nonambiguity:** A representation is ambiguous if 1) the same relationships between intervals may be mapped to different temporal patterns and 2) the patterns cannot reveal the temporal relations between all pairs of intervals. Accordingly,

the following observations indicate that the ambiguity no longer existed in coincidence representation. First, according to Definitions 4 and 5, we can build a unique coincidence sequence by transforming every event sequence into coincidence representation. In other words, the temporal relations among intervals can be mapped one-to-one to a coincidence sequence. Second, in a coincidence sequence, the order relation of the start and finish slices of α and β can be categorized as shown in Fig. 6. We can infer the original temporal relationships between intervals α and β nonambiguously.

- **Simple is good:** Obviously, the complex relations between intervals are the major bottleneck of temporal pattern mining since the mining may have to generate or examine more explosive number of intermediate subsequences. By incision strategy, we can transform event intervals into non-overlapped fragments, event slices. The relations between event slices are simple, just "*before*," "*after*" and "*equal*." The simpler the relations, the less number of intermediate candidate sequences are required generating and processing.
- **Compact space usage:** Since the utilization of meet token, we can omit the intermediate slices within the coincidence sequences or patterns. This tactic can reduce the computation time and consuming memory space effectively and efficiently.

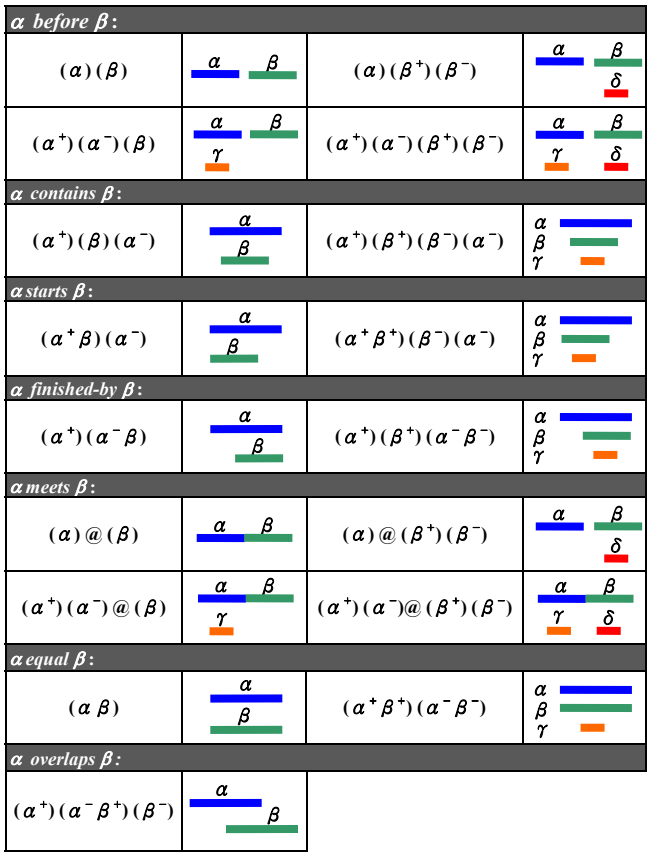


Fig. 6: The coincidence representation of Allen's 13 relations between two intervals

5. CTMiner ALGORITHM

Definition 6 (Projected database) Let Cs be a coincidence sequence in a database D. The Cs - projected database, denoted as $D_{|Cs}$, is the collection of suffixes of sequences in D with regards to prefix Cs.

Definition 7 (Subsequence, support count and temporal pattern) Considering two coincidence sequence $s_1 = \langle a_1 a_2 \ldots a_n \rangle$ and $s_2 = \langle b_1 b_2 \ldots b_m \rangle$, s_1 is called a subsequence of s_2, denoted as $s_1 \sqsubseteq s_2$, if there exist integers $1 \le i_1 \le i_2 \le \ldots \le i_n \le m$ such that $a_1 \subseteq b_{i1}$, $a_1 \subseteq b_{i1}$, …, $a_n \subseteq b_{in}$, and s_2 is also called a supersequence of s_1. Given a coincidence database D_c, a tuple $\langle sid, Cs \rangle$ is said to contain a coincidence sequence s_1, if s_1 is a subsequence of Cs. The support of a coincidence sequence s_1 in a coincidence database D_c is the number of tuples in the database containing s_1, i.e., $\boldsymbol{support}(s_1) = |\{\langle \boldsymbol{sid}, \boldsymbol{Cs} \rangle | (\langle \boldsymbol{sid}, \boldsymbol{Cs} \rangle \in \boldsymbol{D_c}) \wedge (s_1 \sqsubseteq \boldsymbol{Cs})\}|$. Given a positive integer *min_sup* as the support threshold, a coincidence sequence s_1 is called a temporal pattern if $support(s_1) \ge min\_sup$.

Algorithm 2: CTMiner (*D*, *min_sup*)

Input: a temporal database *D*, and the minimum support threshold *min_sup*

Output: all frequent temporal patterns *L*

```
1:  L ← ∅;
2:  scan temporal database D to find all frequent 1-intervals and
    remove infrequent intervals;
3:  L1 ← all frequent 1-intervals;
4:  L ← L1;
5:  for each interval a ∈ L1 do
6:      for each event sequence q has interval "a" in D do
7:          q' = incision_strayegy ( q )
8:          use q' to construct projected database D|a with regard to intact
            slice a;
9:      CPrefixSpan (D|a , a, min_sup, L );  // mining intact slice a
10:     if support (a+) ≥ min_sup then
11:         construct projected database D|a+ ;
12:         CPrefixSpan (D|a+ , a+, min_sup, L );  // mining start slice a+
13: output all frequent temporal patterns L;
```

Fig 7: CTMiner algorithm

Fig. 7 illustrates the main framework which includes the necessary processing steps for discover frequent temporal patterns. Given a temporal database, the event intervals associated with the same sequence ID can be grouped into an event sequence. CTMiner first scans the temporal database to discover all frequent intervals and remove infrequent intervals (Line 2, algorithm 2). For each frequent interval a, we call **incision_strategy** to transform all event sequences who have interval a to corresponding coincidence sequences. Then we collect all coincidence sequences to build projected database $D_{|a}$ for intact slice a (Lines 6-8, algorithm 2).

As mentioned above, an event interval a may be incised to three possible event slices, intact slice a, start slice a^+ and finish slice a^-. Obviously, the support count of intact slice a, $support(a)$ is identical to the number of occurrence of interval a in database. Since interval a is frequent, intact slice a is definitely frequent. Thus, we call **CPrefixSpan** recursively to discover all frequent

coincidence patterns prefixed with the intact slice (Lines 9, algorithm 2). The support count of the start slice a^+ can be accumulated concurrently with constructing projected coincidence database $D_{|a}$. If the start slice a^+ is frequent, we also construct the projected coincidence database $D|_{a+}$ and call **CPrefixSpan** recursively to discover all frequent coincidence patterns prefixed with the start slice (Lines 10-12, algorithm 2). Finally, we output all frequent temporal patterns (Line 13, algorithm 2).

In Fig. 8, CPrefixSpan, is similar to PrefixSpan, however it performs two search space pruning strategy using the concepts of slice-and-coincidence. CPrefixSpan scans projected coincidence database once to collect all local frequent 1-patterns (Lines 1-3, algorithm 3). For each frequent 1-pattern, we can append it to original prefix to generate new pattern with the length increased by 1. This way, the prefixes are extended (Lines 4-9, algorithm 3). Finally, constructing the projected database with the frequently extended prefixes and recursively running until the prefixes cannot be extended will discover all frequent temporal patterns (Lines 10-14, algorithm 3). With the property of event slice and coincidence, we propose two pruning strategies, **pre-pruning** and **post-pruning** to reduce the searching space efficiently and effectively. First, the event start slices and finish slices definitely occur in pairs in a coincidence sequence. We only require projecting the frequent finish slices which have the corresponding start slices in their prefixes (Lines 6-7, algorithm 3). It is called pre-pruning strategy which can prune off non-qualified patterns before constructing projected database.

Algorithm 3: CPrefixSpan ($D_{|a}$, a, *min_sup*, L)

Input: a projected database $D_{|a}$, a coincidence sequence a, the minimum support *min_sup*, and a set of frequent temporal patterns L

Output: a set of frequent temporal patterns L

1: scan $D_{|a}$ once, remove infrequent slices and find every frequent slice b such that:
2: (i) b can be assembled to the last slice of a to form a temporal pattern; or
3: (ii) $\langle b \rangle$ can be appended to a to form a temporal pattern;
4: **for each** frequent slice b **do**
5: **if** b is a "finish slice" **then**
6: **if** exist corresponding start slice b^+ in a **then** // **pre-pruning**
7: append b to a to form a temporal pattern a';
8: **if** b is a "start slice" or "intact slice" **then**
9: append b to a to form a temporal pattern a';
10: **for each** a' **do**
11: construct a'– projected database $D_{|a'}$ with insignificant postfix elimination; // **post-pruning strategy**
12: **if** $|D_{|a'}| >$ *min_sup* **then**
13: $L \leftarrow L \cup a'$;
14: call **CPrefixSpan** ($D_{|a'}$, a', *min_sup*, L);

Fig 8: CPrefixSpan algorithm

Second, when we construct a projected database, some postfixes do not require considering. With respect to a prefix $\langle p \rangle$, a projected postfix is called significant, if all finish slices in postfix have corresponding start slices in $\langle p \rangle$. When constructing the projected database $D_{|\langle p \rangle}$, only the significant postfixes are collected and all insignificant postfixes are eliminated since they can be ignored in the discovery of frequent temporal patterns. The second pruning method is called post-pruning strategy which eliminate insignificant sequence when constructing projected database (Line 11, algorithm 3)

Different from PrefixSpan algorithm, CPrefixSpan can not guarantee that the new temporal patterns formed from appending previously discovered patterns with locally frequent slices are always frequent. We require an additional computation to insure that the support count of the coincidence sequences in a projected database is no less than min_sup (Lines 12-13, algorithm 3). Since $|D_{|a}|$ (number of sequences in $D_{|a}$) can be produced by using a simple counter when we project the database, the computation cost is nearly negligible. The experimental studies indicate that pre-pruning and post-pruning strategies can improve the performance in both computation time and memory usage efficiently.

Notice that when scanning projected coincidence database to calculate the support count of an intact slice b, both b and start slice b^+ occurring in coincidence sequences need to be accumulated. Since the only difference between intact slice and start slice is whether the event interval have incised or not, both of them in the coincidence sequence imply the existence of an event. But when counting the support of start slice b^+ or finish slice b^-, only the occurrence of b^+ or b^- in a database have to be accumulated.

We take the database in Fig. 2 with *min_sup* = 2 as an example. There are 17 event records which can be regarded as 4 event sequences in the temporal database. After scanning database, we find all the frequent intervals, $\langle A \rangle$, $\langle B \rangle$, $\langle D \rangle$ and $\langle E \rangle$. These intervals are also called frequent 1-temporal patterns. The event sequences with corresponding coincidence representation are shown as in first column in Fig. 9. We take the interval A and E as examples to further discuss in details. With considering interval A, we have to process the patterns prefixed with intact slice $\langle A \rangle$ and start slice $\langle A^+ \rangle$. Notice that when counting the support and constructing projected database with regard to intact slice $\langle A \rangle$, we also require considering the occurrence of start slice $\langle A^+ \rangle$ in sequences.

The projected coincidence database with respect to $\langle A \rangle$ has 3 sequences: $\langle (\_B^+)(B^-)(D^+)(E)(D^-) \rangle$, $\langle (\_B^+)(B^-)(@D^+)(E)(D^-) \rangle$ and $\langle (D^+)(E)(D^-) \rangle$. When constructing the $\langle A \rangle$-projected database, we can also count the support count of start slice $\langle A^+ \rangle$ concurrently. Since the support count of $\langle A^+ \rangle$ is $2 \geq$ *min_sup*, we have to construct corresponding projected coincidence database. The projected coincidence database with respect to $\langle A^+ \rangle$ has 2 sequences: $\langle (A^- B^+)(B^-)(D^+)(E)(D^-) \rangle$ and $\langle (A^- B^+)(B^-)(@D^+)(E)(D^-) \rangle$. Continuing the recursive process with the $D_{|\langle A \rangle}$ and $D_{|\langle A^+ \rangle}$, we can discover all frequent coincidence patterns prefixed with $\langle A \rangle$ and $\langle A^+ \rangle$ respectively. In addition, when projecting intact slice $\langle E \rangle$, the generated postfixes will be eliminated by post-pruning strategy directly since $\langle D^- \rangle$ is insignificant. We do not have to consider the $\langle E \rangle$-projected database. The last column in Fig. 9 lists all generated frequent temporal patterns.

event sequences with corresponding coincidence representation	slice prefix	projected coincidence database	frequent temporal patterns
S1: $\langle(A^+)(A^-B^+C^+)(B^-)(C^-)(D^+)(E)(D^-)\rangle$ S2: $\langle(B)(D^+)(EF)(D^-)\rangle$ S3: $\langle(A^+)(A^-B^+)(B^-)(@D^+)(E)(D^-)\rangle$ S4: $\langle(B)(A)(D^+)(E)(D^-)\rangle$ ↓ **infrequent slice elimination** S1: $\langle(A^+)(A^-B^+)(B^-)(D^+)(E)(D^-)\rangle$ S2: $\langle(B)(D^+)(E)(D^-)\rangle$ S3: $\langle(A^+)(A^-B^+)(B^-)(@D^+)(E)(D^-)\rangle$ S4: $\langle(B)(A)(D^+)(E)(D^-)\rangle$	$\langle A \rangle$	S1: $\langle(\_B^+)(B^-)(D^+)(E)(D^-)\rangle$ S3: $\langle(\_B^+)(B^-)(@D^+)(E)(D^-)\rangle$ S4: $\langle(D^+)(E)(D^-)\rangle$	$\langle A \rangle$ $\langle(A)(D)\rangle$ $\langle(A)(E)\rangle$ $\langle(A)(D^+)(E)(D^-)\rangle$ $\langle(A^+)(A^-B^+)(B^-)\rangle$ $\langle(A^+)(A^-B^+)(B^-)(E)\rangle$
	$\langle A^+\rangle$	S1: $\langle(A^-B^+)(B^-)(D^+)(E)(D^-)\rangle$ S3: $\langle(A^-B^+)(B^-)(@D^+)(E)(D^-)\rangle$	
	$\langle B \rangle$	S1: $\langle(D^+)(E)(D^-)\rangle$ S2: $\langle(D^+)(E)(D^-)\rangle$ S3: $\langle(@D^+)(E)(D^-)\rangle$ S4: $\langle(A)(D^+)(E)(D^-)\rangle$	$\langle B \rangle$ $\langle(B)(D)\rangle$ $\langle(B)(E)\rangle$ $\langle(B)(D^+)(E)(D^-)\rangle$
	$\langle B^+\rangle$	S1: $\langle(B^-)(D^+)(E)(D^-)\rangle$ S3: $\langle(B^-)(@D^+)(E)(D^-)\rangle$	
	$\langle D \rangle$	∅	$\langle D \rangle$ $\langle(D^+)(E)(D^-)\rangle$
	$\langle D^+\rangle$	S1: $\langle(E)(D^-)\rangle$ S2: $\langle(E)(D^-)\rangle$ S3: $\langle(E)(D^-)\rangle$ S4: $\langle(E)(D^-)\rangle$	
	$\langle E \rangle$	S1: ~~$\langle(D^-)\rangle$~~ ← **insignificant** S2: ~~$\langle(D^-)\rangle$~~ ← **insignificant** S3: ~~$\langle(D^-)\rangle$~~ ← **insignificant** S4: ~~$\langle(D^-)\rangle$~~ ← **insignificant**	$\langle E \rangle$
	$\langle E^+\rangle$	∅	

Fig. 9: Example of projected databases and frequent temporal patterns

6. EXPERIMENTAL RESULTS

To evaluate the performance of CTMiner, three temporal pattern mining algorithms, TPrefixSpan [13], IEMiner [9], and H-DFS [8] were also implemented for comparison. All algorithms were implemented in C++ language and tested on a Pentium D 3.0 GHz with 2 GB of main memory running Windows XP system. The performance study has been conducted on both synthetic and real world datasets. The synthetic data sets in the experiments are generated using synthetic generation program proposed by Agrawal et al. [1]. Since the original data generation program was designed to generate time point-based data, the generator for the temporal pattern mining algorithms requires modifications accordingly. We adopt the modification proposed by Wu et al. [13]. All the duration times of event intervals are classified into three categories: long, medium and short. The long, medium and short interval events are with an average length of 12, 8 and 4 respectively. For each event interval, we first randomly decide its category and then determine its length by drawing a value from a normal distribution. The temporal data generator takes eight parameters same as previous research, as shown in Fig. 10.

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

Fig. 10: Parameters of synthetic data generator

6.1 Runtime performance on synthetic data sets

In all the following experiments, some parameters are fixed, i.e., $|T| = 2.5$, $|S| = 4$, $N_S = 5{,}000$ and $N_I = 10{,}000$. The other parameters are configured for comparing the temporal pattern

mining algorithms. The first experiment of the four algorithms is on the data set *D*10k–*C*10–*I*1.25–*N*1k. Fig. 11 and Fig. 12 show the running time of the four algorithms and the distribution of generated frequent patterns, respectively. The minimum support thresholds vary from 1 % to 4 %. Obviously, when the minimum support value decreases, the processing time required for all algorithms increases. However, the runtime for IEMiner, H-DFS and TPrefixSpan increase drastically compared to CTMiner.

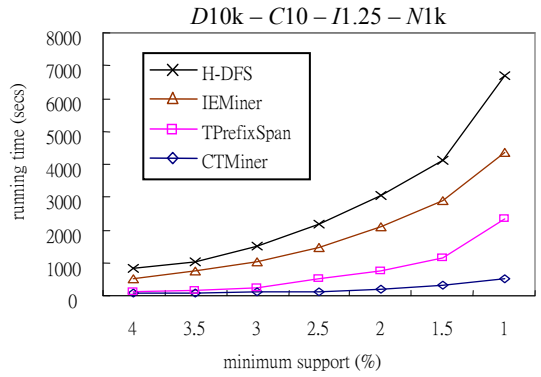


Fig. 11: Performance of the four algorithms on data set *D*10k – *C*10 – *I*1.25 – *N*1k

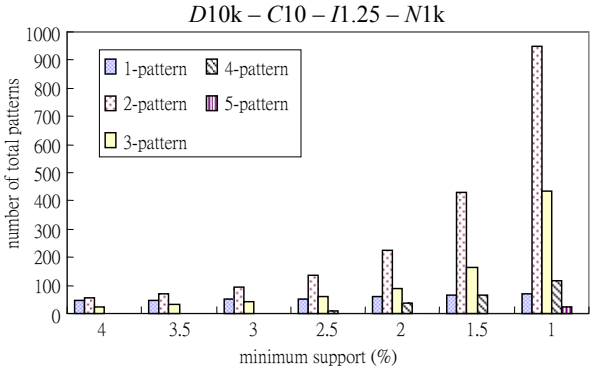


Fig. 12: The distribution of generated patterns on data set *D*10k – *C*10 – *I*1.25 – *N*1k

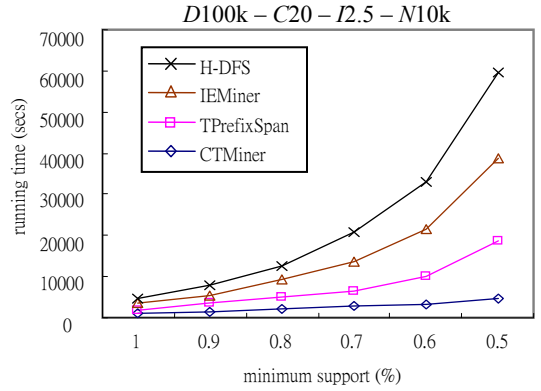


Fig. 13: Performance of the four algorithms on data set *D*100k – *C*20 – *I*2.5 – *N*10k

The second experiment is performed on data set *D*100k–*C*20–*I*2.5–*N*10k, which is much larger than first experiment. Fig. 13 and Fig. 14 show the running time and the distribution of generated frequent patterns at different support thresholds. However, we vary the minimum support thresholds from 0.5 % to 1 % to generate larger number of frequent patterns from large data set. The data set contains a large number of frequent temporal patterns when minimum support is reduced to 0.5 %. We can see that CTMiner is significantly faster than the other algorithms. The lower the minimal support, the faster CTMiner compared with other algorithms.

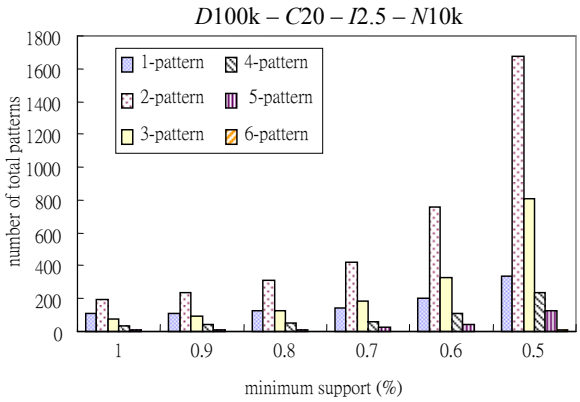


Fig. 14: The distribution of generated patterns on data set *D*100k – *C*20 – *I*1.25 – *N*10k

We also study the scalability of the CTMiner algorithm. Fig. 15 shows the results of scalability tests of the CTMiner algorithm, with the database size growing from 100K to 500K sequences, and with different minimum support threshold settings. Here, we use the data set *C*10–*T*2.5–*I*1.25–*N*10k with varying different database size. As can be seen, CTMiner is linearly scalable with different minimum support threshold and with different database size.

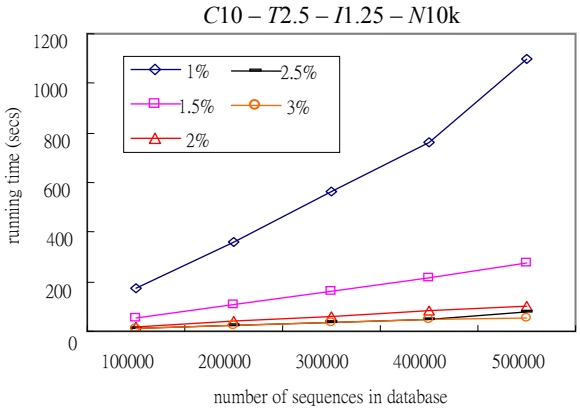


Fig. 15: Performance of the CTMiner algorithm with different database size

Then, we compare the memory usage among the four algorithms using synthetic data set *D*10k – *C*10 – *I*1.25 – *N*1k. Fig. 16 shows the results, from which we can observe that CTMiner is not only more efficient, but also more stable in memory usage than other three algorithms. Based on our analysis, CTMiner only needs memory space to hold the sequence data sets plus a set of header tables and pseudoprojection tables to construct projected databases. Although TPrefixSpan is also designed based on

PrefixSpan, it still consumes memory space to hold the generated candidate sequences since the complex relation between intervals. Both IEMiner and H-DFS need memory space to hold candidate sequences in each level. When the minimal support threshold drops, the set of candidate sequences grows up quickly, which results in memory consumption upsurging.

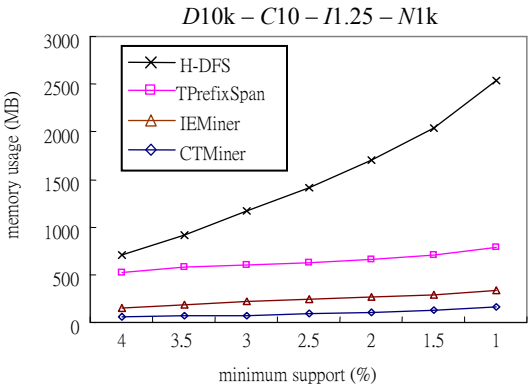


Fig. 16: The memory usage of four algorithms on data set $D10k - C10 - I1.25 - N1k$

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

6.2 Real world dataset analysis

In addition to using synthetic data sets, we have also performed an experiment on real world data set to compare the performance. The database used in experiment consists a collection of 1,098,142 library records (lending and returning) for three years from the National Chiao Tung University Library. The experimented database includes 206,844 books and 28,339 readers. An event interval is constructed by a book ID and corresponding lending and returning time. The size of database is the number of sequences in database (same as number of readers, 28,339). The maximal and the average length of sequence are 262 and 38 respectively. Fig. 17 indicates the running time of four temporal pattern mining algorithms with varying minimum support thresholds from 0.1 % to 0.05 %. The distribution of generated patterns is shown in Fig. 18. As the minimum support drops down to 0.05 %, there are 14,549 frequent patterns and the running time of CTMiner takes 4,771 seconds, which is about 2 times faster than TPrefixSpan, about 4 times faster than IEMiner and H-DFS has never terminated.

7. CONCLUSION AND FUTURE WORK

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

Based on coincidence representation, we develop an efficient algorithm, **CTMiner** to discover frequent temporal patterns without candidate generation. The algorithm further employs two pruning techniques to reduce the search space effectively. The experimental studies indicate that CTMiner is efficient and scalable. Both running time and memory usage of CTMiner outperform state-of-the-art algorithms.

To the best of our knowledge, all previous extensions of mining sequential pattern only focus on time point-based data. No attention has been paid to the related extension studies of mining temporal patterns from time interval-based data. By our observation, the major reason is the complex relation among intervals. In this paper, we utilize proposed coincidence representation to overcome this problem and facilitate the processing. Hence, based on coincidence representation, there are many interesting extensions that may be studied further, such as mining closed and maximal temporal patterns, incremental temporal patterns mining, and the research of method toward data stream.

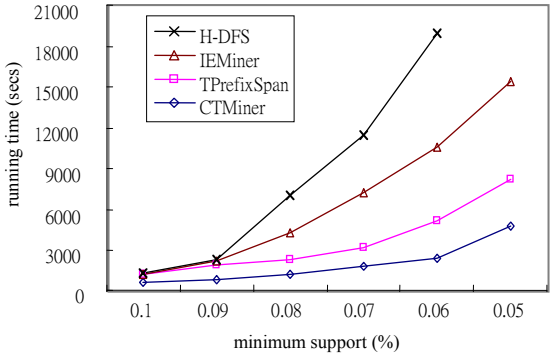


Fig. 17: Performance of the four algorithms on library data set from NCTU

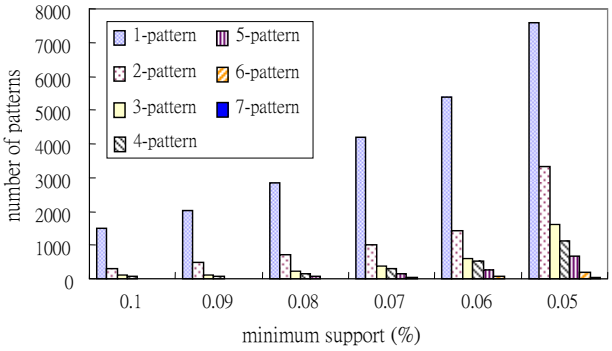


Fig. 18: The distribution of generated patterns on library data set from NCTU

8. ACKNOWLEGEMENT

This work was supported in part by the National Science Council of Taiwan. Suh-Yin Lee was supported by Project No. NSC99-2221-E-009-128-MY2 and Wen-Chih Peng was supported by Project No. NSC97-2221-E-009-053-MY3.

9. REFERENCES

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

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

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

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

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

[6] F. Masseglia, F. Cathala and P. Poncelet. The PSP Approach for Mining Sequential Patterns. *European Conference on Principles of Data Mining and Knowledge Discovery (PKDD'01)*, vol. 1510, pp. 176-184, 1998.

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

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

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

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

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

[12] E. Winarko and J.F Roddick. ARMADA-An algorithm for discovering richer relative temporal association rules from interval-based data. *Data & Knowledge Engineering*, vol. 3, issue 1, pp. 76-90, 2007.

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

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