# Discovering Lag Intervals for Temporal Dependencies

Liang Tang Tao Li
School of Computer Science
Florida International University
11200 S.W. 8th Street
Miami, Florida, 33199
U.S.A
{ltang002,taoli}@cs.fiu.edu

Larisa Shwartz
IBM T.J Watson Research Center
19 Skyline Drive
Hawthorne, NY, 10532
U.S.A
lshwart@us.ibm.com

## ABSTRACT

Time lag is a key feature of hidden temporal dependencies within sequential data. In many real-world applications, time lag plays an essential role in interpreting the cause of discovered temporal dependencies. Traditional temporal mining methods either use a predefined time window to analyze the item sequence, or employ statistical techniques to simply derive the time dependencies among items. Such paradigms cannot effectively handle varied data with special properties, e.g., the interleaved temporal dependencies.

In this paper, we study the problem of finding lag intervals for temporal dependency analysis. We first investigate the correlations between the temporal dependencies and other temporal patterns, and then propose a generalized framework to resolve the problem. By utilizing the sorted table in representing time lags among items, the proposed algorithm achieves an elegant balance between the time cost and the space cost. Extensive empirical evaluation on both synthetic and real data sets demonstrates the efficiency and effectiveness of our proposed algorithm in finding the temporal dependencies with lag intervals in sequential data.

## Categories and Subject Descriptors

H.2.8 [**Database applications**]: Data mining

## Keywords

Temporal dependency, Time lag

## 1. INTRODUCTION

Sequential data is prevalent in business, system management, health-care and many scientific domains. One fundamental problem in temporal data mining is to discover hidden temporal dependencies in the sequential data [23] [11] [13] [8] [29]. In temporal data mining, the input data is typically a sequence of discrete items associated with time stamps [24] [23]. Let $A$ and $B$ be two types of items, a temporal dependency for $A$ and $B$, written as $A \rightarrow B$, denotes that the occurrence of $B$ depends on the occurrence of $A$. The dependency indicates that an item $A$ is often followed by an item $B$. Let $[t_1, t_2]$ be the range of the lag for two dependent $A$ and $B$. This temporal dependency with $[t_1, t_2]$ is written as $A \rightarrow_{[t_1,t_2]} B$ [7]. For example, in system management, disk capacity alert and database alert are two item types. When the disk capacity is full, the database engine often raises a database alert in the next 5 to 6 minutes as shown in Figure 1. Hence, the disk capacity has a temporal dependency with the database. $[5\mathtt{min}, 6\mathtt{min}]$ is the lag interval between the two dependent system alerts. $disk\_capacity\_alert \rightarrow_{[5\mathtt{min},6\mathtt{min}]} database\_alert$ describes the temporal dependency with the associated lag interval. This paper studies the problem of finding appropriate lag intervals for two dependent item types.

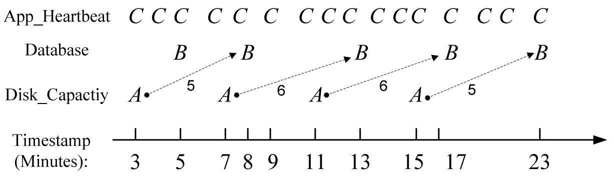

**Figure 1: Lag Interval for Temporal Dependency**

Temporal dependencies are often used for prediction. In Figure 1, $[5\mathtt{min}, 6\mathtt{min}]$ is the predicted time range, indicating when a database alert occurs after a disk capacity alert is received. Furthermore, the associated lag interval characterizes the cause of a temporal dependency. For example, if the database is writing a huge temporal log file which is larger than the disk free space, the database alert is immediately raised in $[0\mathtt{min}, 1\mathtt{min}]$. But if the disk free capacity is consumed by other applications, the database engine can only detect this alert when it runs queries. The associate time lags in such a case would be larger than 1 minute.

Previous work for discovering temporal dependencies does not consider interleaved dependencies [17] [4] [21]. For $A \rightarrow B$, they assume that an item $A$ can only have a dependency with its first following $B$. However, it is possible that an item $A$ has a dependency with any following $B$. For example, in Figure 1, the time lag for two dependent $A$ and $B$ is 5 to 6 minutes, but the time lag for two adjacent $A$'s is only 4 minutes. All $A$'s have a dependency with the second following $B$, not the first following $B$. Hence, the dependencies among these dependent $A$ and $B$ are interleaved. For two item types, the numbers of time stamps are both $O(n)$, The number of possible time lags is $O(n^2)$. Thus, the number of lag intervals is $O(n^4)$. The challenge of our work is how

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*KDD'12,* August 12–16, 2012, Beijing, China.
Copyright 2012 ACM 978-1-4503-1462-6 /12/08 ...$15.00.

to efficiently find appropriate lag intervals over the $O(n^4)$ candidates.

## 1.1 Contributions

In this paper, we study the problem of finding appropriate lag intervals for temporal dependency analysis. The contribution of this paper is summarized as follows:

- Investigates the relationship among the lag intervals and other existing temporal patterns proposed in previous work. It shows that, many existing temporal patterns can be expressed as special cases of temporal dependencies with lag intervals.
- Develops an algorithm for discovering appropriate lag intervals. The time complexity is $O(n^2 \log n)$ and the space complexity is $O(N)$, where $N$ is the number of items, and $n$ is the number of distinct time stamps. This paper also proves that, there is no algorithm can solve this problem with an $o(n^2)$ time complexity.
- Conducts extensive experiments on synthetic and real data sets. The experimental results confirm the theoretical analysis of this paper and show that the performance of the proposed algorithm outperforms baseline algorithms.

## 1.2 Road Map

The rest of the paper is organized as follows: Section 2 summarizes the related work for temporal pattern mining and discusses the relationships with other existing temporal patterns. Section 3 defines the lag interval that we try to find. Section 4 presents several algorithms for finding appropriate lag intervals and analyzes the complexity of our problem. In Section 5, we present the experimental studies on synthetic and real data sets. Finally, Section 6 concludes our paper and discusses the future work.

# 2. RELATED WORK

Previous work of temporal dependency discovery can be categorized by the data set type. The first category is for market basket data, which is a collection of transactions [29] where each transaction is a sequence of items. The purpose of this type of temporal dependency discovery is to find frequent subsequences which are contained by a certain amount of transactions. Typical algorithms are GSP [28], FreeSpan[9], PrefixSpan[26], and SPAM[3]. The second category is for the time series data. A temporal dependency of this category is seen as a correlation on multiple time series variables [32] [5], which determines whether one time series is useful in forecasting another. Our work belongs to the third category, which is for temporal symbolic sequences. The input data is an item sequence and each item is associated with a time stamp. An item may represent an event or a behavior in history [18] [24] [23] [25][16]. The purpose is to find various temporal relationships among these events or behaviors. Many temporal patterns proposed in previous work can be considered as special cases of temporal dependencies with different lag intervals.

## 2.1 Relation with Other Temporal Patterns

Table 1 lists several types of temporal patterns proposed in the literature and their corresponding temporal dependencies with lag intervals. A mutually dependent pattern (**m-pattern**) $\{A, B\}$, can be described as two temporal dependencies $A \rightarrow_{[0,\delta]} B$ and $B \rightarrow_{[0,\delta]} A$. Items of $A$ and $B$ in an **m-pattern** appear almost together so that $t_1 = 0, t_2 \leq \delta$, where $\delta$ is the time tolerance. A partially periodic pattern (**p-pattern**) [20] for a single item $A$, can be expressed as a temporal dependency $A \rightarrow_{[p-\delta,p+\delta]} A$, where $p$ is the period. Frequent episodes $A \rightarrow B \rightarrow C$ can be separated to $A \rightarrow_{[0,p]} B$ and $B \rightarrow_{[0,p]} C$ where $p$ is the parameter of the time window length [21]. [17] proposes *loose temporal pattern* and *stringent temporal pattern*. As shown in Table 1, the two types of temporal patterns can be explained by two temporal dependencies with particular constraints on the lag intervals. One common problem of these algorithms is how to set the precise parameter about the time window [21] [20] [4]. For example, for discovering partially periodic patterns, if $\delta$ is too small, the identification of partially periodic patterns would be too strict and no result can be found; if the $\delta$ is too large, many false results would be found. [15] [14] [22] directly find frequent episodes according to the occurrences of episodes in the data sequence. The discovered frequent episode may not have fixed lag intervals for the represented temporal dependency. Our method proposed in this paper does not require users to specify the parameters about the time window and is able to discover interleaved temporal dependencies.

# 3. QUALIFIED LAG INTERVAL

Given an item sequence $S = x_1 x_2 ... x_N$, $x_i$ denotes the type of the $i$-th item, and $t(x_i)$ denotes the time stamp of $x_i$, $i = 1, 2, ..., N$. Intuitively, if there is a temporal dependency $A \rightarrow_{[t_1,t_2]} B$ in $S$, there must be a lot of $A$'s that are followed by some $B$ with a time lag in $[t_1, t_2]$. Let $n_{[t_1,t_2]}$ denote the observed number of $A$'s in this situation. For instance, in Figure 1, every $A$ is followed by a $B$ with a time lag of 5 or 6 minutes, so $n_{[5,6]} = 4$. Only the second $A$ is followed by a $B$ with a time lag of 0 or 1 minute, so $n_{[0,1]} = 1$. Let $r = [t_1, t_2]$ be a lag interval. One question is that, what is the minimum required $n_r$ that we can utilize to identify the dependency of $A$ and $B$ with $r$. In this example, the minimum required $n_r$ cannot be greater than 4 since the sequence has at most 4 $A$'s. However, if let $r = [0, +\infty]$, we can easily have $n_r = 4$. [20] proposes a chi-square test approach to determine the minimum required $n_r$, where the chi-square statistic measures the degree of the independence by comparing the observed $n_r$ with the expected $n_r$ under the independent assumption. The null distribution of the statistic is approximated by the chi-squared distribution with 1 degree of freedom. Let $\chi^2_r$ denote the chi-square statistic for $n_r$. A high $\chi^2_r$ indicates the observed $n_r$ in the given sequence cannot be explained by randomness. The chi-square statistic is defined as follows:

$$\chi^2_r = \frac{(n_r - n_A P_r)^2}{n_A P_r (1 - P_r)}, \qquad (1)$$

where $n_A$ is the number of $A$'s in the data sequence, $P_r$ is the probability of a $B$ appearing in $r$ from a random sequence. Hence, $n_A P_r$ is the expected number of $A$'s that are followed by some $B$ with a time lag in $r$. $n_A P_r (1 - P_r)$ is the variance. Note that the random sequence should have the same sampling rate for $B$ as the given sequence $S$. The randomness is only for the positions of $B$ items. It is known that a random sequence usually follows the Poisson process, which assumes the probability of an item appearing in an interval

**Table 1: Relation with Other Temporal Patterns**

| **Temporal Pattern** | **An Example** | **Equivalent Temporal Dependency with Lag Interval** |
|---|---|---|
| Mutually dependent pattern [19] | $\{A, B\}$ | $A \rightarrow_{[0,\delta]} B$, $B \rightarrow_{[0,\delta]} A$ |
| Partially periodic pattern [20] | $A$ with periodic $p$ and a given time tolerance $\delta$ | $A \rightarrow_{[p-\delta,p+\delta]} A$ |
| Frequent episode pattern [21] | $A \rightarrow B \rightarrow C$ with a given time window $p$ | $A \rightarrow_{[0,p]} B$, $B \rightarrow_{[0,p]} C$ |
| Loose temporal pattern [17] | $B$ follows by $A$ before time $t$ | $A \rightarrow_{[0,t]} B$ |
| Stringent temporal pattern [17] | $B$ follows by $A$ about time $t$ with a given time tolerance $\delta$ | $A \rightarrow_{[t-\delta,t+\delta]} B$ |

is proportional to the length of the interval [27]. Therefore,

$$P_r = |r| \cdot \frac{n_B}{T}, \tag{2}$$

where $|r|$ is the length of $r$, $|r| = t_2 - t_1 + w_B$, $w_B$ is the minimum time lag of two adjacent $B$'s, $w_B > 0$, and $n_B$ is the number of $B$'s in $S$. For lag interval $r$, the absolute length is $t_2-t_1$. $w_B$ is added to $|r|$ because without $w_B$ when $t_1 = t_2$, $|r| = 0$, $P_r$ is always 0 no matter how large the $n_B$ is. As a result, $\chi^2_r$ would be overestimated. In reality, the time stamps of items are discrete samples and $w_B$ is the observed sampling period for $B$ items. Hence, the probability of a $B$ appearing in $t_2 - t_1$ time units is equal to the probability of a $B$ appearing in $t_2 - t_1 + w_B$ time units.

The value of $\chi^2_r$ is defined in terms of a confidence level. For example, 95% confidence level corresponds to $\chi^2_r = 3.84$. Based on Eq.(1), the observed $n_r$ should be greater than $\sqrt{3.84 n_A P_r(1 - P_r)} + n_A P_r$. Note that we only care positive dependencies, so

$$n_r - n_A P_r > 0. \tag{3}$$

To ensure a discovered temporal dependency fits the entire data sequence, *support* [2] [28] [20] is used in our work. For $A \rightarrow_r B$, the support $supp_A(r)$ (or $supp_B(r)$) is the number of $A$'s (or $B$'s) that satisfy $A \rightarrow_r B$ divided by the total number of items $N$. *minsup* is the minimum threshold for both $supp_A(r)$ and $supp_B(r)$ specified by the user [28] [20]. Based on the two minimum thresholds $\chi^2_c$ and *minsup*, Definition 1 defines the qualified lag interval that we try to find.

DEFINITION 1. *Given an item sequence $S$ with two item types $A$ and $B$, a lag interval $r = [t_1, t_2]$ is qualified if and only if $\chi^2_r > \chi^2_c$, $supp_A(r) > minsup$ and $supp_B(r) > minsup$, where $\chi^2_c$ and minsup are two minimum thresholds specified by the user.*

# 4. LAG INTERVAL DISCOVERY

In this section, we first develop a straightforward algorithm for finding all qualified lag intervals, a *brute-force* algorithm. Then, *STScan* and *STScan*$^*$ algorithms are proposed which are much more efficient. We also present a lower bound of the time complexity for finding qualified lag intervals. Finally, we discuss how to incorporate the domain knowledge to speed up the algorithms.

## 4.1 Brute-Force Algorithm

To find all qualified lag intervals, a straightforward algorithm is to enumerate all possible lag intervals, compute their $\chi^2_r$ and supports, and then check whether they are qualified or not. This algorithm is called *brute-force*. Clearly, its time cost is very large. Let $n$ be the number of distinct time stamps of $S$, $r = [t_1, t_2]$. The numbers of possible $t_1$ and $t_2$ are $O(n^2)$, and then the number of possible $r$ is $O(n^4)$. For each lag interval, there is at least $O(n)$ cost to scan the entire sequence $S$ to compute the $\chi^2_r$ and the supports. Therefore, the overall time cost of the *brute-force* algorithm is $O(n^5)$, which is not affordable for large data sequences.

## 4.2 STScan Algorithm

To avoid re-scanning the data sequence, we develop a sorted table based algorithm. A sorted table is a sorted linked list with a collection of sorted integer arrays. Each entry of the linked list is attached to two sorted integer arrays. Figure 2 shows an example of the sorted array. In our algorithm, we store every time lag $t(x_j) - t(x_i)$ into each entry of linked list, where $x_i = A$, $x_j = B$, $i, j$ are integers from 1 to $N$. Two arrays attached to the entry $t(x_j) - t(x_i)$ are the collections of $i$ and $j$. In other words, the two arrays are the indices of $A$'s and $B$'s. Let $E_i$ denote the $i$-th entry of the linked list and $v(E_i)$ denote the time lag stored at $E_i$. $IA_i$ and $IB_i$ denote the indices of $A$'s and $B$'s that are attached to $E_i$. For example in Figure 2, $x_3 = A$, $x_5 = B$, $t(x_5) - t(x_3) = 20$. Since $v(E_2) = 20$, $IA_2$ contains 3 and $IB_2$ contains 5. Any feasible lag interval can be represented as a subsegment of the linked list. For example in Figure 2, $E_2E_3E_4$ represents the lag interval $[20, 120]$.

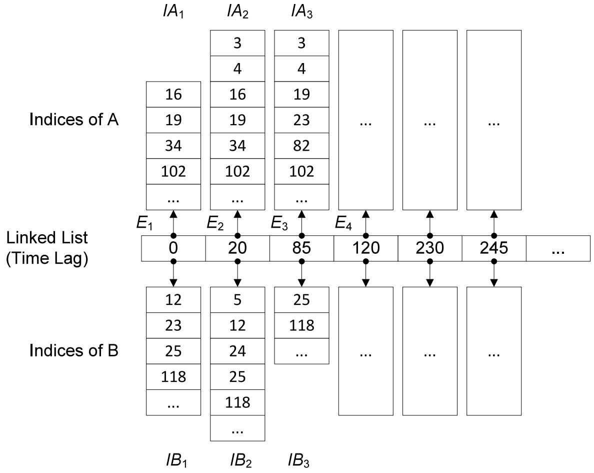

**Figure 2: Sorted Table**

To create the sorted table for a sequence $S$, each time lag between an $A$ and a $B$ is first inserted into a red-black tree. The key of the red-black tree node is the time lag, the value is the pair of indices of $A$ and $B$. Once the tree is built, we traverse the tree in ascending order to create the linked list of the sorted table. In the sequence $S$, the number $A$ and $B$ are both $O(N)$, so the number of $t(x_j) - t(x_i)$ is $O(N^2)$. The time cost of creating the red-black tree is $O(N^2 \log N^2) = O(N^2 \log N)$. Traversing the tree costs $O(N^2)$. Hence, the overall time cost of creating a sorted table is $O(N^2 \log N)$, which is the known lower bound of sorting $X + Y$ where $X$ and $Y$ are two variables [10]. The linked list has $O(N^2)$ entries, and each attached integer array has $O(N)$ elements, so it seems that the space cost of a sorted table is $O(N^2 \cdot N) = O(N^3)$. However, Lemma 1 shows that the actual space cost of a sorted table is $O(N^2)$, which is same as the red-black tree.

LEMMA 1. *Given an item sequence $S$ having $N$ items, the space cost of its sorted table is $O(N^2)$.*

PROOF. Since the numbers of $A$'s and $B$'s are both $O(N)$, the number of pairs $(x_i, x_j)$ is $O(N^2)$, where $x_i = A$, $x_j = B$, $x_i, x_j \in S$. Every pair associated with three entries in the sorted table: the time stamp distance, the index of an $A$ and the index of a $B$. Therefore, each pair $(x_i, x_j)$ introduces 3 space cost. The total space cost of the sorted table is $O(3N^2) = O(N^2)$. □

Once the sorted table is created, finding all qualified lag intervals is scanning the subsegments of the linked list. However, the number of entries in the linked list is $O(N^2)$, so there are $O(N^4)$ distinct subsegments. Scanning all subsegments is still time-consuming. Fortunately, based on the minimum thresholds on the chi-square statistic and the support, the length of a qualified lag interval cannot be large.

LEMMA 2. *Given two minimum thresholds $\chi^2_c$ and minsup, the length of any qualified lag interval is less than $\frac{T}{N} \cdot \frac{1}{minsup}$.*

PROOF. Let $r$ be a qualified lag interval. Based Eq.(1) and Inequality.(3), $\chi^2_r$ increases along with $n_r$. Since $n_r \leq n_A$,

$$\frac{(n_A - n_A P_r)^2}{n_A P_r (1 - P_r)} \geq \chi^2_r > \chi^2_c \quad \Longrightarrow \quad P_r < \frac{n_A}{\chi^2_c + n_A}.$$

By substituting Eq. 2 to the previous inequality,

$$|r| < \frac{n_A}{\chi^2_c + n_A} \cdot \frac{T}{n_B}.$$

Since $n_B > N \cdot minsup$, $\chi^2_c > 0$, we have

$$|r| < \frac{T}{N} \cdot \frac{1}{minsup} = |r|_{max}.$$

□

$\frac{T}{N}$ is exactly the average period of items, which is determined by the sampling rate of this sequence. For example, in system event sequences, the monitoring system checks the system status for every 30 seconds and records system events into the sequence. The average period of items is 30 seconds. Therefore, we consider $\frac{T}{N}$ as a constant. $minsup$ is also a constant, so $|r|_{max}$ is a constant.

Algorithm *STScan* states the pseudocode for finding all qualified lag intervals. $len(ST)$ denotes the number of entries of the linked list in sorted table $ST$. This algorithm sequentially scans all subsegments starting with $E_1$, $E_2$, ..., $E_{len(ST)}$. Based on Lemma 2, it only scans the subsegment with $|r| < |r|_{max}$. To calculate the $\chi^2_r$ and the supports, for each subsegment, it cumulatively stores the aggregate indices of $A$'s and $B$'s and the corresponding lag interval $r$. For each subsegment, $n_r = |IA_r|$, $supp_A(r) = |IA_r|/N$, $supp_B(r) = |IB_r|/N$.

**Algorithm 1** *STScan* $(S, A, B, ST, \chi^2_c, minsup)$

```
Input: S : input sequence; A, B: two item types; ST : sorted table; $\chi^2_c$: minimum chi-square statistic threshold; minsup: minimum support.
Output: a set of qualified lag intervals;
$R \leftarrow \emptyset$
Scan $S$ to find $w_B$
for $i = 1$ to $len(ST)$ do
  $IA_r \leftarrow \emptyset$, $IB_r \leftarrow \emptyset$
  $t_1 \leftarrow v(E_i)$
  $j \leftarrow 0$
  while $i + j \leq len(ST)$ do
    $t_2 \leftarrow v(E_{i+j})$
    $r \leftarrow [t_1, t_2]$
    $|r| \leftarrow t_2 - t_1 + w_B$
    if $|r| \geq |r|_{max}$ then
      break
    end if
    $IA_r \leftarrow IA_r \cup IA_{i+j}$
    $IB_r \leftarrow IB_r \cup IB_{i+j}$
    $j \leftarrow j + 1$
    if $|IA_r|/N \leq minsup$ or $|IB_r|/N \leq minsup$ then
      continue
    end if
    Calculate $\chi^2_r$ from $|IA_r|$ and $|r|$
    if $\chi^2_r > \chi^2_c$ then
      $R \leftarrow R \cup r$
    end if
  end while
end for
return $R$
```

LEMMA 3. *The time cost of* STScan *is $O(N^2)$, where $N$ is the number of items in the data sequence.*

PROOF. For each entry $E_{i+j}$ in the linked list, the time cost of merging $IA_{i+j}$ and $IB_{i+j}$ to $IA_r$ and $IB_r$ is $|IA_{i+j}| + |IB_{i+j}|$ by using a hash table. Let $l_i$ be the largest length of the scanned subsegments starting at $E_i$. Let $l_{max}$ be the maximum $l_i$, $i = 1, ..., len(ST)$. The total time cost is:

$$\begin{aligned} T(N) &= \sum_{i=1}^{len(ST)} \sum_{j=0}^{l_i - 1} (|IA_{i+j}| + |IB_{i+j}|) \\ &\leq \sum_{i=1}^{len(ST)} \sum_{j=0}^{l_{max}-1} (|IA_{i+j}| + |IB_{i+j}|) \\ &\leq l_{max} \cdot \sum_{i=1}^{len(ST)} (|IA_i| + |IB_i|) \end{aligned}$$

$\sum_{i=1}^{len(ST)} (|IA_i| + |IB_i|)$ is exactly the total number of integers in all integer arrays. Based on Lemma 1, $\sum_{i=1}^{len(ST)} (|IA_i| + |IB_i|) = O(N^2)$. Then $T(N) = O(l_{max} \cdot N^2)$. Let $E_k ... E_{k+l}$ be the subsegment for a qualified lag interval, $v(E_{k+i}) \geq 0$, $i = 0, ..., l$. The length of this lag interval is $|r| = v(E_{k+l_{max}}) - v(E_k) < |r|_{max}$, then $l_{max} < |r|_{max}$ and $l_{max}$ is not depending on $N$. Assume $\Delta_E$ is the average $v(E_{k+1}) - v(E_k)$, $k = 1, ..., len(ST) - 1$, we obtain a tighter bound of $l_{max}$, i.e., $l_{max} \leq |r|_{max}/\Delta_E \leq \frac{T}{N \cdot \Delta_E} \cdot \frac{1}{minsup}$. Therefore, the overall time cost is $T(N) = O(N^2)$. □

## 4.3 STScan* Algorithm

To reduce the space cost of *STScan* algorithm, we develop an improved algorithm *STScan*$^*$ which utilizes the increment sorted table and sequence compression.

### 4.3.1 Incremental Sorted Table

Lemma 1 shows the space cost of a complete sorted table is $O(N^2)$. Algorithm *STScan* sequentially scans the subsegments starting from $E_1$ to $E_{len(ST)}$, so it does not need to access every entry at every time. Based on this observation, we develop an incremental sorted table based algorithm with an $O(N)$ space cost. This algorithm incrementally creates the entries of the sorted table along with the subsegment scanning process.

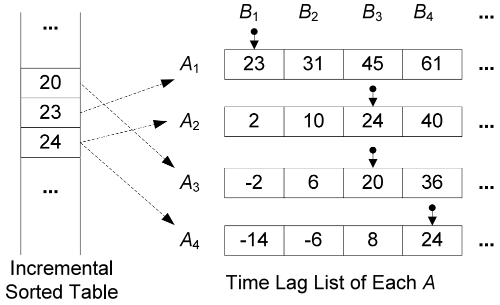

**Figure 3: Incremental Sorted Table**

The linked list of a sorted table can be created by merging all time lag lists of $A$'s ( Figure 3), where $A_i$ and $B_j$ denote the $i$-th $A$ and the $j$-th $B$, $i, j = 1, 2, ....$ The $j$-th entry in the list of $A_i$ stores $t(B_j)-t(A_i)$. The time lag lists of all $A$'s are not necessary to be created in the memory because we only need to know $t(B_j)$ and $t(A_j)$. This can be done just with an indices arrays of all $A$'s and all $B$'s respectively. By using $N$-way merging algorithm, each entry of the linked list would be created sequentially. The indices of $A$'s and $B$'s attached to each entry are also recorded during the merging process. Base on Lemma 2, the length of a qualified lag interval is at most $|r|_{max}$, therefore, we only keep track of the recent $l_{max}$ entries. The space cost for storing $l_{max}$ entries is at most $O(l_{max} \cdot N) = O(N)$. A heap used by the merging process costs $O(N)$ space. Then, the overall space cost of the incremental sorted table is $O(N)$. The time cost of merging $O(N)$ lists with total $O(N^2)$ elements is still $O(N^2 \log N)$.

### 4.3.2 Sequence Compression

In many real-world applications, some items may share the same time stamp since they are sampled within the same sampling cycle. To save the time cost, we compress the original $S$ to another compact sequence $S'$. At each time stamp $t$ in $S$, if there are $k$ items of type $I$, we create a triple $(I, t, k)$ into $S'$, where $k$ is the cardinality of this triple. To handle $S'$, the only needed change of our algorithm is that the $|IA_r|$ and $|IB_r|$ become the total cardinalities of triples in $IA_r$ and $IB_r$ respectively. Clearly, $S'$ is more compact than $S$. $S'$ has $O(n)$ triples, where $n$ is the number of distinct time stamps of $S$, $n \leq N$. Creating $S'$ costs an $O(N)$ time complexity. By using $S'$, the time cost of *STScan*$^*$ becomes $O(N + n^2 \log n)$ and the space cost of the incremental sorted table becomes $O(n)$.

## 4.4 Time Complexity Lower Bound

For analyzing large sequences, an $O(n)$ or $O(n \log n)$ algorithm is needed. However, we find that the time complexity of any algorithm for our problem is at least $O(n^2)$ (Lemma 4). The proof is to reduce the 3SUM$'$ problem to our problem, and the 3SUM$'$ has no $o(n^2)$ solution [6]. To answer whether $O(n^2)$ is the tightest lower bound or not, a further study is needed.

Lemma 4. *Finding a qualified lag interval cannot be solved in $o(n^2)$ in the worst case, where $n$ is the number of distinct time stamps of the given sequence.*

Proof. Assume that an algorithm $\mathbf{P}$ can find a qualified lag interval in $o(n^2)$ in any case, we can construct an algorithm to solve the 3SUM$'$ problem in $o(n^2)$ as follows. Given three sets of integers $X$, $Y$, and $Z$ such that $|X| + |Y| + |Z| = n$, we construct a compressed sequence $S'$ of items which only has two item types $A$ and $B$ as follows:

1. For each $x_i$ in $X$, create an $A$ at time stamp $x_i$.

2. For each $y_i$ in $Y$, create a $B$ at time stamp $y_i$.

3. For each $z_i$ in $Z$, create $n+1$ $A$'s at time stamp $\beta(i+1)+z_i$ and $n+1$ $B$'s at time stamp $\beta(i+1)$, where $\beta$ is the diameter of set $X \cup Y$, which is the largest integer minus the smallest integer in $X \cup Y$.

Only the lag intervals created from $z_i$ have $n_r \geq n+1$. If there are three integers $y_j \in Y$, $x_k \in X$, $z_i \in Z$ such that $y_j - x_k = z_i$, the lag interval of $z_i$ must have $n_r \geq n+2$. Then, we substitute $n_r = n+2$ into Eq. 1 to find the appropriate threshold $\chi^2_c$, and call algorithm $\mathbf{P}$ to find all $z_i$ that have $n_r \geq n+2$. By filtering out the situations of $y_j - y_k = z_i$ and $x_j - x_k = z_i$, we can obtain the desired three integers such that $y_j - x_k = z_i$ if they exist. $S'$ has at most $2n$ distinct time stamps. The time cost of creating $S'$ is $O(2n) = O(n)$. $\mathbf{P}$ is an $o(n^2)$ algorithm. Filtering the result of $\mathbf{P}$ is $O(n)$ since $|Z| \leq n$. Therefore, the overall solution for the 3SUM$'$ problem is $O(n) + o(n^2) + O(n) = o(n^2)$. However, it is believed that the 3SUM$'$ problem has no $o(n^2)$ solution [6]. Therefore, the algorithm $\mathbf{P}$ does not exist. □

## 4.5 Finding Lag Interval with Constraints

As discussed in Section 2, most existing temporal patterns can be described as temporal dependencies with some constraints on the lag interval. In many real-world applications, the users have the domain knowledge or other requirements for desired lag intervals, which are helpful to speed up the algorithm. For example, in system management, the temporal dependencies of certain events can be used to identify false alarms [31]. However, SLA(Service Level Agreement) requires that any real system alarms must be acknowledged within $K$ hours, and $K$ is specified in the contract with customers. If a discovered lag interval is greater than $K$ hours, the corresponding temporal dependency is trivial and can not be used for false alarm identification.

Generally, the constraints for a qualified lag interval $[t_1, t_2]$ can be expressed as a set of inequalities:

$$f_i(t_1, t_2) \leq d_i, \text{for } i = 1, ..., m,$$

where $m$ is the number of constraints, $f_i(t_1, t_2) \leq d_i$ are constraint functions that need to be satisfied. For example, the constraint for the partially periodic pattern is $|t_1 - t_2| \leq \delta$.

The constraints for the predefined time window based temporal pattern are $t_1 = 0$, $t_2 \leq p$, where $p$ is the window length. To incorporate the constraints to our algorithm, a straightforward approach is to filter discovered qualified lag interval by the given constraints. However, this approach does not make use of the constraints to reduce the search space of the problem. On the other hand, since the constraint function $f_i(\cdot, \cdot)$ can be any complex function about $t_1$ and $t_2$, there is no generalized and optimal approach for using them. We only consider two typical cases: $|t_1 - t_2| \leq \delta$ and $t_2 \leq p$. The first case can be utilized in the subsegment scanning. It provides a potential tighter bound than $l_{max}$ if $\delta < l_{max}$, but does not change the order of the overall time cost. The second case can be utilized to reduce the length of the linked list of the sorted table. When $t_2 < p$, each time lag list of $A_i$ has at most $O(p/\Delta_E)$ entries. Then, the overall time cost can be reduced to $O(n \log n \cdot p/\Delta_E) = O(n \log n)$.

# 5. EVALUATION

This section presents our empirical study of discovering lag intervals on both synthetic data sets and real data sets in terms of the effectiveness and efficiency.

## 5.1 Experimental Platform and Algorithms

All comparative algorithms are implemented in Java 1.6 platform. Table 2 summarizes our experimental environment. At present, the most dedicated algorithm for finding lag intervals is the inter-arrival clustering method [17] [20], denoted by *inter-arrival*. For $A \to B$, an inter-arrival is the time lag of an $A$ to its first following $B$. A dense cluster created from all inter-arrivals indicates its time lag frequently appears in the sequence. Thus, a qualified lag interval is probably around this time lag. This algorithm is very efficient and only has a linear time cost, however, it does not consider the interleaved dependencies. We also implement the four algorithms, *brute-force*, *brute-force*$^*$, *STScan* and *STScan*$^*$, to compare with in this experiment. *brute-force*$^*$ is the improved version of *brute-force* which utilizes the pruning strategy about $|r|_{max}$ mentioned in Lemma 2.

**Table 2: Experimental Machine**

| OS | CPU | bits | Memory | JVM Heap Size |
|---|---|---|---|---|
| Linux 2.6.18 | Intel Xeon(R) @ 2.5GHz, 8 core | 64 | 16G | 12G |

For each test, we enumerate all pairwise temporal dependencies for discovering qualified lag intervals.

## 5.2 Synthetic Data

The synthetic data consists of 7 data sequences. Each sequence is first generated from a random item sequence with 8 item types, denoted by $I_1$,...,$I_8$. The average sample period of items is 100. Three predefined temporal dependencies are randomly embedded into each random sequence and shown in Table 3. For each temporal dependency $I_i \to_{[t_1,t_2]} I_j$, we first randomly choose an item $x_i$ and an integer $t \in [t_1, t_2]$, and then let $x_i = I_i$ and the item at $t(x_i) + t$ be $I_j$. We repeat this process until $\chi^2_{[t_1,t_2]}$ and the support are greater than the specified thresholds. Note that the time lags in these lag intervals are larger than the average sample period of items, so all three temporal dependencies are very likely to be interleaved dependencies.

**Table 3: Embedded Temporal Dependencies**

| Embedded Temporal Dependencies | Support |
|---|---|
| $I_1 \to_{[400,500]} I_2$ | 0.1 |
| $I_2 \to_{[1000,1100]} I_3$ | 0.12 |
| $I_4 \to_{[5500,5800]} I_5$ | 0.15 |

### 5.2.1 Effectiveness

The effectiveness of the algorithm result is validated by comparing the discovered results with the embedded lag intervals and measured by the recall [29]. We do not care the precision because every algorithm can achieve the 100% precision if this algorithm is correct. We let $\chi^2_c = 10.83$ which represents a 99.9% confidence level, $minsup = 0.1$. There is no surprise that all the algorithms proposed in this paper, *brute-force*, *brute-force*$^*$, *STScan* and *STScan*$^*$, find all the embedded lag intervals since they scan the entire space of the lag interval. Thus, the recalls of these methods are 1.0. The parameter $\delta$ of *inter-arrival* is varied from 1 to 2000. However, *inter-arrival* does not find any qualified lag interval in the synthetic data and its recall is 0. The reason is that, the qualified lag intervals are [400,500], [1000,1100] and [5500,5800], but most inter-arrivals in the sequence are close to the average sample period 100. Thus, *inter-arrival* can only probe the lag intervals around 100.

### 5.2.2 Efficiency

The empirical efficiency is evaluated by the CPU running time (Figure 4). *inter-arrival* is a linear algorithm, so it runs much faster than other algorithms. The running time of the *brute-force* algorithm increases extremely fast so that it can only handle very tiny data sets. By adding the pruning strategy about $|r|_{max}$ to *brute-force*, the *brute-force*$^*$ algorithm runs a little bit faster than the *brute-force* algorithm, but it still can only handle small data sets. *STScan*$^*$ compresses the sequence before the lag interval discovering, therefore, *STScan*$^*$ is a little bit more efficient than *STScan*.

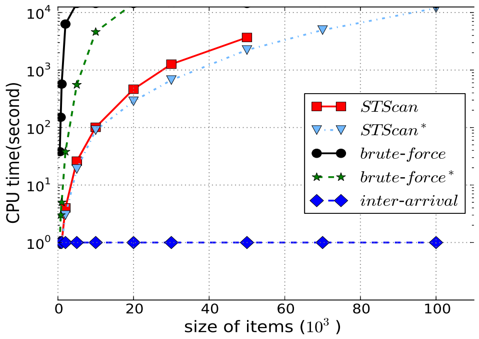

**Figure 4: Runtime on Synthetic Data**

*STScan* has not finish the tests for larger data sets because it runs out of memory. Table 5 lists the approximate peak numbers of allocated objects in Java heap memory (not including the data sequence). It confirms Lemma 1 that the sorted table takes an $O(N^2)$ space cost. It also shows that, the space costs of *STScan*$^*$, *brute-force* and *brute-force*$^*$ are all $O(N)$ as mentioned in Section 4. Assuming each Java object only occupies an integer(8 bytes), *STScan* would cost

**Table 4: Discovered Temporal Dependencies with Lag Intervals**

| Data set | Dependency | $\chi^2_r$ | *support* |
|---|---|---|---|
| Account1 | $MSG\_Plat\_APP \rightarrow_{[3600,3600]} MSG\_Plat\_APP$ | $>$ 1000.0 | 0.07 |
| | $Linux\_Process \rightarrow_{[0,96]} Process$ | 134.56 | 0.05 |
| | $SMP\_CPU \rightarrow_{[0,27]} Linux\_Process$ | 978.87 | 0.06 |
| | $AS\_MSG \rightarrow_{[102,102]} AS\_MSG$ | $>$ 1000.0 | 0.08 |
| Account2 | $TEC\_Error \rightarrow_{[0,1]} Ticket\_Retry$ | $>$ 1000.0 | 0.12 |
| | $Ticket\_Retry \rightarrow_{[0,1]} TEC\_Error$ | $>$ 1000.0 | 0.12 |
| | $AIX\_HW\_ERROR \rightarrow_{[25,25]} AIX\_HW\_ERROR$ | 282.53 | 0.15 |
| | $AIX\_HW\_ERROR \rightarrow_{[8,9]} AIX\_HW\_ERROR$ | 144.62 | 0.24 |

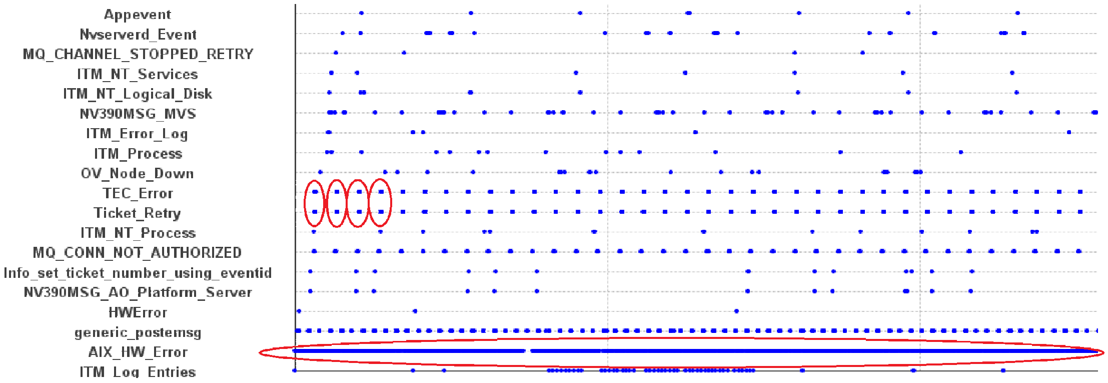

**Figure 5: Plotting for Account2 Data**

**Table 5: Space Cost on Synthetic Data**

| Data size<br>Algorithm | $10^3$ | $10 \times 10^3$ | $50 \times 10^3$ | $100 \times 10^3$ |
|---|---|---|---|---|
| *STScan* | $3 \times 10^4$ | $3 \times 10^6$ | $8 \times 10^7$ | OutOfMemory |
| *STScan** | $10^3$ | $10^4$ | $5 \times 10^4$ | $10^5$ |
| *brute-force* | $9 \times 10^2$ | $10^4$ | $5 \times 10^4$ | $9 \times 10^4$ |
| *brute-force** | $9 \times 10^2$ | $10^4$ | $5 \times 10^4$ | $9 \times 10^4$ |
| *inter-arrival* | $< 10^2$ | $< 10^2$ | $< 10^2$ | $< 10^2$ |

over 10G bytes memory for $50 \times 10^3$ items. Hence, it runs out of memory when the data size becomes larger. However, by using the incremental sorted table, for the same data set, *STScan** only costs 10M memory. *inter-arrival* only stores the clusters of all inter-arrivals, so its space cost is small.

## 5.3 Real Data

**Table 6: Real System Events**

| Data set | Time Frame | #Events | #Event Types |
|---|---|---|---|
| Account1 | 54 days | 1,124,834 | 95 |
| Account2 | 32 days | 2,076,408 | 104 |

Two real data sets are collected from IT outsourcing centers by IBM Tivoli monitoring system [1] [30], denoted as Account1 and Account2. Each data set is a collection of system events from hundreds of application servers and data server. These system events are mostly system alerts triggered by some monitoring situations (e.g. the CPU utilization is above a threshold). Table 6 shows the time frames and the sizes of the two real data sets. To discover the temporal dependencies with qualified lag intervals, we let $\chi^2_c = 6.64$ which corresponds to the confidence level 99%, and $minsup = 0.05$. A constraint that $t_2 \leq$ 1hour is applied to this testing from the domain experts. $\delta$ of *inter-arrival* is varied from 1 to 2000.

Figures 6 and 7 show the running times of all algorithms on the two real data sets. As for *STScan* and *STScan** , the running times grow slower than in Figure 4 because the constraint $t_2 \leq$ 1hour reduces their time complexities. Table 7 lists the peak numbers of allocated memory objects in JVM on Account2 data. The results on Account1 data is similar to this table.

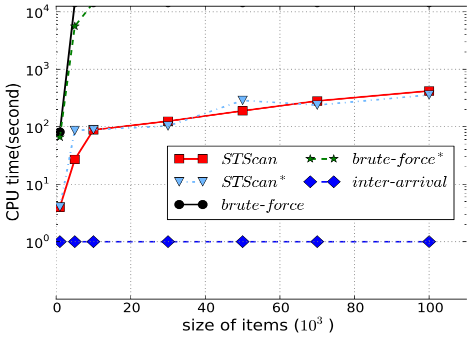

**Figure 6: Running Time on Account1 Data**

**Table 7: Space Cost on Account2 Data**

| Data size<br>Algorithm | $10^3$ | $10 \times 10^3$ | $50 \times 10^3$ | $100 \times 10^3$ |
|---|---|---|---|---|
| *STScan* | $4 \times 10^4$ | $3 \times 10^6$ | $1 \times 10^7$ | $3 \times 10^7$ |
| *STScan** | $10^3$ | $6 \times 10^3$ | $5 \times 10^4$ | $10^5$ |
| *brute-force* | $9 \times 10^2$ | $3 \times 10^3$ | $3 \times 10^3$ | $3 \times 10^3$ |
| *brute-force** | $9 \times 10^2$ | $3 \times 10^3$ | $3 \times 10^3$ | $3 \times 10^3$ |
| *inter-arrival* | $< 10^2$ | $< 10^2$ | $< 10^2$ | $< 10^2$ |

Table 4 lists several discovered temporal dependencies with qualified lag intervals. *inter-arrival* only finds the first two temporal dependencies on Account2 data. The reason is

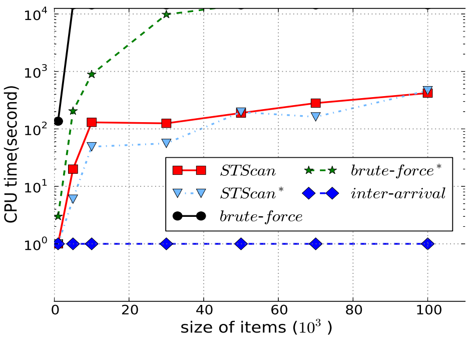

**Figure 7: Running Time on Account2 Data**

that, only the two temporal dependencies have very small lag intervals which are just the inter-arrivals of the events. However, the lag intervals for other temporal dependencies are larger than most inter-arrivals, so *inter-arrival* fails.

In Table 4, the first discovered temporal dependency for Account1 shows that *MSG_Plat_APP* is a periodic pattern with a period of 1 hour. This pattern indicates this event *MSG_Plat_APP* is a heartbeat signal from an application. The second and third discovered temporal dependencies can be viewed as a case study for event correlation [12]. Since most servers are Linux servers, so the alerts from processes must be also from Linux processes. Therefore, for Account1, process events and Linux process events can be automatically correlated. High CPU utilization alerts (*SMP_CPU*) can only be triggered by abnormal processes, so *SMP_CPU* events can also be correlated with *Linux_Process* events. In Account2, the first two temporal dependencies compose a mutual dependency pattern between *TEC_Error* and *Ticket_Retry*. It can be explained by a programming logic in IBM Tivoli monitoring system. When the monitoring system fails to generate the incident ticket to the ticketing system, it will report a *TEC* error and retry the ticket generation. Therefore, *TEC_Error* and *Ticket_Retry* events are often raised together. The third and fourth discovered temporal dependencies for Account2 are related to a hardware error of an AIX server but with different lag intervals. This is caused by a polling monitoring situation. When an AIX server is down, the monitoring system continuously receive *AIX_HW_Error* events when polling that AIX server. Thus, this *AIX_HW_Error* event exhibits a periodic pattern. To validate the discovered results, we plot the temporal events into a graphical chart. Figure 5 is a screen shot of the plotting for Account2 data. The x-axis is the time stamp, the y-axis is the event type. As shown by this figure, *TEC_Error* and *Ticket_Retry* exhibit a mutually dependency since they are always generated at the almost same time. *AIX_HW_Error* is a polling event.

### 5.3.1 Parameter Sensitivity

To test the sensitivity of parameters, we vary $\chi_c^2$ and *minsup* and test the numbers of discovered temporal dependencies (Figures 8 and Figure 9) and the running time (Figure 10 and Figure 11). When varying $\chi_c^2$, $minsup = 0.05$; When varying *minsup*, $\chi_c^2 = 6.64$ (with 99% confidence level). $\chi_c^2$ is not sensitive to the algorithm result because the associated confidence level is only from 95% to 99.99% although $\chi_c^2$ is varied from 3.84 to 100. By varying *minsup*, the number of discovered temporal dependencies exponentially decreases as shown in Figure 9. As mentioned in [20], the effective choice of *minsup* is 0.001 to 0.1.

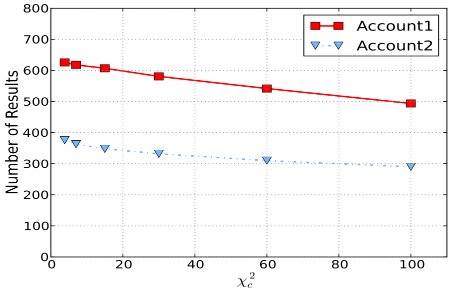

**Figure 8: Num. of Results by Varying $\chi_c^2$**

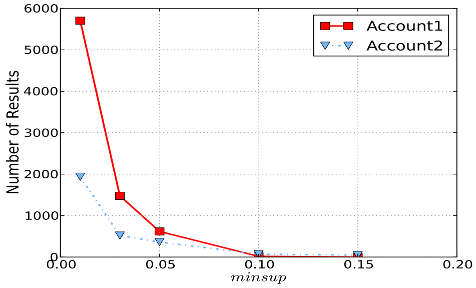

**Figure 9: Num. of Results by Varying *minsup***

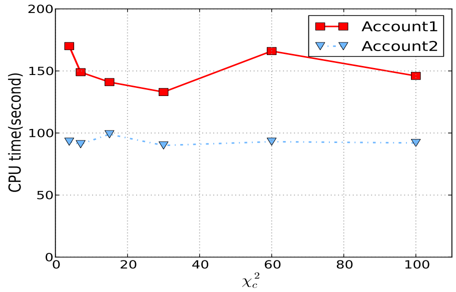

**Figure 10: Running time by Varying $\chi_c^2$**

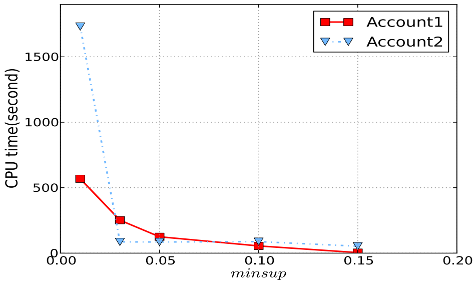

**Figure 11: Running time by Varying *minsup***

## 6. CONCLUSION AND FUTURE WORK

In this paper, we study the problem of finding appropriate lag intervals for temporal dependencies over item sequences. We investigate the relationship between the temporal dependency with other existing temporal patterns. Two algorithms *STScan* and *STScan*$^*$ are proposed to efficiently discover the appropriate lag intervals. Extensive empirical evaluation on both synthetic and real data sets demonstrates the efficiency and effectiveness of our proposed algorithm in finding the temporal dependencies with lag intervals in sequential data. As for the future work, we will continue to investigate more efficient algorithms that can handle large data sequences. We hope to find an $O(n^2)$ time complexity algorithm with a linear or constant space cost.

## Acknowledgement

The work is supported in part by NSF grants IIS-0546280 and HRD-0833093.

## 7. REFERENCES

[1] IBM Tivoli Monitoring. `http://www-01.ibm.com/software/tivoli/products/monitor/`.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proccedings of VLDB*, pages 487–499, 1994.

[3] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of KDD*, pages 429–435, 2002.

[4] K. Bouandas and A. Osmani. Mining association rules in temporal sequences. In *Proceedings of CIDM*, pages 610–615, 2007.

[5] A. Dhurandhar. Learning maximum lag for grouped graphical granger models. In *ICDM Workshops*, pages 217–224, 2010.

[6] A. Gajentaan and M. H. Overmars. On a class of $O(n^2)$ problems in computational geometry. *Computational Geometry*, 5:165–185, 1995.

[7] L. Golab, H. J. Karloff, F. Korn, A. Saha, and D. Srivastava. Sequential dependencies. *PVLDB*, 2(1):574–585, 2009.

[8] R. Gwadera, M. J. Atallah, and W. Szpankowski. Reliable detection of episodes in event sequences. In *Proccedings of ICDM*, pages 67–74, 2003.

[9] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M. Hsu. Freespan: frequent pattern-projected sequential pattern mining. In *Proccedings of KDD*, pages 355–359, 2000.

[10] A. Hernandez-Barrera. Finding an $o(n^2 \log n)$ algorithm is sometimes hard. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, pages 289–294, August 1996.

[11] E. J. Keogh, S. Lonardi, and B. Y. chi Chiu. Finding surprising patterns in a time series database in linear time and space. In *Proccedings of KDD*, pages 550–556, 2002.

[12] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. J. Stolfo. A coding approach to event correlation. In *Integrated Network Management*, pages 266–277, 1995.

[13] S. Laxman and P. S. SASTRY. A survey of temporal data mining. *Sadhana*, 31(2):173–198, 2006.

[14] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan. Discovering frequent episodes and learning hidden markov models: A formal connection. *IEEE Trans. Knowl. Data Eng.*, 17(11):1505–1517, 2005.

[15] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan. A fast algorithm for finding frequent episodes in event streams. In *Proceedings of ACM KDD*, pages 410–419, August 2007.

[16] T. Li, F. Liang, S. Ma, and W. Peng. An integrated framework on mining logs files for computing system management. In *Proceedings of ACM KDD*, pages 776–781, August 2005.

[17] T. Li and S. Ma. Mining temporal patterns without predefined time windows. In *Proceedings of ICDM*, pages 451–454, November 2004.

[18] Z. Li, B. Ding, J. Han, R. Kays, and P. Nye. Mining periodic behaviors for moving objects. In *Proccedings of KDD*, pages 1099–1108, 2010.

[19] S. Ma and J. L. Hellerstein. Mining mutually dependent patterns. In *Proceedings of ICDE*, pages 409–416, 2001.

[20] S. Ma and J. L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proceedings of ICDE*, pages 205–214, 2001.

[21] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[22] N. Méger and C. Rigotti. Constraint-based mining of episode rules and optimal window sizes. In *Proccedings of PKDD*, pages 313–324, 2004.

[23] T. Mitsa. *Temporal Data Mining.* Chapman and Hall/CRC, 2010.

[24] F. Mörchen. Algorithms for time series knowledge mining. In *Proccedings of KDD*, pages 668–673, 2006.

[25] F. Mörchen and D. Fradkin. Robust mining of time intervals with semi-interval partial order patterns. In *Proccedings of SDM*, pages 315–326, 2010.

[26] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of EDBT*, pages 215–224, 2001.

[27] S. M. Ross. *Stochastic Processes.* Wiley, 1995.

[28] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of EDBT*, pages 3–17, 1996.

[29] P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining.* Addison Wesley, 2005.

[30] L. Tang, T. Li, F. Pinel, L. Shwartz, and G. Grabarnik. Optimizing system monitoring configurations for non-actionable alerts. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium*, 2012.

[31] W. Xu, L. Huang, A. Fox, D. A. Patterson, and M. I. Jordan. Mining console logs for large-scale system problem detection. In *SysML*, December 2008.

[32] W.-X. Zhou and D. Sornette. Non-parametric determination of real-time lag structure between two time series: The 'optimal thermal causal path' method with applications to economic data. *Journal of Macroeconomics*, 28(1):195 – 224, 2006.