Discovery of Temporal Patterns from Process Instances

San-Yih Hwang*, Chih-Ping Wei, and Wan-Shiou Yang

Department of Information Management
National Sun Yat-Sen University
Kaohsiung, Taiwan, R.O.C.

Abstract

Existing work in process mining focuses on the discovery of the underlying process model from their instances. In this paper, we do not assume the existence of a single process model to which all process instances comply, and the goal is to discover a set of frequently occurring temporal patterns. Discovery of temporal patterns can be applied to various application domains to support crucial business decision-making. In this study, we formally defined the temporal pattern discovery problem, and developed and evaluated three different temporal pattern discovery algorithms, namely TP-Graph, TP-Itemset and TP-Sequence. Their relative performances are reported.

Keywords: Process Mining, Knowledge Discovery, Data Mining, Temporal Patterns, Association Rules, Sequential Patterns

January 2003

*: Corresponding author.
Tel: +886-7-5252-000 ext 4723; fax: +886-7-525-4799; e-mail: syhwang@mis.nsysu.edu.tw

1. Introduction

As organizations become more process conscious, management of processes and process data with temporal context is gaining increased attention. For example, business process reengineering, referring to the fundamental analysis and radical redesign of business processes to achieve dramatic improvements in such critical measures of performance as cost, quality, service and speed (Hammer and Champy, 1993), represents an important endeavor toward business revolution in the last decade. On the other hand, workflow management aims at improving process efficiency and customer satisfaction by automating and actively monitoring process execution. To achieve these goals, the description of constituent activities, the control/data flow, the potential participants of various activities, the organizational model and reference data pertaining to a business process have to be specified precisely (WMC, 1994).

Business process reengineering and workflow management are directed toward systematic analysis and management of organizational processes. Not surprisingly, even where processes have been managed and monitored by organizations, exceptional executions often occur. The importance of handling exceptions in the context of workflow management has been recognized in the recent development of several research prototypes (e.g., EXOTICA (Alonso et al., 1994), METEOR (Sheth and Kochut, 1997), ADOME (Chiu et al., 1999; Chiu et al., 2001), ADEPT (Reichert and Dadam, 1998), WAMO (Eder and Liebhart, 1998), and WIDE (Casati et al., 1999)). On the other hand, since the difficulty of defining a process model that represents all properties of an underlying business process has been acknowledged (Davenport 1993), ad hoc processes frequently have been observed (Krishnakumar and Sheth, 1995; Tang and Hwang, 1996). Process executions or instances, ad hoc or exceptional, are valuable to an organization, since they not only record execution tracks but, more importantly, also embed execution practices and heuristics. With the widespread diffusion of information systems in organizations, large volumes of process data are continuously generated and collected, creating an urgent need for new analysis techniques that can intelligently and automatically extract implicit and potentially useful knowledge to support business decision making.

Most of the previous work in process mining (or called workflow mining or process discovery) assumes the existence of an underlying process model that generates process instances. Thus, given a set of process instances, the goal of process mining is to discover a process model that best describes them. In contrast to the previous work, this work does not assume the existence of such an underlying process model. Under such an assumption, it may not be practical to discover a single process model from a set of process instances. Given a set of process instances, we aim to discover knowledge of frequently occurring activities and respective temporal relationships that exist within these process instances. Since the discovered knowledge involves activities with temporal relationships, they are referred to as temporal patterns in this study. The discovery of temporal patterns can be applied to various application domains for supporting crucial business decisions. For example, to provide better patient care and high quality health services, implementation of clinical pathways are desirable for health care providers (Quigley et al., 1998). Evidently, design of a clinical pathway for a particular diagnosis, procedure or symptom is a time-consuming and knowledge-intensive process. Thus, use of clinical care logs for supporting design of clinical pathways represents a promising approach (Lin et al., 2001). However, since clinical cares for patients with the same diagnosis, procedure or symptom may vary with physicians' practice preferences and styles, attempting to discover from clinical care logs a universal clinical pathway for a particular diagnosis, procedure or symptom may not be practical. Instead, it would be essential to discover from clinical care logs the frequent (partial) clinical paths that become valuable inputs when design a clinical pathway or a set of clinical pathways for a particular diagnosis, procedure or symptom. In the context of project management, it is desirable to discover how plan patterns that frequently result in desired plan executions differ from those that are frequently associated to undesired executions. In the following, a real world application that applied our proposed algorithms is depicted for illustration purpose.

Example

Health care has become a major focus of concern and even a political, social, and economic issue in modern society. People rely on government-sponsored and -managed health insurance systems (such as in Australia, France, and Taiwan), private health insurance systems, or both to share the expensive health care costs. With such an intensive need for health insurances, health care fraud and abuse become a serious problem. Of the various parties involved in the health care fraud and abuse, medical institutes seem to be responsible for most of the damages. Conceivably, an insurance claim is likely to be fraudulent if its constituent activities order suspiciously. In this case, it is sensible to discover temporal patterns from normal and fraudulent practices (as provided by experts) and subsequently identify those patterns that are capable of distinguishing fraudulent practices from normal ones.

We gathered 2543 insurance claim instances about Pelvic Inflammatory Disease from regional hospitals in Taiwan during July 2001 and June 2002. After removing instances that contain missing or noisy data, we were left with 2011 cases, among which 906 instances were identified by two physicians as fraudulent. We applied the temporal pattern mining algorithms proposed in this study to identify a set of frequent temporal patterns. We then applied a classification analysis technique (specifically, C4.5) for fraud/abuse detection by using the discovered temporal patterns as predictive features. The resultant detection model was able to achieve a detection accuracy of around 65%. □

Numerous data mining techniques have been proposed to extract implicit and potential useful knowledge from large databases. Based on the types of knowledge to be discovered, data mining techniques can be broadly classified into several categories, including classification (Breiman et al., 1984; Quinlan, 1986; Rumelhart et al., 1986; Quinlan, 1993), clustering (Anderberg, 1973; Kohonen, 1989; Kaufman and Rousseeuw, 1990; Ng and Han, 1994; Kohonen, 1995), association rule (Agrawal et al., 1993; Agrawal and Srikant, 1994; Srikant and Agrawal, 1997; Aggarwal and Yu, 2001), sequential pattern (Agrawal and Srikant, 1995; Lesh et al., 2000; Srikant and Agrawal, 1996), data visualization (Keim and Kriegel, 1996), etc. A review of existing data mining techniques suggests that little attention has been paid to the discovery of temporal patterns. Thus, the current research was motivated by recognition of the importance of mining temporal patterns from process data. Specifically, we formalized the temporal pattern discovery problem and designed three algorithms that are based on the notions of temporal graphs, temporal relationship sets, and quasi-sequences, respectively. Subsequently, the performance and scalability of these algorithms were empirically evaluated over a range of data characteristics using synthetic data sets.

The remainder of the paper is organized as follows. Section 2 reviews prior research work related to this study. Section 3 formally defines concepts related to temporal pattern discovery. The three temporal pattern discovery algorithms proposed are detailed in Section 4. Section 5 describes a model for generating synthetic data sets for performance evaluation and reports evaluation results of applying the proposed algorithms over a range of data characteristics. The paper is concluded in Section 6 with a summary and some future research directions.

2. Related Work

The work reported in (Agrawal et al., 1998; Datta, 1998; Aalst et al., 2002; Hwang and Yang, 2002) deals with the problem of discovering a process model from a set of process instances and assumes the existence of a process model (i.e., control dependencies between activities) underlying the given set of process instances. In this vein, such discovery, using a directed graph (Agrawal et al., 1998; Hwang and Yang, 2002) , a finite state machine (Datta, 1998), or a Petri-net (Aalst et al., 2002) for representing process instances, aims at discovering a process model that best describes the set of process instances. This discovery problem has its origin in identifying software development processes and has found its way into business process construction (Datta, 1998). Our study significantly differs from the process model discovery in that we do not assume the existence of an underlying process model. That is, our study is designed to identify frequently observed temporal dependencies within process instances rather than control dependencies that are presumably genuine in the process instances.

Our work is closest to sequential pattern discovery that discovers frequent sequential occurrence of activities (e.g., items purchased or events) across transactions of the same entity (e.g., customer or a user-specified time window). The sequential pattern discovery reported in (Agrawal and Srikant, 1995; Srikant and Agrawal, 1996) finds the maximal sequences among all sequences that have a certain user-specified minimum support. Each such maximal sequence is referred to as a sequential pattern. Mannila et al. (1995) reported a technique for finding frequent episodes from a given event sequence, where an episode is defined as a partial order on a set of events. The given event sequence is first partitioned into a set of windows, each of which is an event subsequence with time width equal to a user-specified interval. An episode *e* is said to occur in a window *w* if all (partial) orders attained in *e* are present in *w*. The goal of the episode discovery is to find the set of episodes that occur in a sufficient number of windows. The concept of episodes was further generalized by taking into account attributes associated with events (Mannila and Toivonen, 1996). The sequential pattern discovery problem assumes that a transaction contains a set of activities occurring at the same time and that transactions of the same entity are sequentially ordered, while the episode discovery problem takes as input a sequence of events occurring at different times. In contrast, we assume that an activity appears over a temporally extended interval, two activities may temporally overlap or occur in sequence. Both sequential pattern discovery and episode discovery address only one type of temporal relationships (i.e., the sequential order), while our work seeks to identify patterns that are composed by both sequential and overlapping temporal relationships.

In addition, a structure discovery system called Subdue was proposed in (Cook and Holder, 2000; Gonzalez et al., 2000). Subdue is used to identify interesting and repetitive substructures within structural data that can be aggregately represented as a graph. The substructure discovery technique enumerates all possible subgraphs of the given graph and for each subgraph, uses a graph match algorithm to identify all its occurrences. Priority is given to the substructure that has smaller Minimum Description Length (MDL) (Rissanen, 1983). A MDL of a subgraph is proportional to the summation of the length of the subgraph and the length of the given graph compressed by the subgraph. In other words, an ideal subgraph is the one that is not very large but is powerful in compressing the graph. Iteration of the substructure discovery and replacement process constructs a hierarchical description of the structural data in terms of the discovered substructures. Such a hierarchy provides varying levels of abstraction for subsequent data analysis. By its nature, this technique identifies repetitive subgraphs from a large graph, rather than a large set of graphs as assumed by our work. Besides, it discovers only substructures that are regionally connected subgraphs and disregards transitive relationships among objects, limiting its applicability to our temporal pattern discovery problem, where transitivity in temporally sequential relationships prevails.

Finally, Bettini et al. (1998) deals with the discovery of frequent-event patterns in a time sequence that consists of a set of time-stamped events. The discovery process starts with a user-specified event structure that consists of a set of variables representing events and temporal constraints between variables. Its goal is to identify instantiations of variables in the event structure that appear frequently in the time sequence. The event pattern discovery differs from our work in several ways. First, it assumes an event appears at a time point rather than over a time interval. Second, it searches for instantiations of a user-specified event structure within a time sequence rather than discovering all possible frequent temporal relationships among events from a set of process instances.

3. Formalization of Temporal Pattern Discovery Problem

A process consists of a set of activities, each of which is an execution unit that leads to the transition of state in the process. The execution of an activity spans a temporally extended period. Each activity may also be associated with such information as execution entity(s) involved, execution location and execution outcome. However, since the main intent of this research is to discover frequent activities and their associated temporal dependencies, we exploit only the starting time and ending time of an activity execution. Our view on a process instance can be formally described as below.

Definition 1. A process instance P is a set of triplets $(v_i, st(v_i), et(v_i))$, where v_i uniquely identifies an activity, and $st(v_i)$ and $et(v_i)$ are timestamps representing the starting time and ending time of the execution of v_i in P, respectively.

Given a process instance, the temporal relationship between any activity pair can be classified into two types: *followed* and *overlapped*.

Definition 2. In a process instance P, an activity v_i is *followed* by another activity v_j if $st(v_j) \geq et(v_i)$.

Definition 3. In a process instance P, two activities, v_i and v_j, are *overlapped* if $st(v_j) \leq st(v_i) < et(v_j)$ or $st(v_i) \leq st(v_j) < et(v_i)$.

Definition 4. An activity v_i is *directly followed* by another activity v_j in a process instance P if v_i is followed by v_j and there does not exist a distinct activity v_k in P such that v_i is followed by v_k

and v_k is followed by v_j.

To represent temporal relationships between activities in a process instance concisely, a *temporal graph* is defined as follows.

Definition 5. The pertinent *temporal graph* of a process instance P is a directed acyclic graph $G = (V, E)$, where V is the set of activities in P, and E is a set of edges. Each edge in G is an ordered pair (v_i, v_j), where $v_i, v_j \in V$, $v_i \neq v_j$, and v_i is directly followed by v_j.

Transforming a process instance into its corresponding temporal graph representation is straightforward. We first traverse the activities in the given process instance by the ascending order of their starting times. For each activity v, the set F of activities that directly follow v are identified. Subsequently, edges connecting v to each activity in F are created. As shown in Figure 1(a), activity B will be processed first due to its earliest starting time among all activities in the process. Activities C and D directly follow B; thus, two edges are created from B to C and D, respectively, as shown in Figure 1(b). The subsequent traversal of this process instance processes activities A, C, D, and E in sequence. The resulting temporal graph corresponding to this process is graphically illustrated in Figure 1(b). From a given temporal graph G, it is evident that an activity v_i is followed by another activity v_j if and only if there exists a path from v_i to v_j in G, and v_i and v_j are overlapped otherwise. As shown in Figure 1(b), activity B is followed by E since there exists a path from B to E. In contrast, activities A and B are overlapped since there does not exist a path between them.

A temporal pattern can also be represented as a temporal graph that has a certain user-specified minimum support and satisfies the maximality property.

Definition 6. A temporal graph G is said to be *supported* by a process instance P if all followed and overlapped relationships that exist in G are present in P.

Definition 7. Given a set of process instances, a temporal graph *G* is said to be *frequent* if it is supported by no less than *s*% of the process instances, where *s*% is a user-defined minimum support threshold.

Definition 8. A temporal graph *G*=(*V*, *E*) is a *temporal subgraph* of another temporal graph *G'*=(*V'*, *E'*) if $V \subseteq V'$ and for any pair of vertices $v_1, v_2 \in V$, there is a path in *G* connecting v_1 to v_2 if and only if there is a path in *G'* connecting v_1 to v_2. If *G* is a temporal subgraph of *G'*, then *G'* is a *temporal supergraph* of *G*.

Definition 9. Given a set *TGS* of temporal graphs, a temporal graph *G* in *TGS* is *maximal* if *G* is not a temporal subgraph of any other temporal graph in *TGS*.

Problem statement: Given a set of process instances, the temporal pattern discovery is to find the maximal temporal graphs among all frequent temporal graphs. Each such temporal graph is referred to as a temporal pattern.

4. Temporal Pattern Discovery Algorithms

In this section, three different algorithms, namely TP-Graph, TP-Itemset, and TP-Sequence, are proposed for the described temporal pattern discovery problem. TP-Graph precedes its discovery process directly based on the temporal graph representation. On the other hand, TP-Itemset extends the Apriori algorithm (Agrawal and Srikant, 1994) for discovering temporal patterns from a set of process instances, each of which is represented as a set of temporal relationships. Finally, in the TP-Sequence algorithm, each process instance is represented as a quasi-sequence where the overlapping and followed-by relationships in each process instance are properly preserved. Accordingly, a sequential pattern discovery technique, specifically the AprioriAll algorithm (Agrawal and Srikant, 1995), is extended to discover temporal patterns from the set of quasi-sequences.

4.1 TP-Graph Algorithm

As with association rule (Agrawal and Srikant, 1994) and sequential pattern (Agrawal and Srikant, 1995) algorithms, the TP-Graph algorithm exploits the downward closure property of the support measure to improve the efficiency of searching for frequent temporal graphs. The downward closure property suggests that if a temporal graph G has support of at least s%, any temporal subgraph of G must have a support of at least s%. In other words, if a temporal graph G has a support of less than s%, any temporal supergraph of G definitely will have support of less than s%. Accordingly, we adopted an iterative procedure similar to that in the Apriori (Agrawal and Srikant, 1994) and AprioriAll (Agrawal and Srikant, 1995) algorithms. Specifically, potentially frequent temporal graphs (or called *candidate temporal graph*) of size k are constructed by joining frequent temporal graphs of size $k–1$. The process instances are then scanned to identify frequent temporal graphs of size k from the set of candidate temporal graphs of the same size. The resultant frequent temporal graphs are then used to prune the non-maximal frequent temporal graphs derived in the previous iteration (i.e., $k–1$). This procedure is iteratively executed until no further frequent temporal graphs can be found. Let C_k and L_k denote the set of candidate temporal graphs and the set of frequent temporal graphs of size k, respectively. Each iteration k performs the following three steps whose challenges and solutions are detailed in the following subsections, respectively.

1. If k=1, C_k is the set of all single-activity temporal graphs. Otherwise, C_k is generated by joining in pair-wise the frequent temporal graphs of size $k–1$ (i.e., L_{k-1}).
2. Scan the process instances to determine L_k from C_k.
3. If k>1, prune from L_{k-1} all non-maximal temporal graphs that are temporal subgraphs of any temporal graph in L_k.

4.1.1 Joining Frequent Temporal Graphs

Intuitively, two frequent temporal graphs of size $k–1$ can be joined if they differ only in one activity and contain the same temporal relationships for any pair of common activities. However, this simple-minded joining process will result in many redundant candidate temporal graphs. Consider the following example. Suppose the set of frequent temporal graphs in iteration 2 be $\{A \rightarrow B$[1], $B \rightarrow C$, $A \rightarrow C\}$. Any pair in the set can be joined to form the candidate temporal graph of $A \rightarrow B \rightarrow C$. That is, three identical candidate temporal graphs of size 3 will be generated. In the following, a joining algorithm is proposed to eliminate or reduce such redundancy.

Definition 10. Let *G* be a temporal graph and *v* be a vertex in *G*. The operation of subtracting *v* from *G*, denoted as $G-\{v\}$, deletes *v* and its associated edges from *G*. In addition, transitive edges via *v* are reconstructed by connecting each source vertex of incoming edges of *v* to each destination vertex of outgoing edges of *v*.

This subtraction operation can be illustrated as follows. Figure 2(b)-(f) show all of the temporal subgraphs resulted from subtracting a vertex from the temporal graph *G* shown in Figure 2(a). When the vertex *B* is subtracted from *G*, edges $A \rightarrow C$ and $A \rightarrow D$ are reconstructed as shown in Figure 2(b). As shown in Figure 2(c), the deletion of the vertex *C* from Figure 2(a) does not introduce any new edge since *C* does not have any outgoing edge in *G*. Figure 2(d), (e), and (f) illustrate the remaining temporal subgraphs derived from Figure 2(a) by deleting *D*, *E*, and *A*, respectively.

Observation 1. Let *s* be a vertex without incoming edges (called a source vertex) and *e* be another vertex without outgoing edges (called a sink vertex) in a temporal graph *G*. If *G* is frequent, both $G-\{s\}$ and $G-\{e\}$ must be frequent.

[1] This simplified representation differs from the temporal graph representation defined in Definition 5. Here, $A \rightarrow B$ denotes a temporal graph consisting of activities *A* and *B* where *A* is directly followed by *B*.

Based on this observation, to determine whether two frequent temporal graphs can be joined, only their source vertices and sink vertices need to be considered. Accordingly, we formally define joinable temporal graphs as follows.

Definition 11. Two temporal graphs G_i and G_j are said to be *joinable*, if there exists a source vertex s in G_i and a sink vertex e in G_j such that $G_i-\{s\} = G_j-\{e\}$ and $s \neq e$.

Consider the temporal graphs shown in Figure 3. Designating vertex B as a source activity of G_1 shown in Figure 3(a) and vertex D as a sink activity of G_2 shown in Figure 3(b), these two temporal graphs are *joinable* since $G_1-\{B\} = G_2-\{D\}$. The temporal graphs G_1 and G_3 or G_2 and G_3, however, are not joinable.

Given two joinable temporal graphs G_i (with s being a source vertex) and G_j (with e being a sink vertex), the temporal relationship between any pair of activities (except that between s and e) present in G_i or G_j will be preserved in a resulting candidate temporal graph. Since more than one permissible temporal relationship between s and e may exist, the joining of two joinable temporal graphs of size k–1 can lead to multiple candidate temporal graphs of size k. The temporal relationship between s and e in a candidate temporal graph can be: 1) no edge exists between s and e or 2) an edge connects s to e. Note that the case where an edge connects e to s need not be considered, as it results in a temporal graph with s and e not being source and sink vertices respectively. Formally, the join set of two joinable temporal graphs G_i (with s being the source vertex) and G_j (with e being the sink vertex) is composed of

1. $G_i \cup G_j$[2], and
2. $G_i \cup G_j \cup \{s \rightarrow e\}$ if there does not exist a path from s to e in $G_i \cup G_j$.

Consider the two joinable temporal graphs G_1 and G_2 shown in Figure 3. The join set of G_1

[2] The union of two graphs $G_i=(V_i, E_i)$ and $G_j=(V_j, E_j)$ results in a new graph $G=(V_i \cup V_j, E_i \cup E_j)$.

(with B being a source vertex) and G_2 (with D being a sink vertex) includes two candidate temporal graphs of size 4 as shown in Figure 4.

The previously described downward closure property can further be exploited to reduce the set of resulting candidate temporal graphs. A candidate temporal graph G of size k will not be frequent if any of its temporal subgraphs of size k–1 is not in L_{k-1} and, hence, should be eliminated from C_k. Such pruning process requires, for each candidate temporal graph of size k, the derivation (using the subtraction operation defined in Definition 10) of all of its temporal subgraphs of size k–1. The pseudo code of *GenerateCandidateGraph*() for generating a set of candidate temporal graphs of size k from a set of frequent temporal graphs of size k–1 and that of *DeriveSubgraph*() for deriving all temporal subgraphs of size $|G|$–1 for a temporal graph G are listed below.

```
GenerateCandidateGraph(a set of frequent temporal graphs: TGS): a set of temporal graphs
{
    CandidateSet = Ø;
    For (each pair of graphs ($G_i$, $G_j$) in TGS) {
        For (each source vertex s in $G_i$) {
            For (each sink vertex e in $G_j$) {
                If ($s \neq e$ and $G_i – \{s\} = G_j – \{e\}$) { /* joinable */
                    $UG1 = G_i \cup G_j$; $UG2 = G_i \cup G_j \cup \{s \rightarrow e\}$;
                    CandidateSet = CandidateSet ∪ {UG1};
                    If there exists no path from s to e in UG1
                    Then CandidateSet = CandidateSet ∪ {UG2};
                } /* end-if */
            } /* end-for */
        } /* end-for */
    } /* end-for */
    For (each graph G in CandidateSet) {
        If DeriveSubgraph(G) ∩ TGS ≠ DeriveSubgraph(G)
        Then CandidateSet = CandidateSet – {G};
    } /* end-for */
    Return CandidateSet;
}
```

```
DeriveSubgraph(a temporal graph: G): a set of temporal graphs
{
    Subgraph = Ø;
    For (each vertex v in G) {
        Source = the set of vertices incident to v;
        Sink = the set of vertices incident from v;
        SG = G – {v};
        For (each vertex pair ($v_s$, $v_d$) where $v_s \in$ Source and $v_d \in$ Sink) {
            If there does not exist a path between $v_s$ and $v_d$ in SG then SG = SG $\cup \{v_s \rightarrow v_d\}$;
        } /* end-for */
        Subgraph = Subgraph $\cup$ {SG};
    }
    Return Subgraph;
}
```

4.1.2 Scanning Process Instances

To find frequent temporal graphs from a set of candidate temporal graphs, we have to compute their support by scanning the set of process instances. To efficiently decide the set of candidate temporal graphs that a given process instance supports, we adopted the *hash-tree* data structure proposed by Agrawal and Srikant (1994 and 1995). Use of the hash-tree to store candidate temporal graphs of the same size requires a total order on the vertices in each temporal graph. In this study, the vertices of each temporal graph are sorted based on its graph topology. A topological sort of graph *G* is a linear ordering of all its vertices such that if *G* contains an edge (*u*, *v*), then *u* appears before *v* in the ordering (Cormen et al., 1989). To ensure a unique topological sort for a given temporal graph, the lexicographic order is applied to vertices that are temporally overlapped. The resulting order is called the temporal sequence of a temporal graph. For instance, the temporal sequence of the temporal graph shown in Figure 4(a) is <*A, B, C, D*>.

A node in the hash-tree either contains a set of temporal graphs (a leaf node) or a hash table (an interior node). Each bucket in the hash table of an interior node points to a child node. To insert a candidate temporal graph *G*, we start from the root and follow appropriate pointers until a leaf node is reached. At an interior node at depth *d* (assuming the depth of the root node of the hash-tree be 1 and that of a child node of an interior node at depth *d* be *d*+1), we decide which branch to follow by applying a hash function to the *d*-th vertex in the temporal sequence of *G*. Initially, the root node is a leaf node. When a leaf node *L* at depth *d* overflows (i.e., the number of temporal graphs in the leaf node exceeds a specified threshold), *L* is converted to an interior node and several leaf nodes are created as the child nodes of *L*. Subsequently, the temporal graphs originally stored in *L* are distributed to these leaf nodes by applying the hashing function to the *d*-th vertices in their temporal sequences.

Such a hash-tree, once constructed, can be used to determine the subset of candidate temporal graphs that is supported by a given process instance *P* by traversing the hash-tree. Let *S* denote the temporal sequence of *P*. The traversal starts at the root node by applying the hashing function on every vertex in *S* to determine the set of nodes at depth 2 to visit. At an interior node to which a vertex *a* in *S* has just hashed, the hashing function is then applied to each vertex after *a* in *S*. The traversal process continues until leaf nodes are reached. At each leaf node reached, the candidate temporal graphs in the leaf supported by *P* are identified and their support counts are incremented by one. After scanning all the process instances, the candidate temporal graphs whose support exceeds the user specified minimum threshold form the set of frequent temporal graphs of this iteration.

Consider a segment of hash-tree for candidate temporal graphs of size 3 as shown in Figure 5(b). By hashing on every vertex in the temporal sequence <*B*, *C*, *E*, *D*> of the process instance *P* shown in Figure 5(a), we examine those nodes that start with *B*, *C*, *E*, or *D*, respectively. In this case, the nodes 3 and 4 will be visited next. At node 3 in the hash-tree shown in Figure 5(b), we can only hash on vertex *C*, *E* and *D* since we have reached node 3 by previously hashing on vertex *B*. As a result, node 7 will then be visited. On the other hand, since node 4 is a leaf node, whether its candidate temporal graph (i.e., G_6) is supported by *P* is then examined. Similarly, the temporal graphs (i.e., G_4 and G_5) in node 7 will be examined against *P* and the support of G_4 is incremented by 1 since it is supported by *P*.

The pseudo code for inserting a candidate temporal graph into a hash-tree, named *AddOneGraph*(), and that for traversing the hash-tree for a given process instance, named *Traverse*(), are listed below. Note that *Traverse*() is a recursive function that takes three parameters, namely the current node *C* of the hash-tree, the target process instance *P*, and the position of the vertex in *P* that previously hashed to *C*. Initially, we call *Traverse*(*root* of the hash-tree, *P*, 0).

```
AddOneGraph(a hash-tree: T, a temporal graph: G with temporal sequence $<v_1, v_2, ..., v_n>$)
{
    C = root of T; Level = 1; /* initialize */
    While C is not a leaf node of T do {
        C= C.Hash($v_{Level}$);
        Level++;
    }
    Insert G into C;
    If C is full
    {
        Create a hash table H with each entry pointing to a new leaf node;
        For each temporal graph R in C
        {
            Assign R to a leaf node by hashing on the Level'th vertex of R's temporal sequence;
        } /* end-for */
        Assign the hash table H to C;
  } /* end-if */
}

Traverse(a node pointer: C, a process instance: P with temporal sequence $<v_1, v_2, ..., v_n>$,
previous position: s)
{
    If (C is a leaf node) {
        For (each temporal graph G in C) {
            If G is supported by P then G.count++;
        } /* end-for */
    }
    else {
        position = s;
        Do {
            NewC = C.Hash($v_{position}$);
            Traverse(NewC, P, position+1);
            position++;
        } While ($position \le n$);
    } /* end-if */
}
```

Theorem 1 shows that this traversal procedure indeed returns the desired result.

Lemma 1. For each candidate temporal graph G supported by a process instance P, the temporal sequence of G must be a subsequence[3] of the temporal sequence of P.

Proof: Let S_G = <v_1, v_2, …, v_k> and S_P be the temporal sequences of G and P, respectively. For any pair of vertices v_i and v_j in S_G where $i < j$, there are two possibilities on their temporal relationships: v_i is followed by v_j in G, and v_i and v_j are overlapped but v_i precedes v_j lexicographically. Since P supports G, it is clear that in either case the same relationship holds between v_i and v_j in P. Therefore, v_i appears before v_j in S_P. □

Theorem 1. For a given process instance P, the traversal of the hash-tree examines every candidate temporal graph supported by P.

Proof: It is obvious that the traversal of the hash-tree for P exhausts all the subsequences of the temporal sequence of P. According to Lemma 1, these nodes include all the candidate temporal graphs supported by P. □

Use of a hash-tree at iteration k can not only facilitate fast counting the support of candidate temporal graphs of size k but also improve efficiency of constructing the set of candidate temporal graphs of size k+1. Considering the hash-tree for candidate temporal graphs of size 3 shown in Figure 5, suppose that the temporal graph G_1 (i.e., $A \rightarrow B \rightarrow C$) has been found to be frequent. To find temporal graphs that are joinable with G_1, we need only to visit those leaf nodes that are reachable by hashing on B followed by C (assuming A is selected as the source vertex in G_1). In this case, both temporal graphs contained in the node 7 (i.e., G_4 and G_5) are joinable with G_1. This search process of which correctness is ensured by Theorem 2 greatly reduces the overhead required to establish joinable temporal graphs for a given temporal graph.

Theorem 2. Let G be a frequent temporal graph with the temporal sequence being $<v_1, v_2, \ldots, v_k>$. There must exist two joinable frequent temporal subgraphs G_1 and G_2 with temporal sequences being $<v_1, v_2, \ldots, v_{k-1}>$ and $<v_2, v_3, \ldots, v_k>$ respectively. In addition, the join set of G_1 and G_2 includes G.

Proof: It is obvious from Observation 1 that subtracting v_k from G results in a frequent temporal subgraph G_1 with the temporal sequence being $<v_1, v_2, \ldots, v_{k-1}>$ and that subtracting v_1 from G results in a frequent temporal subgraph G_2 with a temporal sequence $<v_2, v_3, \ldots, v_k>$. Besides, since $G_2-\{v_k\} = G_1-\{v_1\}$, G_1 and G_2 are joinable, and their join set includes G. □

4.1.3 Pruning Non-maximal Temporal Graphs

Apparently, if a temporal graph G is frequent, all of its temporal subgraphs also are frequent. According to Definition 9, the temporal subgraphs of G are not maximal and should be pruned since they reveal less information on frequent activities and temporal relationships than G. To retain only maximal temporal graphs, *PruneSubgraph*() is invoked at each iteration to eliminate any non-maximal temporal graphs obtained at the previous iteration.

```
/* $TGS_n$ is the set of frequent temporal subgraphs obtained at iteration $n$ */
PruneSubgraph(a set of graphs: $TGS_{n-1}$, a set of graphs: $TGS_n$): a set of temporal graphs
{
    For (each graph $G$ in $TGS_n$) {
        $SG$ = DeriveSubgraph($G$)[4];
        $TGS_{n-1} = TGS_{n-1} - SG$;
    } /* end-for */
    Return $TGS_{n-1}$;
```

[3] A sequence $X = <x_1, x_2, \ldots, x_m>$ is a subsequence of another sequence $Y = <y_1, y_2, \ldots, y_n>$ if there exists a strictly increasing sequence $<i_1, i_2, \ldots, i_m>$ of indices of Y such that for all $j = 1, 2, \ldots, m$, we have $x_j = y_{i_j}$ (Cormen et al., 1989).

[4] For ease of illustration, *PruneSubgraph*(TGS_{n-1}, TGS_n) invokes *DeriveSubgraph*(G) to determine all the temporal subgraphs of G by dropping a vertex in G. However, by maintaining an appropriate data structure that keeps the links between a temporal graph and its subgraphs when generating candidate temporal graphs, *DeriveSubgraph*() need not be invoked and this step can efficiently be performed.

}

4.2 TP-Itemset Algorithm

TP-Itemset extends the Apriori algorithm (Agrawal and Srikant, 1994) for discovering temporal patterns from a set of process instances. In TP-Itemset, each possible temporal relationship in a process instance P is explicitly represented as an item in the itemset for P. Each item in an itemset is of the form $v_i \rightarrow v_j$ if the activity v_i is followed by v_j or $v_i \sim v_j$ if the durations of activities v_i and v_j are temporally overlapped (where ~ denotes an overlapping relationship and $v_i < v_j$ in their lexicographical order). For example, as shown in Figure 6, the process instance 1 is represented as $\{A \sim B, A \rightarrow C, B \rightarrow C\}$, while the process instance 2 is represented as $\{A \sim B, A \rightarrow C, B \sim C\}$. With this representation, $n(n-1)/2$ items are required to represent a process instance possessing n activities.

As with the association rule discovery technique, an itemset that has certain user-specified minimum support is called a large itemset, while a potentially large itemset is called a candidate itemset. The TP-Itemset algorithm is similar to the Apriori algorithm but with several distinctions. First, unlike in the Apriori algorithm where resulting large itemsets are unrestricted, a large itemset generated by the TP-Itemset algorithm needs to satisfy additional constraints. Let $V_I = \{v_1, v_2, \ldots, v_k\}$ be the set of distinct activities involved in an itemset I. An itemset I is referred to as a *full* itemset if the temporal relationship between any pair of activities, v_i and v_j where $v_i \in V_I$, $v_j \in V_I$, and $v_i \neq v_j$, exists in I. Otherwise, I is a *partial* itemset due to the absence of some temporal relationships in I. In effect, each full and large itemset corresponds to a frequent temporal graph defined in Definition 7. For instance, $\{A \sim B, A \rightarrow C, B \rightarrow C\}$ is a full itemset, while $\{A \sim B, A \rightarrow C\}$ is a partial itemset since the relationship between activities B and C is unspecified. Hence, large itemsets generated by the TP-Itemset algorithm are required to be full itemsets. Second, the set of large itemsets generated by the Apriori algorithm need not retain only the maximal itemsets each of which is not a subset of any other large itemsets. However, since temporal pattern discovery is interested only in finding the maximal temporal patterns among all frequent temporal patterns, pruning non-maximal itemsets is deemed necessary.

Given a set of process instances, the TP-Itemset algorithm transforms them into a set of full itemsets and generates the maximal itemsets among all full and large itemsets. Each such maximal, full and large itemset represents a temporal pattern. Let L_k be a set of large k-itemsets each of which has k items (i.e., k temporal relationships) and C_k be a set of candidate k-itemsets, where C_k can be constructed by joining large itemsets in L_{k-1}. In the Apriori algorithm, the joining procedure requires that items within an itemset be kept in their lexicographic order. In this study, since each item ($v_i \rightarrow v_j$ or $v_i \sim v_j$) involves two activities, the lexicographical order of a set of items is based on their first activities (i.e., v_i) and then on their second ones (i.e., v_j). Moreover, the partial itemsets from L_{k-1} should not be removed immediately at each iteration k–1, since two partial (k–1)-itemsets may result in a full itemset in C_k. For instance, joining two partial, large itemsets in L_2, $\{A \rightarrow B, A \rightarrow C\}$ and $\{A \rightarrow C, B \sim C\}$, results in a full itemset $\{A \rightarrow B, A \rightarrow C, B \sim C\}$. Finally, to facilitate fast counting the support for the candidate itemsets in C_k, candidate itemsets are stored in a hash-tree as employed by the Apriori algorithm (Agrawal and Srikant, 1994). Accordingly, the TP-Itemset algorithm for discovering temporal patterns is listed below.

```
TP-Itemset(a set of process instances: $P$, the minimum support: minsup): a set of large, full and
maximal itemsets
{
    Transform each process instance $p \in P$ into a process itemset $i$ in $I$;
    $L_1$ = {large 1-itemsets};
    $MaxK = 1$;
    For ($k = 2$; $L_{k-1} \neq \emptyset$; $k$++)
    {
        Generates candidate itemsets $C_k$ from $L_{k-1}$;
        For each process itemset $i \in I$
        {
            $C_t$ = subset[5]($C_k$, $i$); /* find candidate itemsets that are supported by $i$ */
            For each candidate $c \in C_t$ do $c$.count++;
        } /* end-for */
        $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$;
        If ($L_k \neq \emptyset$) then $MaxK = MaxK+1$;
    } /* end-for */
    For ($k$=$MaxK$; $k$>1; $k$--)
    {
        Prunes partial itemsets in $L_k$;
        For each itemset $s$ in $L_k$
        {
            Prunes all sub-itemsets of $s$ from $L_{k-1}, L_{k-2}, \ldots, L_1$;
        } /* end-for */
    } /* end-for */
    Return $\cup_{k\geq 1} L_k$;
}
```

4.3 TP-Sequence Algorithm

TP-Sequence is based on the sequential pattern discovery technique (specifically, the AprioriAll algorithm) to discover temporal patterns from a set of process instances. In the TP-Sequence algorithm, the overlapped and followed relationships in each process instance are explicitly represented as a sequence, where an itemset is a non-empty set of overlapping activities, and a sequence is an ordered list of itemsets. The itemset $\{x, y\}$ denotes that activities x and y are temporally overlapped. A sequence with an ordered list of k itemsets is called a k-sequence. For instance, a 2-sequence $<\{x\}\{y\}>$ denotes that the activity x is followed by y. Furthermore, a 2-sequence $<\{x, y\}\{y, z\}>$ denotes that activity x is followed by z, while y overlaps with x and z. As shown in Figure 6, the process instance 1 is represented as a sequence of $<\{A, B\}\{C\}>$, while the process instance 2 is represented as $<\{A, B\}\{B, C\}>$. Using this representation, a process instance is represented as an m-sequence where $l \leq m \leq k$, l is the

[5] Using the hash-tree constructed for C_k, the subset(C_k, i) function is to find all the candidate itemsets in C_k that are supported by the itemset i. We employed and implemented the subset function as proposed in (Agrawal and Srikant, 1995).
[7] In pass 2 and 3, the difference on the number of candidates between TP-Graph and TP-Sequence was less than 0.02%, while at pass 11, TP-Sequence generated about 10% more candidates than TP-Graph did.

number of activities in the longest path in the respective temporal graph, and k is the number of activities in the process instance.

In the traditional sequential pattern discovery, no specific constraint is imposed on itemsets in a sequence. However, since a sequence in the TP-Sequence algorithm is used to represent both the followed and overlapped relationships, a meaningful sequence needs to satisfy certain constraints. We call a sequence $<\{x\}\{x, y\}>$ a non-canonical sequence since it is identical to $<\{x, y\}>$. We say a sequence $<\{x, y\}\ \{y, z\}\ \{x, z\}>$ is an *illegitimate* sequence since x appears over discontinuous intervals. Furthermore, an illegitimate sequence such as $<\{x, y\}\ \{w\}\ \{x, z\}>$ even exhibits contradictory temporal relationships as both "x followed by w" and "w followed by x" exist; thus, violating the irreflexivity of followed relationships. Hence, for discovering temporal patterns, the AprioriAll algorithm needs to be extended to ensure that any candidate sequence generated be canonical and legitimate. Let a_i be an itemset. Canonical and legitimate sequences are formally defined as follows.

Definition 12. A sequence $s = <a_1\ a_2\ \dots\ a_m>$ is canonical if for each itemset a_j, $1 \leq j < m$, $a_j \not\subset a_{j+1}$ and $a_{j+1} \not\subset a_j$.

Definition 13. A sequence $s = <a_1\ a_2\ \dots\ a_m>$ is *legitimate* if for each item x involved in s, the itemsets that contain x form a continuous subsequence in s.

To distinguish an unconstrained sequence from a sequence with canonicity and legitimacy properties (as required by the TP-Sequence algorithm), the latter is called a *quasi-sequence*. The transformation of a process instance P into its respective quasi-sequence proceeds in the following iterative manner. The quasi-sequence is initialized as an empty list. We traverse the starting and ending times of activities in P in ascending order. The set O of overlapped activities starts to accumulate when the first starting time is visited. When the first ending time is encountered, the set O is appended to the quasi-sequence. Subsequently, we continue the traversal until the next starting time (assuming its respective activity be *v*) is visited. At this moment, the subset of activities in *O* whose ending times appear before the starting time of *v* are removed from *O* since this subset of activities that have appeared in the quasi-sequence are followed by *v*. This traversal procedure continues until all the timestamps are visited. Let us illustrate this transformation using the process instance shown in Figure 7(a). As shown, when the first ending time (which belongs to *A*) is visited, *O*={*A*, *B*} and, thus, the current quasi-sequence is <{*A*, *B*}>. Since only *A*'s ending time appears before the next starting time (that pertains to *C*), only *A* is removed from *O*. When the next ending time (that pertains to *B*) is visited, *O*={*B*, *C*, *D*} and, therefore, the quasi-sequence becomes <{*A*, *B*}{*B*, *C*, *D*}>. When the following starting time (that is possessed by *E*) is visited, *O* becomes empty because the ending times of *B*, *C*, and *D* have all been traversed. When the last ending time (which belongs to *E*) is visited, *O*={*E*}, and the resultant quasi-sequence is <{*A*, *B*}{*B*, *C*, *D*}{*E*}> as shown in Figure 7(b). The pseudo-code of the described transformation is listed in the following.

```
Generate-Quasi-Sequence(a process instance: P): a quasi-sequence
{
    Quasi-Seq = <>; /* Initialization of a quasi-sequence */
    Sort the timestamps of activities in P and place them in a queue time;
    O = Ø;
    While time ≠ null {
        if time.event = 'starting time' {
            add time.activity to O;
            time = time.next;
        }
        else {/* time.event = 'ending time' */
            Append O to Quasi-Seq;
            While (time ≠ null) and (time.event = 'ending time') {
                O = O − time.activity;
                time = time.next;
            } /* end-while */
        } /* end-if */
    } /*end-while */
    Return Quasi-Seq;
}
```

In a quasi-sequence, when an activity *v* appears in two consecutive itemsets I_j and I_{j+1}, *v* is temporally overlapped with the remaining activities in I_j and I_{j+1}, rather than v in I_j taking place before the activities in I_{j+1} and v in I_{j+1} occurring after the activities in I_j. This unique interpretation requires re-definition of the subsequence relationship used by the sequential pattern discovery algorithm. Using the example shown in Figure 7, the process instance is represented as a 3-quasi-sequence $<\{A, B\}\{B, C, D\}\{E\}>$. In the sequential pattern discovery, the sequence $<\{B\}\{C\}>$ is considered to be supported by (or a subsequence of) $<\{A, B\}\{B, C, D\}\{E\}>$. However, the quasi-sequence $<\{B\}\{C\}>$ indeed denotes a followed relationship between B and C, which differs from an overlapped relationship in the process instance under discussion. Thus, the quasi-sequence $<\{B\}\{C\}>$ is not supported by $<\{A, B\}\{B, C, D\}\{E\}>$ in the temporal pattern discovery. The re-defined subsequence relationship (formally defined in Definition 14) is needed when determining support for candidate quasi-sequences as well as pruning non-maximal quasi-sequences.

Definition 14. A quasi-sequence $s = <a_1\ a_2\ \ldots\ a_n>$ is supported by (or called *a subsequence of*) another quasi-sequence $t = <b_1\ b_2\ \ldots\ b_m>$, if there exists integers $i_1 < i_2 < \ldots < i_n$ such that $a_1 \subseteq b_{i1}$, $a_2 \subseteq b_{i2}$, …, $a_\text{n} \subseteq b_{i_n}$, and there exist no consecutive itemsets a_j and a_{j+1} in s such that $v \in a_j$, $u \in a_{j+1}$, and the itemset $\{u, v\} \subseteq b_{ij}$ or $\{u, v\} \subseteq b_{ij+1}$.

Accordingly, the TP-Sequence algorithm, extending the AprioriAll algorithm, finds the maximal quasi-sequences among all frequent quasi-sequences. Each such quasi-sequence corresponds to a temporal pattern. The TP-Sequence algorithm employs the same hash-tree data structure as the AprioriAll algorithm for storing candidate quasi-sequences in C_k and fast counting their support (Agrawal and Srikant, 1995). The pseudo-code for the proposed TP-Sequence algorithm is listed as follows.

```
TP-Sequence (a set of process instances: TPS, the minimum support: minsup): a set of large
quasi-sequences
{
    QSS = Ø; /* QSS contains a set of quasi-sequences for process instances */
    For each process instance P in TPS
    {
        QSS = QSS ∪ Generate-Quasi-Sequence(P);
    } /* end-for */
    $L_1$ = {large 1-quasi-sequences};
    For ($k$ = 2; $L_{k-1} \neq Ø$; $k$++)
    {
        $C_k$ = candidate sequences generated from $L_{k-1}$;
        Delete non-canonical and illegitimate sequences in $C_k$;
        For each quasi-sequence qs in QSS
        {
            Increment the count of all candidates in $C_k$ that are supported by qs;
        } /* end-for */
        $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$;
        For each quasi-sequence s in $L_k$ do
        {
            Delete all sub-sequences of s from $L_{k-1}$;
        } /* end-for */
    } /*end-for */
    Return $\cup_{k\geq 1} L_k$;
}
```

5. Performance Evaluation

In this section, we evaluate the performance and scale-up properties of the three proposed algorithms for finding temporal patterns. The experiments were conducted on an IBM compatible PC with a CPU clock rate of 500 MHz and 128 MB of main memory, running FreeBSD 4.1. Since we intend to examine the performance and scalability of the proposed algorithms over a wide range of data characteristics, finding real-world data sets for our evaluation would have been extremely difficult, if not impossible. Thus, synthetic data sets were generated and employed for the evaluation.

5.1 Generation of Synthetic Data

To generate synthetic data set consisting of process instances, we adopted and extended the transaction generation model proposed in (Agrawal and Srikant, 1994; Agrawal and Srikant, 1995; Srikant and Agrawal, 1996) for evaluating the Apriori and AprioriAll algorithms. In our model of process executions, process instances are not randomly designed but tend to contain sets of temporally related activities, each of which is a potential temporal pattern. Furthermore, process instances are generated based on these potential temporal patterns. However, a process instance might include only a subset of activities from a potential temporal pattern.

Given a set *A* of available activities with size *N*, we first generate a pool of potential temporal patterns. The number of such patterns generated is set to *PN*. A potential temporal pattern *P* is generated by first determining its size (i.e., the number of activities) from a Poisson distribution with mean equal to *PS*. Activities in the first potential temporal pattern are chosen randomly. To model the phenomenon where temporal patterns may involve common activities, some percentage of activities in *P*, determined by an exponentially distributed random variable with mean equal to the correlation ratio (*CR*), are randomly chosen from the potential temporal pattern *Q* generated immediately prior to *P*. Subsequently, the remaining activities in *P* are selected randomly without repetition from the rest of activities in *A* (i.e., excluding all activities in *Q*). In addition, to determine temporal relationships among those activities in *P*, some fraction of activities are chosen and arranged in sequence; thus exhibiting followed relationships. We use an exponentially distributed random variable with mean equal to the length ratio *LR*, defined as the maximal number of sequential activities to the total number of activities, to decide this fraction for each pattern. Furthermore, without loss of generality, we assume the execution duration of each such sequential activity in *P* be identical. For each remaining activity in *P*, its execution interval e_i, bounded by the earliest starting time and the latest ending time of the sequential activities decided previously, is randomly determined, thus creating overlapped relationships. However, e_i should not reside in the time gap between any two consecutive sequential activities in order to preserve the pre-decided length ratio for *P*.

After the generation of the set of potential temporal patterns, each pattern is assigned a weight, which corresponds to its probability of being selected when generating a process instance. The weight is initially picked from an exponential distribution with unit mean and then normalized so that the sum of the weights for all of the patterns is 1. Finally, a set of process instances is generated. The size of a process instance is picked from a Poisson distribution with mean equal to *IS*. For each process instance, one of the potential temporal patterns is randomly chosen by tossing a *PN*-sided weighted coin, where the weight for a side is the probability of picking the associated pattern. If the size of the chosen pattern is not the same as that of the instance, surplus activities are randomly dropped or additional activities are randomly added in an overlapped manner.

The parameters and their respective default values used for the generation of synthetic data sets are summarized in Table 1. Depending on the type of experiments conducted, the respective parameter will be examined over a range of values, while the rest of parameters adopt their default values. For each particular experiment, 10 trials were performed and the overall performance was then estimated by averaging the performance across all trials.

5.2 Effects of Minimum Support Thresholds

Ten synthetic data sets were generated using the default values for all parameters as depicted in Table 1. We investigated the effects of minimum support thresholds, ranging from 2% to 10% at increments of 2%, on the execution times of each proposed algorithm. Figure 8(a) shows the execution times of the three proposed algorithms as a function of minimum support. As expected, the execution times of the three algorithms decreased as the minimum support increased. Over the range of minimum supports investigated, a decrease in minimum support appeared to have shown marginal effects on the execution times of TP-Graph and TP-Sequence. On the other hand, a decrease in minimum support resulted in a noticeable increase in the execution times of TP-Itemset. Across the range of minimum supports examined, TP-Graph appeared to have exhibited the best performance, while TP-Itemset performed worst, mainly because it generated and counted a much larger number of candidates than the other two algorithms. As shown in Figure 8(b), when the minimum support was 2%, the number of candidates generated and the number of iterations (i.e., passes) taken by TP-Itemset were significantly higher than those produced/required by its counterparts. Such dramatic differences could be attributed to their underlying structures for representing and manipulating process instances and candidates. TP-Itemset explicitly represents each temporal relationship (followed or overlapped) as an item in an itemset and generates candidates at the temporal relationship level. Thus, the size of C_1 (i.e., containing all possible relationships between pairs of activities) considered by TP-Itemset is $3n(n–1)/2$, where n is the number of activities, leading to even larger candidate sets in the first few passes. TP-Graph and TP-Sequence form a candidate temporal graph and a candidate quasi-sequence, respectively, by adding an additional activity from the previous iteration. Hence, the number of activities considered in pass 1 by either algorithm is n, which is far fewer than those generated by the TP-Itemset when n is large. On the other hand, assuming the maximal number of activities in the temporal patterns to discover to be s, the number of passes required by TP-Graph and TP-Sequence is at most s+1. However, the number of passes for generating and counting candidate itemsets would be $s(s–1)/2$ or higher. The larger candidate sets and higher number of passes considered by TP-Itemset resulted in its inferior performance. TP-Graph and TP-Sequence, which similarly represent and manipulate process instances and candidates, generated similar numbers of candidates at all iterations[7] and required the same number of passes for the target temporal pattern discovery, leading to superior performances measured by execution time. However, a more concise representation of process instances and temporal patterns employed in TP-Graph appeared to contribute to its better performance (about 20% faster) than TP-Sequence.

5.3 Effects of Process Characteristics

The performances of the three temporal pattern discovery algorithms were evaluated over a range of process characteristics described by the size of potential temporal patterns (*PS*), correlation ratio (*CR*), length ratio (*LR*), and number of activities (*N*) available for generating potential temporal patterns and process instances. We did not examine the effects of number of potential temporal patterns (*PN*) since varying the value of *PN* is similar to adjusting the minimum support threshold for a given value of *PN*.

Synthetic data sets were generated for various sizes of potential temporal patterns, ranging from 5 to 20 at increments of 5. Remaining parameters received their default values, as defined in Table 1. The minimum support was set to 2%. Figure 9(a) shows the execution times of the three temporal pattern algorithms as functions of the size of potential temporal patterns. The performance of the three algorithms remained largely stable across the sizes of potential temporal patterns examined. The resulting execution times of TP-Graph, while slightly lower than those of TP-Sequence, were significantly lower than those required by TP-Itemset. Various correlation ratios, ranging from 0.1 to 0.9 at increments of 0.1 were also investigated. At a minimum support of 2%, a steady performance was achieved by all of the proposed algorithms across all correlation ratios examined, as shown in Figure 9(b). As with the previous experiment, TP-Graph was relatively comparable to TP-Sequence and outperformed TP-Itemset.

In addition, we investigated the effects of length ratios, ranging from 0.1 to 0.9 at increments of 0.1, on the performance of the three algorithms. As shown in Figure 9(c), at a minimum support of 2%, the execution times of TP-Graph and TP-Itemset remained stable across different levels of length ratio examined. However, the execution times attained by TP-Sequence increased as length ratio grew from 0.1 to 0.9. A larger length ratio represents a scenario in which potential temporal patterns and their respected process instances were more likely to contain sequential activities; thus requiring a longer quasi-sequence for representing each process instance. As a result, as length ratio increased, the total size of itemsets involved in a quasi-sequence of a process instance or a temporal pattern increased and the performance of TP-Sequence degraded. Conversely, given the same set of activities appearing in a process instance, an increase in its length ratio did not increase the size of the resulting temporal graph or itemset. Thus, length ratio appeared to have no effect on the execution times of TP-Graph and TP-Itemset. Overall, TP-Graph was the most efficient algorithm, followed by TP-Sequence and finally TP-Itemset.

Finally, the effects of the numbers of activities (ranging from 400 to 1600 at increments of 200) available for generating potential temporal patterns and the process instances on the performance of the three algorithms were examined. The minimum support was again set to 2%. As shown in Figure 9(d), the performance of the three algorithms remained largely stable across different numbers of activities examined. The execution times needed by TP-Graph, largely comparable to those by TP-Sequence, were significantly lower than those attained by TP-Itemset.

5.4 Scale-up Experiments

The scalability experiments in this study were designed from two different perspectives: (1) by increasing the average size of process instances (*IS*) while keeping the number of process instances constant and (2) by increasing the number of process instances (*D*) while keeping the average size of process instances constant. The first scale-up experiment increased the average size of process instances, ranging from 10 to 60 at increments of 10. The remaining parameters received their default values as depicted in Table 1. Figure 10(a) shows the execution times required by TP-Graph, TP-Itemset and TP-Sequence, respectively, at a minimum support of

2%. We did not plot the execution times for TP-Itemset when the size of process instances was greater than 40, since TP-Itemset generated too many candidates and ran out of memory. When the size of process instances increased from 10 to 40, the execution time of TP-Itemset increased proportionally. However, the execution times of TP-Graph appeared to scale fairly quadratically across the range of sizes of process instances examined. Because each process instance is represented as a graph that requires quadratic manipulation, the execution time increases toward a quadratic trend as the size of process instances expands linearly. On the other hand, the execution times of TP-Sequence appeared to increase with the number of instances at a slower pace. Such a near-linear performance with respect to the size of process instances can be attributed to the linear manipulation of sequences of itemsets. When the size of process instances was below 30, TP-Graph was the most efficient algorithm. However, as the size of process instances exceeded 30, TP-Sequence exhibited better performance.

The second scale-up experiment varied the number of process instances (*D*), ranging from 10,000 to 50,000 at increments of 10,000, while adopting their default values for the remaining parameters. Figure 10(b) shows the performances of the proposed algorithms as a function of the number of process instances, at a minimum support of 2%. As shown, all of the proposed algorithms grew almost linearly with the number of process instances. The increasing rate for TP-Graph appeared to be the smallest, while TP-Itemset exhibited the worst performance and scalability with respect to the number of process instances.

6. Conclusions and Future Research Directions

As huge volumes of process data with temporal context are collected by and maintained in organizations, discovering within these data frequently occurring activities and their respective temporal relationships (referred to as temporal patterns in this study) is essential to establishing a foundation for reengineering a business process and managing workflow evolution and exceptions. The discovery of temporal patterns can also be applied to various application domains (e.g., healthcare and project management) for crucial business decision support. Motivated by the importance of and need for discovering such temporal patterns from process data, we formally defined the temporal pattern discovery problem, and developed and evaluated three different temporal pattern discovery algorithms, namely TP-Graph, TP-Itemset and TP-Sequence, for finding a set of temporal patterns from process instances.

Using synthetic data sets, we analyzed the performance, over a range of data characteristics, and scale-up properties of the three proposed algorithms. The experimental results showed that the size of potential temporal patterns, correlation ratio, length ratio and the number of available activities had no, or at most marginal, effects on the execution times of the proposed algorithms. Overall, TP-Graph appeared to achieve the best performance. Due to its representation and manipulation that treat each temporal relationship in a process instance as an individual item, TP-Itemset exhibited the worst performance. In terms of scale-up properties, the experimental results suggested that the execution times of TP-Sequence and TP-Itemset grew linearly as the size of process instances expanded linearly, while those of TP-Graph increased toward a quadratic growth. The experimental results also suggested that the three proposed algorithms scaled linearly with the number of process instances, with the TP-Graph algorithm achieving the best scalability.

Some ongoing and future directions along this line of research are summarized as follows. The described process mining problem concentrates on finding temporal patterns among activities; thus, only their respective temporal relationships in process instances were considered. However, activities in a process instance are often described by such properties as execution entity(s) involved, execution location, and execution outcome. Hence, the process mining problem and the proposed algorithms can be extended for discovering temporal patterns at the activity property level. From the theoretical viewpoint, the proposed techniques can be generalized and extended beyond the temporal context by considering other types of relationships (e.g., spatial or structural relationships). Such extension can be applied to a broader spectrum of application domains. From the practical viewpoint, applying the proposed techniques to support of workflow evolutions and exceptions, and other business decisions represents interesting and desirable directions for future research.

Acknowledgements

This work was supported by National Science Council of the Republic of China under the grant NSC-89-2416-H-110-037 and NSC-90-2213-E-110-022.

References

Aalst, W., A.Weijters and L. Maruster, 2002, Workflow Mining: Which Processes can be Rediscovered? BETA Working Paper Series, WP 74, (Eindhoven University of Technology, Eindhoven).

Aggarwal, C.C. and P.S. Yu, 2001, Mining Associations with the Collective Strength Approach, IEEE Transactions on Knowledge and Data Engineering, Vol. 13, No. 6, pp.863-873.

Agrawal, R., T. Imielinski and A. Swarmi, 1993, Mining Association Rules Between Sets of Items in Large Databases, Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington DC.

Agrawal, R. and R. Srikant, 1994, Fast Algorithms for Mining Association Rules, Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chile.

Agrawal, R. and R. Srikant, 1995, Mining Sequential Patterns, Proceedings of International Conference on Data Engineering, Taipei, Taiwan.

Agrawal, R., D. Gunopulos and F. Leymann, 1998, Mining Process Models from Workflow Logs, Proceedings of the 6th International Conference on Extending Database Technology (EDBT), Valencia, Spain.

Alonso, G., M. Kamath, D. Agrawal, A. El Abbadi, R. Guenthoer and C. Mohan, 1994, Failure Handling in Large Scale Workflow Management Systems, IBM Research Report RJ9913, (IBM Almaden Research Center).

Anderberg, M.R., 1973, Cluster Analysis for Applications, (Academic Press Inc.).

Bettini, C., X.S. Wang, S. Jajodia and J.L. Lin, 1998, Discovering Frequent Event Patterns with Multiple Granularities in Time Sequences, IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 2, pp.222-237.

Breiman, L., J. Friedman, R. Olshen and C. Stone, 1984, Classification and Regression Trees, (Wadsworth, Pacific Grove).

Casati, F., S. Ceri, S. Paraboschi, and G. Pozzi, 1999, Specification and Implementation of Exceptions in Workflow Management Systems, ACM Transactions on Database Systems, Vol. 24, No. 3, pp.405-451.

Chiu, D., Q. Li and K. Karlapalem, 1999, A Meta Modeling Approach to Workflow Management Systems Supporting Exception Handling, Information Systems, Vol. 24, No. 2, pp.159-184.

Chiu, D., Q. Li and K. Karlapalem, 2001, Web Interface-Driven Cooperative Exception Handling in ADOME Workflow Management System, Information Systems, Vol. 26, No. 2, pp.93-120.

Cook, D. and L.B. Holder, 2000, Graph-based Data Mining, IEEE Intelligent Systems, Vol. 15, No. 2, pp.32-41.

Cormen, T., C.E. Leiserson and R.L. Rivest, 1989, Introduction to Algorithms, (MIT Press) 485-488.

Datta, A., 1998, Automating the Discovery of AS-IS Business Process Models: Probabilistic and Algorithmic Approaches, Information Systems Research, Vol. 9, No. 3, pp.275-301.

Davenport, T., 1993, Process Innovation—Reengineering Work through Information Technology, (Harvard Business School, Boston).

Eder, J., and W. Liebhart, 1998, Contributions to Exception Handling in Workflow Systems, Proceedings of Extending Database Technology (EDBT) Workshop on Workflow Management Systems, Valencia, Spain.

Gonzalez, J., I. Jonyer, L. B. Holder and D. J. Cook, 2000, Efficient Mining of Graph-based Data, Proceedings of International Conference on Artificial Intelligence.

Hammer, M. and J. Champy, 1993, Reengineering the Corporation: A Manifesto for Business Revolution, (Harper Business Press, New York).

Hwang, S.-Y. and W.-S. Yang, 2002, On the Discovery of Process Models from Their Instances, To appear in Decision Support Systems, Vol. 34, No. 1, pp. 41-57.

Kaufman, L. and P.J. Rousseeuw, 1990, Finding Groups in Data: An Introduction to Cluster Analysis, (John Wiley & Sons, Inc., New York).

Keim, D.A. and H. Kriegel, 1996, Visualization Techniques for Mining Large Databases: A Comparison, IEEE Transactions on Knowledge and Data Engineering, Vol. 8, No. 6, pp.923-927.

Kohonen, T., 1989, Self-organization and Associative Memory, (Springer).

Kohonen, T., 1995, Self-organizing Maps, (Springer).

Krishnakumar, N. and A.P. Sheth, 1995, Managing Heterogeneous Multi-system Tasks to Support Enterprise-wide Operations, Distributed and Parallel Databases, Vol. 3, No. 2, pp.155-186.

Lesh, N., M.J. Zaki and M. Ogihara, 2000, Scalable Feature Mining for Sequential Data, IEEE Intelligent Systems, Vol. 15, No. 2, pp.48-56.

Lin, F.R., S.C. Chou, S.M. Pan and Y.M. Chen, 2001, Mining Time Dependency Patterns in Clinical Pathways, International Journal of Medical Informatics, Vol. 62, No. 1, pp.11-25.

Mannila, H., H. Toivonen and A. I. Verkamo, 1995, Discovering Frequent Episodes in sequences, Proceedings of the First International Conference on Knowledge Discovery and Data Mining.

Mannila, H. and H. Toivonen, 1996, Discovering Generalized Episodes Using Minimal Occurrences, Proceedings of the Second International Conference on Knowledge Discovery and Data Mining.

Ng, R., and J. Han, 1994, Efficient and Effective Clustering Methods for Spatial Data Mining, Proceedings of International Conference on Very Large Data Bases, Santiago, Chile.

Quigley, P.A., S.W. Smith and J. Strugar, 1998, Successful Experiences with Clinical Pathways in Rehabilitation, Journal of Rehabilitation.

Quinlan, J.R., 1986, Induction of Decision Trees, Machine Learning, Vol. 1, No. 1, pp.81-106.

Quinlan, J.R., 1993, C4.5: Programs for Machine Learning, (Morgan Kaufmann, San Mateo).

Reichert, M. and P. Dadam, 1998, ADEPT—Supporting Dynamic Changes of Workflows without Losing Control, Journal of Intelligent Information System, Vol. 10, No. 2, pp.93-129.

Rissanen, J., 1983, A universal prior for integers and estimation by minimum description length, Annals of Statistics, Vol. 11, pp.416-431.

Rumelhart, D.E. and G.E. Hinton and R.J. Williams, 1986, Learning Internal Representations by Error Propagation, in: R.J. Williams, Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Vol. 1, (MIT Press, Cambridge) 318-362.

Sheth, A. and K. J. Kochut, 1997, Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems, Advances in Workflow Management Systems and Interoperability, (Istanbul, Turkey).

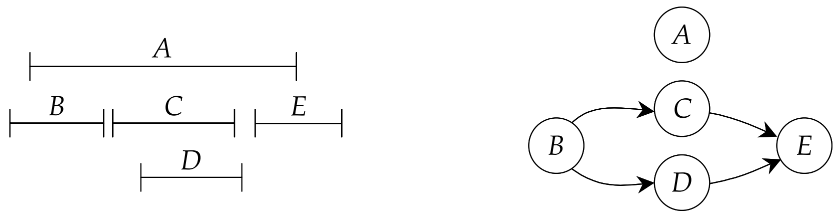
Srikant, R. and R. Agrawal, 1996, Mining Sequential Patterns: Generalizations and Performance Improvements, Proceedings of the 5th International Conference on

Extending Database Technology (EDBT), Avignon, France.

Srikant, R. and R. Agrawal, 1997, Mining Generalized Association Rules, Future Generation Computer Systems, Vol. 13, No. 2-3, pp.161-180.

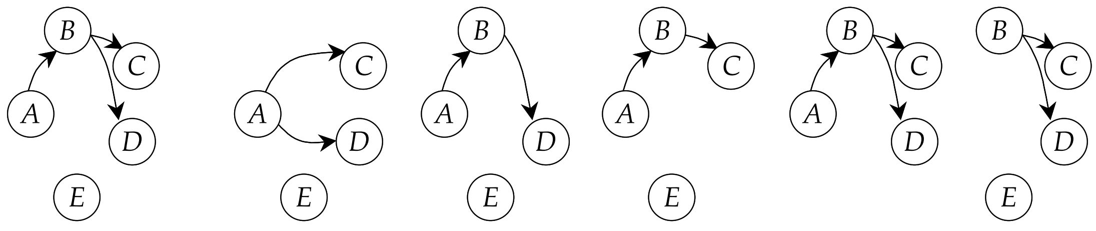
Tang, J. and S.Y. Hwang, 1996, Handling Uncertainties in Workflow Applications, Proceedings of International Conference on Information and Knowledge Management (CIKM'96).

WMC, 1994, The Workflow Reference Model, (Workflow Management Coalition).



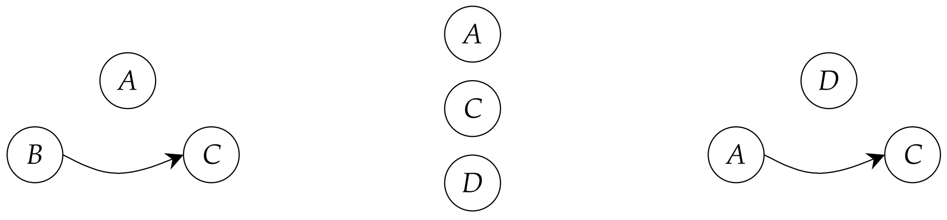
(a) Process Instance (b) Temporal Graph for the Process Instance in (a)

Figure 1. Example of Process Instance and Corresponding Temporal Graph



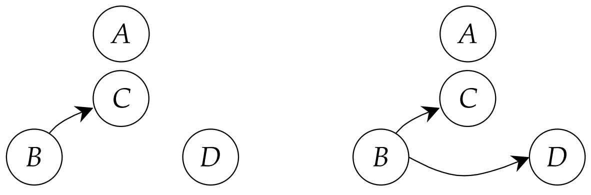
(a) Temporal Graph G (b) G–{B} (c) G–{C} (d) G–{D} (e) G–{E} (f) G–{A}

Figure 2. Examples of Subtraction Operation



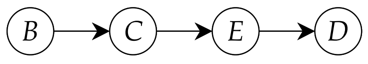
(a) Temporal Graph G_1 (b) Temporal Graph G_2 (c) Temporal Graph G_3

Figure 3. Examples of Temporal Graphs of Size 3

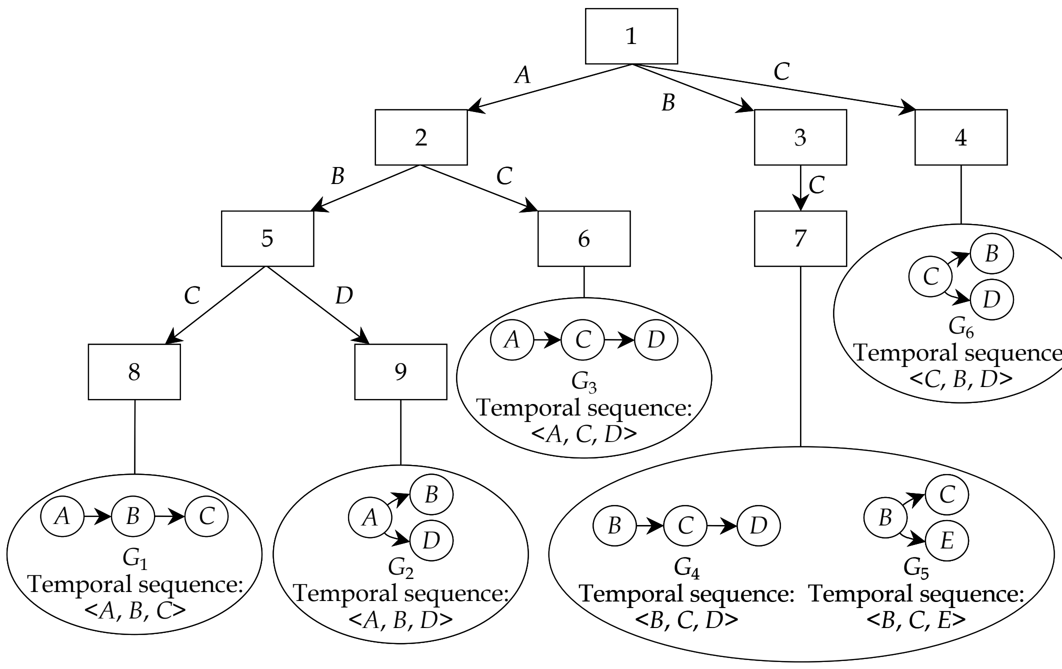


(a) (b)

Figure 4. Two Candidate Temporal Graphs Resulting from Joining G_1 and G_2 in Figure 3

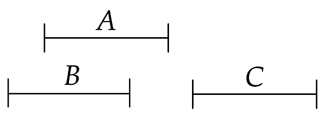


(a) Process Instance

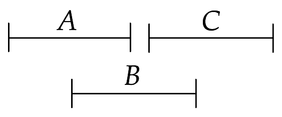


(b) A Segment of Hash-tree

Figure 5. Hash-tree for Candidate Temporal Graphs of Size 3

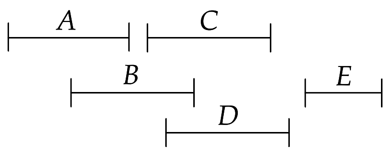


(a) Process Instance 1



(b) Process Instance 2

Figure 6: Examples of Three Process Instances

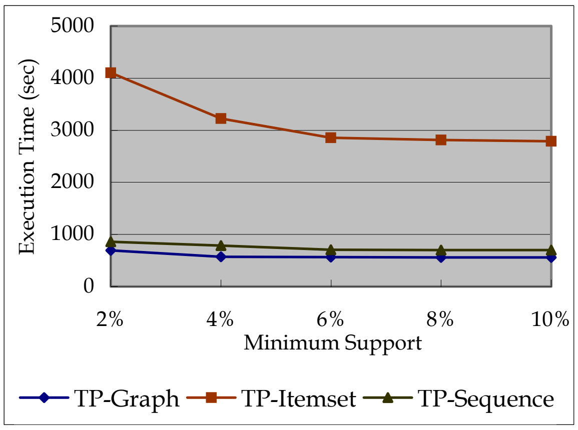


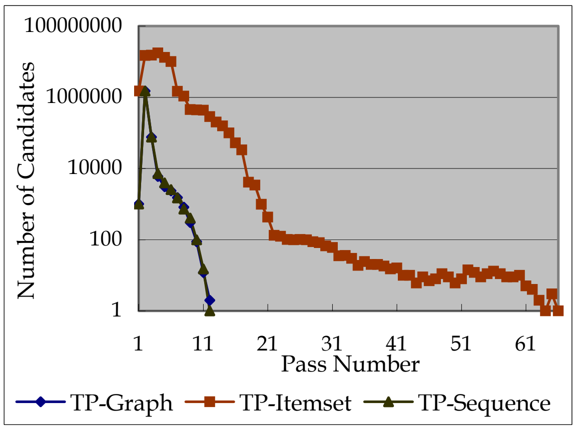
(a) Process Instance

<{A, B}{B, C, D}{E}>

(b) Respective Quasi-sequence

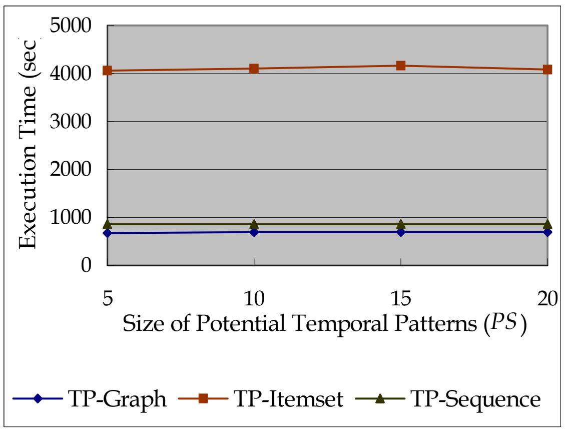
Figure 7: Examples of Process Instance and Quasi-sequence

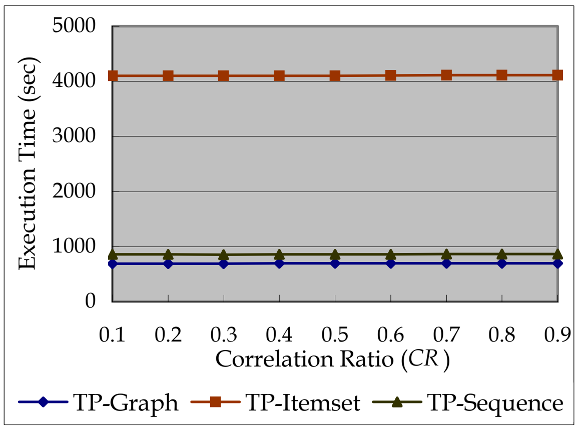




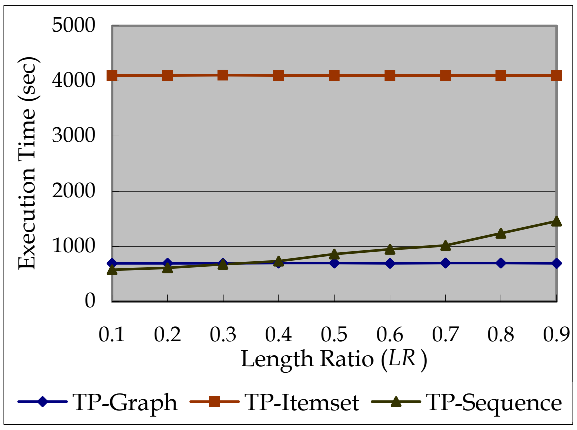
(a) Effects of Minimum Supports on Execution Times (b) Size of Candidates (Minimum Support = 2%)

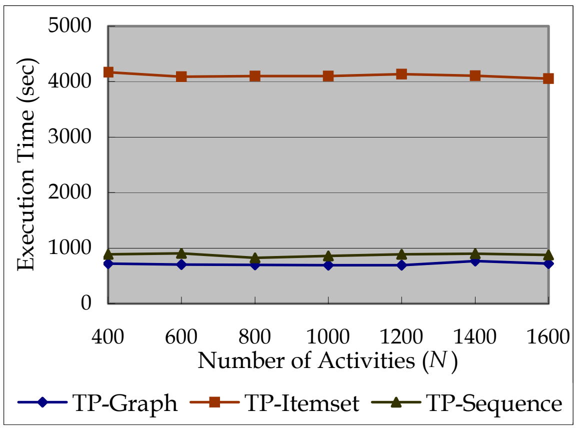
Figure 8. Experimental Results: Effects of Minimum Support Thresholds





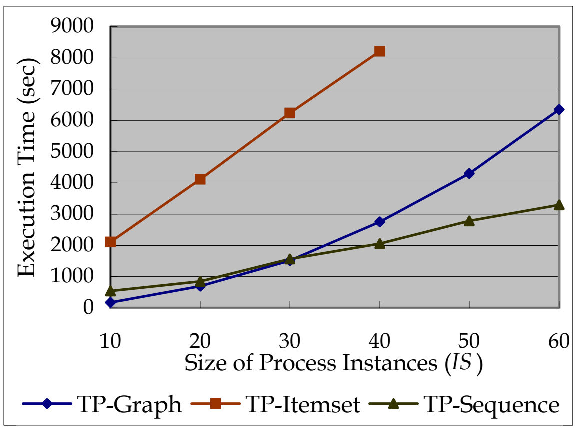
(a) Effects of Sizes of Potential Temporal Patterns (b) Effects of Correlation Ratios



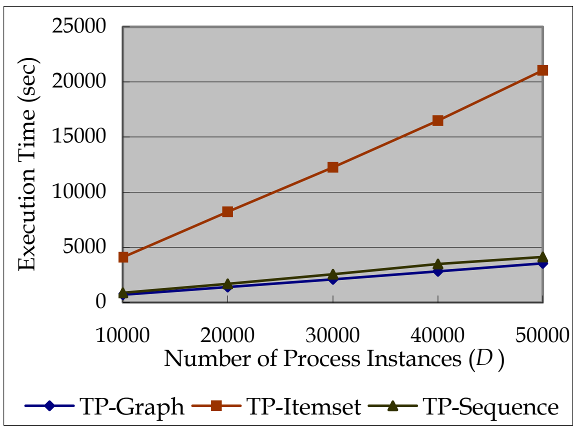


(c) Effects of Length Ratios (d) Effects of Numbers of Activities

Figure 9. Experimental Results: Effects of Process Characteristics



(a) Effects of Sizes of Process Instances (*IS*)



(b) Effects of Numbers of Process Instances (*D*)

Figure 10. Results of Scale-up Experiments

Table1. Parameters and Default Values for Synthetic Data Generation

Symbol	**Description**	**Default**
N	Number of activities	1,000
D	Number of process instances	10,000
IS	Size of process instances	20
PN	Number of potential temporal patterns	1,000
PS	Size of potential temporal patterns	10
CR	Correlation ratio	0.5
LR	Length ratio	0.5