# Discovering Frequent Arrangements of Temporal Intervals

Panagiotis Papapetrou, George Kollios, Stan Sclaroff
Department of Computer Science
Boston University
{panagpap,gkollios,sclaroff}@cs.bu.edu

Dimitrios Gunopulos
Department of Computer Science
University of California, Riverside
dg@cs.ucr.edu

## Abstract

*In this paper we study a new problem in temporal pattern mining: discovering frequent arrangements of temporal intervals. We assume that the database consists of sequences of events, where an event occurs during a* time-interval. *The goal is to mine arrangements of event intervals that appear frequently in the database. There are many applications where these type of patterns can be useful, including data network, scientific, and financial applications. Efficient methods to find frequent arrangements of temporal intervals using both breadth first and depth first search techniques are described. The performance of the proposed algorithms is evaluated and compared with other approaches on real datasets (American Sign Language streams and network data) and large synthetic datasets.*

## 1 Introduction

Sequential pattern mining has received particular attention in the last decade [2, 3, 5, 6, 10, 24, 9, 18, 11, 19, 23, 13, 12, 21]. Despite advances in this area, nearly all proposed algorithms concentrate on the case where events occur at single time instants. However, in many applications events are not instantaneous; they instead occur over a time interval. Furthermore, since different temporal events may occur concurrently, it would be useful to extract frequent temporal patterns of these events. In this paper the goal is to develop methods that discover temporal arrangements of correlated event intervals which occur frequently in a database.

There are many applications that require mining such temporal relations. Consider an ASL (American Sign Language) database that contains useful linguistic information on a variety of grammatical and syntactic structures, as well as manual and gestural fields [15]. Detecting relations between the above structures and fields could be interesting to the linguists and may help them discover new types of relations they were unaware of. Another application is in network monitoring, where the goal is to analyze packet and router logs. Multiple types of events occurring over certain time periods can be stored in a log, and the goal is to detect general temporal relations of these events that with high probability would describe regular patterns in the network. This could then be used for prediction and intrusion detection.

Existing sequential pattern mining methods are hampered by the fact that they can only handle instantaneous events, not event intervals. Nonetheless, such algorithms could be retrofitted for the purpose, via converting a database of event intervals to a transactional database, by considering only the start and end points of every event interval. An existing sequential pattern mining algorithm could be applied to the converted database, and the extracted patterns could be post-processed to produce the desired set of frequent arrangements. However, an arrangement of $k$ intervals corresponds to a sequence of length $2k$. Hence, this approach will produce up to $2^{2k}$ different sequential patterns. Moreover, post-processing will also be costly, since the extracted patterns consist of event start and end points, and for each event interval all the relations with the other event intervals must be determined.

To the best of our knowledge, there have been no efficient methods developed that consider general types of temporal arrangements between event intervals. Our main contributions include: (a) a formal definition for the problem of mining frequent temporal arrangements of intervals in an interval database, (b) two efficient algorithms for mining frequent arrangements of temporally correlated events using breadth first and depth first techniques in an enumeration tree of temporal arrangements, and (c) an extensive experimental evaluation of these techniques and a comparison with a standard sequential pattern mining method using both real and synthetic datasets.

The remainder of this paper is organized as follows: Sec. 2 provides the problem formulation along with the appropriate background and an overview of the existing methods related to our work. Sec. 3 presents two tree-based ap-

This work was supported in part by NSF grants IIS-0308213, IIS-0329009, and IIS-0330481, and by ONR grant N00014-03-1-0108.

proaches for mining frequent arrangements of temporally correlated events. Sec. 4 describes experimental evaluation, and Sec. 5 gives conclusions.

## 2 Background and related work

In this section we give the problem formulation along with the appropriate background. Moreover, we provide a review of the most related work.

### 2.1 Event Interval Temporal Relations

In this paper we consider five types of temporal relations between two event intervals. Using these relations we define more general temporal arrangements. However, our methods are not limited to these relations and can be easily extended to include more types of temporal relations, as the the ones described in [4].

Consider two event-intervals $A$ and $B$. Furthermore, assume that the user specifies an $\epsilon$ that is used to define more flexible matchings between two time instants. The following relations are studied (see also Fig. 1):

- **Meet($A, B$):** In this case, $B$ follows $A$, with $B$ starting at the time $A$ terminates, i.e. $t_e(A) = t_s(B) \pm \epsilon$. This case is denoted as $AB$ and we say that $A$ $meets$ $B$.

- **Match($A, B$):** In this case, $A$ and $B$ are parallel, beginning and ending at the same time, i.e. $t_s(A) = t_s(B) \pm \epsilon$ and $t_e(A) = t_e(B) \pm \epsilon$. This case is denoted as $A||B$ and we say that $A$ $matches$ with $B$.

- **Overlap($A, B$):** In this case, the start time of $B$ occurs after the start time of $A$, and $A$ terminates before $B$, i.e. $t_s(A) < t_s(B)$, $t_e(A) < t_e(B)$ and $t_s(B) < t_e(A)$. This case is denoted as $A|B$ and we say that $A$ $overlaps$ with $B$.

- **Contain($A, B$):** In this case, the start time of $B$ follows the start time of $A$ and the termination of $A$ occurs after the termination of $B$, i.e. $t_s(A) < t_s(B)\ and\ t_e(A) > t_e(B)$ and $A$ does not match with $B$. This case is denoted as $A > B$ and we say that $A$ $contains$ $B$.

- **Follow($A, B$):** In this case, B occurs after A terminates, i.e. $t_e(A) \pm \epsilon < t_s(B)$. This case is denoted as $A \rightarrow B$ and we say that $B$ $follows$ $A$.

### 2.2 Problem Formulation

Let $\mathcal{E}$ = $\{E_1, E_2, ..., E_m\}$ be an ordered set of event intervals, called *event interval sequence* or *e-sequence*. Each $E_i$ is a triple $(e_i,\ t^i_{start},\ t^i_{end})$, where $e_i$ is an event label, $t^i_{start}$ is the event start time and $t^i_{end}$ is the end time. The events are ordered by the start time. If an occurrence of $e_i$ is instantaneous, then $t^i_{start} = t^i_{end}$. An e-sequence of size k is called a *k-e-sequence*. For example, let us consider the 5-e-sequence shown in Fig. 2. In this case the e-sequence can be represented as follows: $\mathcal{E}$ = $\{(A, 1, 7),\ (B, 3, 19),\ (D, 4, 30),\ (C, 7, 15),\ (C, 23, 42)\}$. Finally, an *e-sequence database* $D$ = $\{\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_k\}$ is a set of e-sequences.

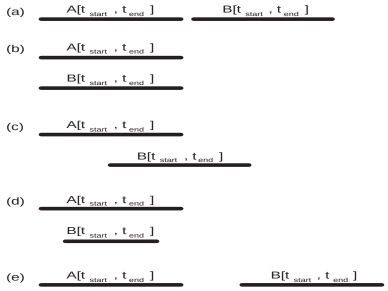


**Figure 1. Basic relations between two event-intervals: (a) Meet, (b) Match, (c) Overlap, (d) Contain, (e) Follow.**

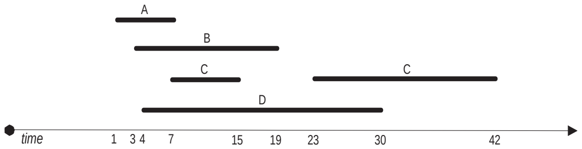


**Figure 2. An Example of an e-sequence.**

In an e-sequence database there may be patterns of temporally correlated events; such patterns are called *arrangements*. The definitions given in Section 2.1 can describe temporal relations between two event intervals but they are insufficient for relations between more than two. Consider for example the two cases in Fig. 3. Case $(a)$ can be easily expressed using the current notation as: $A|B \rightarrow C$. This is sufficient to determine that $A$ overlaps with $B$, $C$ follows $B$ and $C$ follows $A$. On the other hand, the expression for case $(b)$, i.e. $A|B > C$, is insufficient, since it gives no information about the relation between A and C. Thus, we need to add one more operand in order to express this relation concisely. In order to define an arrangement of more than two events we need to clearly specify the temporal relations between every pair of its events. This can be done by using the "AND" operand denoted by $*$. Therefore, the above example can be sufficiently expressed as follows: $A|B * A|C * B > C$. Based on the previous analysis, we can efficiently express any kind of relation between any number of event intervals, using the set of operands: $\mathcal{R} = \{|, ||, >, \rightarrow\}$ and *.

Consequently, an arrangement $\mathcal{A}$ of $n$ events is defined as $\mathcal{A}$ = $\{\mathcal{E},\ R\}$, where $\mathcal{E}$ is the set of event intervals that occur in $\mathcal{A}$, with $|\mathcal{E}|$ = $n$, and $R$ = $\{R\,(E_1,\ E_2),\ R\,(E_1,\ E_3),\ \dots,\ R\,(E_1,\ E_n),\ R\,(E_2,\ E_3),\ R\,(E_2,\ E_4),\ \dots\ ,\ R\,(E_2,\ E_n),\ R\,(E_{n-1},\ E_n)\}$. $R$ is the set or temporal relations between each pair $(E_i,\ E_j)$, for $i = 1,\ \dots,\ n$ and $j = i+1,\ \dots\ , n$, and $R\,(E_i,\ E_j) \in \mathcal{R}$ de-

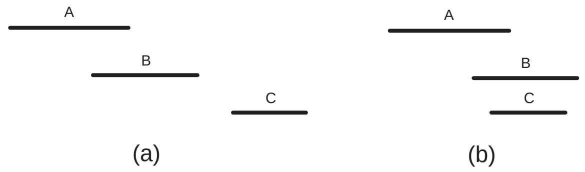


**Figure 3. (a) $S'$ can be expressed with four operands and (b) $S''$ cannot.**

fines the temporal relation between $E_i$ and $E_j$. The size of an arrangement $\mathcal{A} = \{\mathcal{E},\ R\}$ is equal to $|\mathcal{E}|$. An arrangement of size $k$ is called a *$k$-arrangement*. For example, consider arrangement $S'$ of size 3 shown in Fig. 3 (a). In this case $\mathcal{E} = \{A,\ B,\ C\}$ and $R = \{R\,(A,\ B) = |,\ R\,(A,\ C) = \rightarrow,\ R\,(B,\ C) = \rightarrow\}$. The *absolute support* of an arrangement in an e-sequence database is the number of e-sequences in the database that contain the arrangement. The *relative support* of an arrangement is the percentage of e-sequences in the database that contain the arrangement. Given an e-sequence $s$, $s$ *contains* an arrangement $\mathcal{A} = \{\mathcal{E},\ R\}$, if all the events in $\mathcal{A}$ also appear in $s$ with the same relations between them, as defined in $R$. Consider again arrangement $S'$ in Fig. 3(a) and e-sequence $s$ in Fig. 2. We can see that all the event intervals in $S'$ appear in $s$ and further, they are similarly correlated, i.e. Overlap ($A$,$B$), Follow ($B$,$C$), Follow ($A$,$C$). Thus, $S'$ is contained in or $supported$ by $s$. Given a minimum support threshold $min\_sup$, an arrangement is *frequent* in an e-sequence database, if it occurs in at least $min\_sup$ e-sequences in the database.

Based on the above definitions we can now formulate the problem of *mining frequent temporal arrangements* as follows: given an e-sequence database $D$ and a support threshold $min\_sup$, our task is to find set $F = \{\mathcal{A}_1, \mathcal{A}_2,\ ...,\mathcal{A}_n\}$, where $\mathcal{A}_i$ is a frequent arrangement in $D$.

### 2.3 Related Work

Discovering all frequent sequential patterns (or episodes) in large databases is a very challenging task since the search space is large. Consider for instance the case of a database with $m$ attributes. If we are interested in finding all the length $k$ *frequent sequences*, there are O($m^k$) potentially frequent ones. Increasing the number of objects might definitely lead to a paramount computational cost. The AprioriAll algorithm suggested in [3] employs a bottom-up search enumerating every single frequent sequence. This implies that in order to produce a frequent sequence of length $l$, all $2^l$ subsequences have to be generated, according to the apriori principle stated in [2]. It can be easily deduced that this exponential complexity is limiting all the apriori-based algorithms to discover only short patterns. According to [6], the candidate production can be done faster and more efficiently using a set-enumeration tree. Based on this, recent algorithms [9, 24, 13, 5, 20] have introduced more efficient techniques and data-structures in order to improve the pattern mining performance. Some of these algorithms resulted in two or more orders of magnitude in performance improvements over Apriori on some data-sets.

If small differences in the problem definition are ignored, the vast majority of the former algorithms extract frequent sequential patterns based on a support threshold. This threshold limits the results to the most common or “famous” ones among all the sequences in the database, causing a lack of *user-controlled focus* in the pattern mining process that results in an overwhelming volume of potentially useless patterns. A solution to this problem suggested in [10], was to introduce user-specified constraints, modeled by regular expressions. Sequential pattern mining algorithms developed so far, despite their outstanding performance in databases of short sequences, yield dramatically poor performance when the support threshold is low or when the databases consist of very long sequences. A similar problem occurs when mining frequent itemsets [12, 2]. In order to overcome this problem, a very interesting solution has been proposed in [16], where the mining process focuses only on *closed itemsets*. An itemset $I$ is *closed* if there is no superset of $I$ in the database with the same support. Consequently, there have been some efficient algorithms developed for mining frequent closed itemsets [17, 22, 25, 7] and closed sequences [23, 21]. Moreover, [8, 1] consider the discovery of association rules in temporal databases and thus the extraction of temporal features of associated items. Also, [14] introduces the notion of episodes, i.e. combination of events with a partially specified order, where each episode may have some minimal duration.

## 3 Proposed Algorithms

A straightforward approach to mine frequent patterns from a database of e-sequences $D$ is to reduce the problem to a sequential pattern mining problem by converting $D$ to a transactional database $D'$. Without any loss of information, we can keep only the start and end time of each event interval. For example, for every event interval $(e_i,\ t_s,\ t_e)$ in $D$, that describes an event $e_i$ starting at $t_s$ and ending at $t_e$, we only keep $t_s$ and $t_e$ in $D'$. Now, we can apply an efficient existing sequential pattern mining algorithm, e.g., SPAM [5], to generate the set of frequent sequences $FS$ in $D'$. Every pattern in $FS$ should be post-processed to be converted to an arrangement. However, this approach has two basic drawbacks, regarding cost and efficiency: (1) post-processing can be very costly, since in the worst case the number of frequent patterns in $FS$ will be exponential ($O(2^{|N|})$), where $N$ is the number of distinct items in the database, and the cost of converting every pattern $f$ in $FS$ to an arrangement is $O(|f|^2)$, (2) the patterns in $FS$ will carry lots of redundant information. Next, we propose two efficient algorithms for mining frequent arrangements of temporal intervals that address the previous problems. Both al-

| Database $D$ | |
|---|---|
| *esid* | *e-sequence* |
| 1 | A [1, 3], B [1, 3], A [6, 12], B [8, 11], C [ 9, 10] |
| 2 | A [1, 2], B [2, 6], A [10, 12], B [11, 15], C [14, 17] |
| 3 | B [1, 3], A [4, 7], A [9, 11], B [11, 12] , C [12, 14] |
| 4 | B [1, 5], A [6, 14], B [7, 10], C [8, 9] |

**Figure 4. An e-sequence database $D$.**

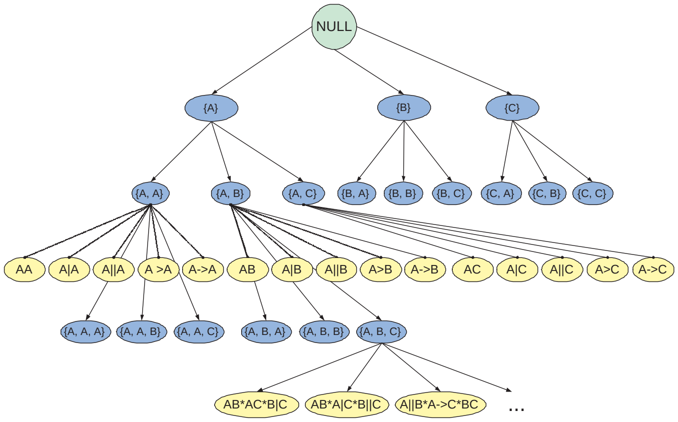


**Figure 5. An arrangement enumeration tree.**

gorithms employ a tree-based enumeration structure like the one introduced in [6]. The first algorithm uses breadth first search to generate the candidate arrangements, whereas the second uses depth first search.

### 3.1 The Arrangement Enumeration Tree

The tree-based structure used by our algorithms is called *arrangement enumeration tree*. An arrangement enumeration tree is shown in Fig. 5. Each level $k$ consists of a set of nodes, denoted as $N(k)$, that hold the complete set of $k$-arrangements. Let $n_i^k$ denote node $i$ on level $k$, where $i$ indicates the position of $n_i^k$ in the $k$-th level based on the type of traversal used by the algorithm. For every node $n_i^k \in N(k)$, we consider the arrangement $\mathcal{A}$=$\{\mathcal{E}, R\}$ defined by the node, based on which, an intermediate set of nodes (as shown in Fig. 5) is created, denoted as $IM^k(n_i^k)$, linking to $n_i^k$. Each node in $IM^k(n_i^k)$ represents a temporal relation in $R$. In the case shown in Fig. 5, $\mathcal{E} = \{A, B, C\}$ and on level 1, $N(1) = \{\{A\}, \{B\}, \{C\}\}$, i.e. we have one node for every item in $\mathcal{E}$. Then, performing temporal joins on the nodes of level 1, the set of the 2-arrangements of Level 2 is generated, with $N(2)$ = $\{\{A, A\}, \{A, B\}, \{A, C\}, \{B, A\}, \{B, B\}, \{B, C\}, \{C, A\}, \{C, B\}, \{C, C\}\}$, and for each node $n_i^2$ set $IM^k(n_i^k)$ is defined. In general, on level $k$: (1) $N(k)$ is created by joining the nodes in $N(k$-1) with those in $N(1)$, (2) for every node $n_i^k$, $IM^k(n_i^k)$ is defined and then linked to $n_i^k$. The arrangement enumeration tree is created as described above, using the set of operands defined in Section 2 and it is traversed using either breadth-first or depth-first search.

| A | |
|---|---|
| *isid* | *Intv-List* |
| 1 | [1, 3] |
| 1 | [6, 12] |
| 2 | [1, 2] |
| 2 | [10, 12] |
| 3 | [4, 7] |
| 3 | [9, 11] |
| 4 | [6, 14] |

| B | |
|---|---|
| *isid* | *Intv-List* |
| 1 | [1, 3] |
| 1 | [8, 11] |
| 2 | [2, 6] |
| 2 | [11, 15] |
| 3 | [1, 3] |
| 3 | [11, 12] |
| 4 | [1, 5] |
| 4 | [7, 10] |

**Figure 6. ISId-Lists for items A and B.**

### 3.2 BFS-based Approach

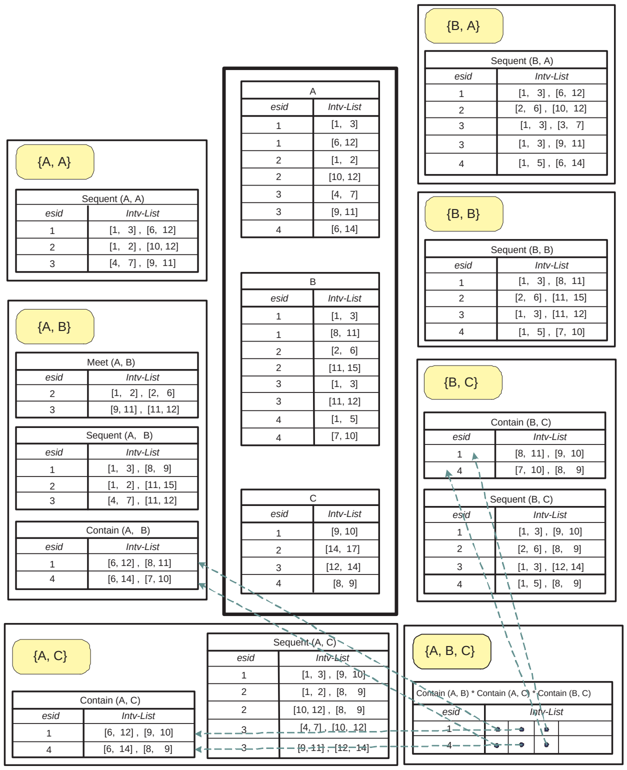
In this section we propose an event interval mining algorithm that uses the arrangement enumeration tree described above. First, we introduce the *ISIdList* structure, that attains a compact representation of the intervals and a relatively low join cost. More specifically, an ISIdList is defined for every arrangement generated by this process. The head of the list is the representation of the arrangements using $\mathcal{R}$ and the event labels comprised in it; each record is of type $(id, intv\text{-}List)$, where $id$ is the e-sequence id in $D$ that supports the arrangement, and $intv\text{-}List$ is a double-linked list of all the time intervals during which the arrangement occurs in the corresponding e-sequence in $D$.

Consider, for example, an e-sequence database $D$ with three unique items $A$, $B$ and $C$, as in Fig. 4. The ISIdLists of $A$ and $B$ is shown in Fig. 6. Let $F_k$ denote the complete set of frequent $k$-arrangements and $C_k$ the set of candidate frequent $k$-arrangements. Our algorithm will first scan $D$ to find $F_1$, i.e. the complete set of 1-arrangements. To achieve this, a scan will be performed on $D$ for every event type $e_i$. If the number of e-sequences in $D$ that contain an interval of $e_i$ satisfies the support threshold, $e_i$ will be added to $F_1$, and its ISIdList will be updated accordingly.

In order to generate the candidate 2-arrangements, we use the arrangement enumeration tree described above to get the nodes of level 2, along with the set of their corresponding intermediate nodes. Then, removing those that do not satisfy the support threshold constraint we get set $F_2$ of frequent 2-arrangements.

Moving to the next levels, i.e. generating the set of frequent $k$-arrangements, we traverse the nodes on level $k$-1. Note that these nodes correspond to the set of frequent ($k$-1)-arrangements. For every node $n_i^{k-1}$, a new node $n_i^k$ is created on level $k$, along with the set of intermediate nodes $IM^k(n_i^k)$, one for every type of correlation of the items in $n_i^k$. For every node in $IM^k(n_i^k)$ an ISIdList is created that contains: (1) the set of items of $n_i^k$, (2) the types of 2-relations between them, (3) for every type of 2-relation a pointer to the intermediate nodes on Level 2 that correspond to that 2-relation. Also, note that if an arrangement is found to be infrequent, then the node in the tree that corresponds to that arrangement is no further expanded.

The above process is more clear through the following example: consider database $D$ in Fig. 4 and assume that



| A | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [1, 3] |
| 1 | [6, 12] |
| 2 | [1, 2] |
| 2 | [10, 12] |
| 3 | [4, 7] |
| 3 | [9, 11] |
| 4 | [6, 14] |

| B | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [1, 3] |
| 1 | [8, 11] |
| 2 | [2, 6] |
| 2 | [11, 15] |
| 3 | [1, 3] |
| 3 | [11, 12] |
| 4 | [1, 5] |
| 4 | [7, 10] |

| C | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [9, 10] |
| 2 | [14, 17] |
| 3 | [12, 14] |
| 4 | [8, 9] |

| Sequent (A, A) | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [1, 3] , [6, 12] |
| 2 | [1, 2] , [10, 12] |
| 3 | [4, 7] , [9, 11] |

| Meet (A, B) | |
|---|---|
| *esid* | *Intv-List* |
| 2 | [1, 2] , [2, 6] |
| 3 | [9, 11] , [11, 12] |

| Sequent (A, B) | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [1, 3] , [8, 9] |
| 2 | [1, 2] , [11, 15] |
| 3 | [4, 7] , [11, 12] |

| Contain (A, B) | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [6, 12] , [8, 11] |
| 4 | [6, 14] , [7, 10] |

| Sequent (B, A) | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [1, 3] , [6, 12] |
| 2 | [2, 6] , [10, 12] |
| 3 | [1, 3] , [3, 7] |
| 3 | [1, 3] , [9, 11] |
| 4 | [1, 5] , [6, 14] |

| Sequent (B, B) | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [1, 3] , [8, 11] |
| 2 | [2, 6] , [11, 15] |
| 3 | [1, 3] , [11, 12] |
| 4 | [1, 5] , [7, 10] |

| Contain (B, C) | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [8, 11] , [9, 10] |
| 4 | [7, 10] , [8, 9] |

| Sequent (B, C) | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [1, 3] , [9, 10] |
| 2 | [2, 6] , [8, 9] |
| 3 | [1, 3] , [12, 14] |
| 4 | [1, 5] , [8, 9] |

| Contain (A, C) | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [6, 12] , [9, 10] |
| 4 | [6, 14] , [8, 9] |

| Sequent (A, C) | |
|---|---|
| *esid* | *Intv-List* |
| 1 | [1, 3] , [9, 10] |
| 2 | [1, 2] , [8, 9] |
| 2 | [10, 12] , [8, 9] |
| 3 | [4, 7] , [10, 12] |
| 3 | [9, 11] , [12, 14] |

| Contain (A, B) * Contain (A, C) * Contain (B, C) | | | | |
|---|---|---|---|---|
| *esid* | *Intv-List* | | | |
| 1 | ● | ● | ● | |
| 4 | ● | ● | ● | |

**Figure 7. The set of frequent $2$ and $3$-arrangements.**

$min\_sup = 2$. Scanning $D$ and filtering with $min\_sup$, we get $F_1 = \{\{A\}, \{B\}, \{C\}\}$. Based on $F_1$ and the enumeration tree, set $F_2$ of the frequent 2-arrangements is generated. In our case, we get all the possible pairs of the 1-arrangements in $F_1$, i.e. N(2), and for every pair of events in the arrangements, $D$ is scanned to get all the types of relations between them, i.e. $IM^2$. If these relations satisfy the support threshold they are added to $F_2$. Then we produce $F_3$ based on $F_2$. The algorithm first creates $N(3)$, following a breadth-first search traversal, along with the set of intermediate nodes. Every node in $IM^3$ that satisfies $min\_sup$ is added to $F_3$, which in our case consists of only one arrangement: $A > B * A > C * B > C$. $F_1$, $F_2$ and $F_3$ are shown in Fig. 7. The main steps of this approach are described in Algorithm 1, considering an input database $D$ and a minimum support threshold $min\_sup$.

### 3.3 DFS-based Approach

In a breadth-first search approach the arrangement enumeration tree is explored in a top-bottom manner, i.e. all the children of a node are processed before moving to the next level. On the other hand, when using a depth-first search approach, we must completely explore all the sub-arrangements on a path before moving to another one. A DFS-based approach for mining frequent sequences has been proposed in [20]. Based on this, the previous algorithm can be easily modified to use a depth-first search candidate generation method. This can be done by adjusting function $generate\_candidates()$ so that it follows a depth-first search traversal. Consider the previous example: our algorithm will first generate node $n_1^1 = \{A\}$ followed by $IM(n_1^1)$, then $n_1^2 = \{A,\ A\}$ followed by $IM(n_1^2)$, and so on.

The advantage of DFS over BFS is that DFS can lead us very quickly to large frequent arrangements and therefore we can avoid some expansions in the other paths in the tree. For example, say that a $k$-arrangement $\mathcal{A}$ is found to be frequent. Then, the set of all sub-arrangements of $\mathcal{A}$ will also be frequent according to the Apriori principle. Thus, we can skip those expansions in the enumeration tree reducing the cost of computation. To do so, one more step is added to Algorithm 1: when a node is found to contain a frequent arrangement, each sub-arrangement is added to $F$ and the corresponding expansions are made on the tree. However, in BFS there is more information available for pruning. For example, knowing the set of 2-arrangements before constructing the set of 3-arrangements can prevent us from making expansions that will lead to infrequent arrangements. This information, however, is not available in DFS.

### 3.4 Hybrid DFS-based Approach

In this section we consider a hybrid event interval mining approach based on the following observation: since the ISIdLists contain pointers to the nodes on the second Level of the tree, a DFS-based approach would be inappropriate since for every node $n_i^k$ we would have to scan the database multiple times to detect the set of 2-relations among the items in that node. In the BFS-based approach these nodes will already be available, since they have been generated in the second step of the algorithm. Thus, we use a hybrid DFS approach that generates the first two levels of the tree using BFS and then follows DFS for the rest of the tree. This would compensate for the multiple database scans discussed above, since the set of frequent 2-arrangements will already be available thereby eliminating the need for multiple database scans.

## 4 Experimental Evaluation

In this section we present experimental results on the performance of our two algorithms in comparison with SPAM [5]. All the experiments have been performed on a 2.8Ghz Intel(R) Pentium(R) 4 dual-processor machine with 2.5 gigabytes main memory, running Linux with kernel 2.4.20. The algorithms have been implemented in C++, compiled using g++ along with the -O3 flag, and their runtime has been measured with the output turned off. Note that for

The code was obtained from: `http://himalaya-tools.sourceforge.net/Spam/`.

```
input  : $D$: a database of e-sequences.
         $min\_sup$: minimum support threshold.
output : The set $F$ of the frequent arrangements in $D$.
$F = \emptyset$;
foreach event type $e_i$ do
    if $e_i$ exists in $D$ then
        $C_1 = C_1 \cup e_i$;
    end
end
$F_1 = \{e_i \in C_1 \mid e_i.cupport \geq min\_sup\}$;
while $F_{k-1} \neq \emptyset$ do
    $N(k) = generate\_candidates\ (N(k-1), N(2))$;
    // the next set of nodes on the tree is determined.
    // It is based on a BFS traversal and it uses the
    // nodes on level (k-1) and on level 2.
    foreach node $n_i^k \in N(k)$ do
        $IM^k(n_i^k) = generate\_relations()$;
        // this function generates the nodes in $IM^k$,
        // along with their ISIdLists.
        $C_k = IM^k$;
        foreach candidate $c \in C_k$ do
            if $c.support < min\_sup$ then
                $C_k.remove(c)$; // removes $c$ from $C_k$.
                $prune\_subtree(c)$; // prunes subtree($c$).
            end
        end
        $F_k = C_k$;
    end
    $F = F \cup F_k$;
end
```

Algorithm 1: *A BFS-based algorithm for discovering frequent temporal arrangements in a database of e-sequences.*

SPAM, the post-processing time of converting the sequential patterns to arrangements has not been counted. Also, as mentioned in Sec. 3, SPAM is tuned as follows: for every event interval we keep only the start and end time; as for the postprocessing phase the frequent arrangements are extracted from the sequential patterns as described in Sec. 3. The patterns found by SPAM comprise a set of start and end points of event intervals, which are converted to arrangements after the postprocessing phase. SPAM finds all patterns found by our two algorithms. However, it produces a great number of redundant patterns. For our experimental evaluation we have used both real and synthetic datasets.

## 4.1 Experiments on Real Data

We have performed a series of experiments on two real datasets. One was an American Sign Language (ASL) database ($http : //www.bu.edu/asllrp/$) and the other was a sample network dataset of ODFlows taken from Abilene, which is an Internet2 backbone network, connecting over 200 US universities and peering with research networks in Europe and Asia. It consists of 11 Points of Presence (PoPs), spanning the continental US. Three weeks of sampled IP-level traffic flow data was collected from every PoP in Abilene for the period December 8, 2003 to December 28, 2003.

### 4.1.1 Experiments on the ASL SignStream Database

The first series of experiments have been performed on the American Sign Language database created by the National Center for Sign Language and Gesture Resources at Boston University. The SignStream database consists of a collection of $884$ utterances, where each utterance associates a segment of video with a detailed transcription. Every utterance contains a number of ASL gestural and grammatical fields (e.g. eye-brow raise, head tilt forward, wh-question), each one occurring over a time interval. We first tested our algorithms on a small part of the database that only comprised all the utterances that contained a "wh-question". Our goal was to detect all frequent arrangements that occurred during a "wh-question". In this dataset, called $Dataset$ 1, the number of e-sequences was $73$ with an average number of items per sequence equal to $52$. As shown in Fig. 8(a), Hybrid DFS outperformed both BFS and SPAM for supports less than $30\%$. Then we tested our algorithms on the whole Signstream database that contained $884$ utterances with an average sequence of $102$ items per sequence. The algorithms have been tested for various supports and have been compared in terms of runtime. The experimental results in Figure 8(b) show that the Hybrid DFS-based approach outperforms the BFS-based especially in small supports. In both cases SPAM starts with a runtime between that of BFS and Hybrid DFS and for small supports the runtime increases dramatically.

### 4.1.2 Experiments on Network Data

Our algorithms have also been tested on a network dataset of $960$ sequences with an average sequence size of $100$ items per sequence. The data has been obtained from a collection of ODFlows. We have selected two routers that were shown to have a high communication rate with each other, and have monitored the IP connections from one (LOSA: router in LA) to the other (ATLA: router in Altanta) for three days. A sequence in our dataset is the set of IP connections from LOSA to ATLA for every $15$ minutes. Due to the huge number of IP addresses, we have selected $200$ IPs that appear most frequently in these three days.

Our experimental results are shown in Figure 8(c), where again Hybrid DFS outperforms both BFS and SPAM in low supports.

## 4.2 Experiments on Synthetic Data

Due to the relatively small size of the current SignStream database, we have generated numerous synthetic datasets to

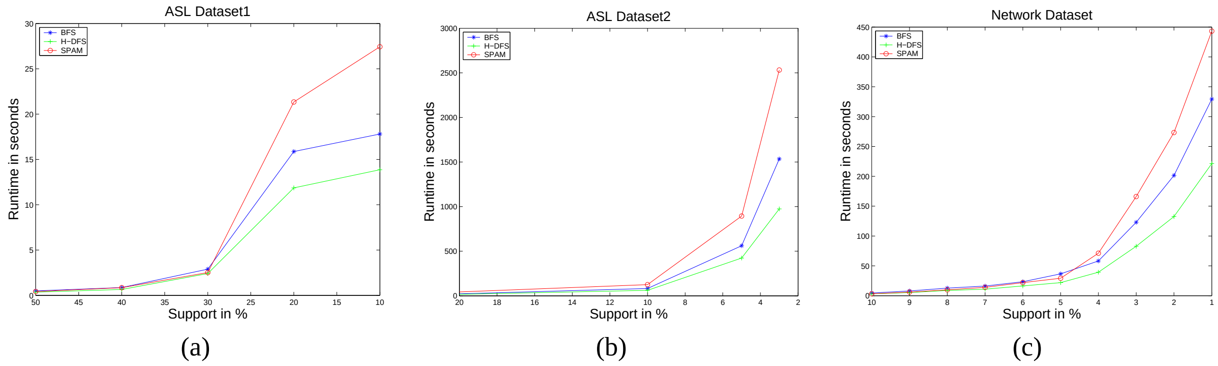


**Figure 8.** Results on Real Datasets: (a) ASL Dataset 1: $|S|$: 73, $|A|$: 52, $|\mathcal{E}|$: 400.; (b) ASL Dataset 2: $|S|$: 884, $|A|$: 102, $|\mathcal{E}|$: 400.; (c) Network Dataset: $|S|$: 960, $|A|$: 100, $|\mathcal{E}|$: 200 (where $|S|$ denotes the size of the dataset, $|A|$ the average sequence size), and $|\mathcal{E}|$ the number of distinct items in the dataset.

test the efficiency of our algorithms.

### 4.2.1 Synthetic Data Generation

The following factors have been considered for the generation of the synthetic datasets: (1) number of e-sequences, (2) average e-sequence size, (3) number of distinct items, (4) density of frequent patterns. Using different variations of the above factors we have generated several datasets. In particular, our datasets were of sizes 200, 500, 1000, 2000, 5000 and 10000, with average sequence sizes of 3, 10, 50, 100 and 150 items per e-sequence. Moreover, we have tried various numbers of distinct items, i.e. 400, 600 and 1000. Also, we have considered different densities of frequent patterns. We first created a certain number of frequent patterns that with medium support thresholds of 20% (sparse), 40% (medium density) and 60% (dense) would generate a lot of frequent patterns and then added random event intervals on the generated sequences.

### 4.2.2 Experimental Results

The experimental results have shown that Hybrid DFS clearly outperforms BFS, and especially in low support values and large database sizes Hybrid DFS is twice as fast as BFS. Regarding the performance of SPAM, we have concluded that in medium support values and small database sizes SPAM performs better than BFS but worse than Hybrid DFS, whereas in small support values and large datasets BFS outperforms SPAM. We compared the three algorithms on several small, medium and large datasets for various support values. The results of these tests are shown in Fig. 9. Due to space limitations we present only a portion of our results focusing on the most significant ones. As expected, SPAM performs poorly in large sequences and small supports. This behavior is expected since for every arrangement produced by BFS and Hybrid DFS, SPAM generates all the possible subsets of the start and end points of the events in that arrangement. As the database size grows along with the average e-sequence size, SPAM will be producing a great number of redundant frequent patterns that yield to a rapid increase of its runtime.

## 5 Conclusion

We have formally defined the problem of mining frequent temporal arrangements of event interval sequences and presented two efficient methods to solve it. The key novelty of our methods is that they do not make the assumption that events occur instantaneously. The BFS-based approach uses an arrangement enumeration tree to discover the set of frequent arrangements. The DFS-based method further improves performance over BFS by reaching longer arrangements faster and hence eliminating the need for examining smaller subsets of these arrangements. Our experimental evaluation demonstrates the applicability and usefulness of our methods. An interesting direction for future work is to incorporate additional constraints and partial knowledge about the frequency of some arrangements.

## References

[1] T. Abraham and J. F. Roddick. Incremental meta-mining from large temporal data sets. In *ER '98: Proceedings of the Workshops on Data Warehousing and Data Mining*, pages 41–54, 1999.

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

[3] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of IEEE ICDE*, pages 3–14, 1995.

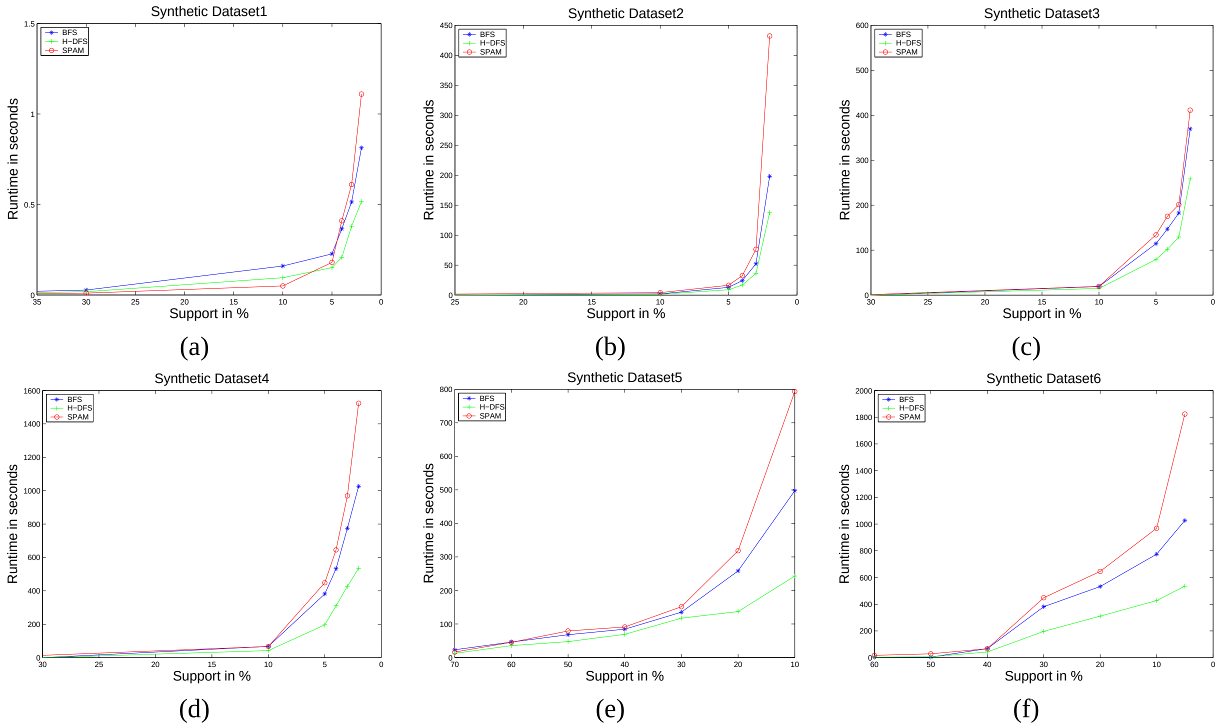
[4] J.F. Allen and G. Ferguson. Actions and events in interval temporal logic. Technical Report 521, The University of Rochester, July 1994.

[5] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using a bitmap representation. In *Proc. of ACM SIGKDD*, pages 429–435, 2002.

[6] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proc. of ACM SIGMOD*, pages 85–93, 1998.

[7] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *Proc. of IEEE ICDE*, pages 443–452, 2001.

[8] X. Chen and I. Petrounias. Mining temporal features in association rules. In *Proc. of PKDD*, pages 295–300, London,



(a) (b) (c)

(d) (e) (f)

**Figure 9.** Results on Synthetic Datasets: (a) Dataset 1: $|S|$: 200, $|A|$: 20, $|\mathcal{E}|$: 200, frequent patterns of medium density.; (b) Dataset 2: $|S|$: 1000, $|A|$: 50, $|\mathcal{E}|$: 600, sparse frequent patterns.; (c) Dataset 3: $|S|$: 5000, $|A|$: 50, $|\mathcal{E}|$: 400, sparse frequent patterns.;(d) Dataset 4: $|S|$: 1000, $|A|$: 100, $|\mathcal{E}|$: 800, frequent patterns of medium density.;(e) Dataset 5: $|S|$: 5000, $|A|$: 100, $|\mathcal{E}|$: 600, dense frequent patterns.;(f) Dataset 6: $|S|$: 10000, $|A|$: 100, $|\mathcal{E}|$: 800, dense frequent patterns.

UK, 1999. Springer-Verlag.

[9] J. Fei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. of IEEE ICDE*, pages 215–224, 2001.

[10] M. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *Proc. of VLDB*, pages 223–234, 1999.

[11] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.C. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *Proc. of ACM SIGKDD*, pages 355 – 359, 2000.

[12] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD*, pages 1–12, 2000.

[13] M. Leleu, Ch. Rigotti, J. Boulicaut, and G. Euvrard. Go-spade: Mining sequential patterns over databases with consecutive repetitions. In *Proc. of MLDM*, pages 293–306, 2003.

[14] H. Mannila, H. Toivonen, and A. Verkamo. Discovering frequent episodes in sequences. In *Proc. of ACM SIGKDD*, pages 210–215, 1995.

[15] C. Neidle, S. Sclaroff, and V. Athitsos. Signstream: A tool for linguistic and computer vision research on visual-gestural language data. *Behavior Research Methods, Instruments and Computers*, 33:311–320, 2001.

[16] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of ICDT*, pages 398–416, 1999.

[17] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *Proc. of DMKD*, pages 11–20, 2000.

[18] J. Pei, J. Han, and W. Wang. Constraint-based sequential pattern mining in large databases. In *Proc. of CIKM*, pages 18–25, 2002.

[19] M. Seno and G. Karypis. Slpminer: An algorithm for finding frequent sequential patterns using length-decreasing support constraint. In *Proc. of IEEE ICDM*, pages 418–425, 2002.

[20] I. Tsoukatos and D. Gunopulos. Efficient mining of spatiotemporal patterns. In *Proc. of the SSTD*, pages 425–442, 2001.

[21] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Proc. of IEEE ICDE*, pages 79–90, 2004.

[22] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proc. of ACM SIGKDD*, pages 236–245, 2003.

[23] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large databases. In *Proc. of SDM*, 2003.

[24] M. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.

[25] M. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proc. of SIAM*, pages 457–473, 2002.