ALGORITHMS FOR MINING PARALLEL-OF-SERIAL EPISODES

By
GUOHUAN WANG
Master of Science
Tianjin University
Tianjin, China
1997

Bachelor of Science
Dalian University of Technology
Dalian, China
1992

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in the partial fulfillment of
the requirement for
the Degree of
MASTER OF SCIENCE
May, 2004

ALGORITHMS FOR MINING

PARALLEL-OF-SERIAL

EPISODES

Thesis Approved

[illegible]

Thesis Advisor

Blayne E. Mayfield

[illegible]

[illegible]

Dean of the Graduate College

ACKNOWLEDGEMENT

I would like to give my sincere appreciation to my advisor, Dr. H. K. Dai for his intelligent guidance, continuous support, and hard work on this thesis. My appreciation extends to other professors served in my thesis committee: Dr. K. M. George and Dr. Blayne E. Mayfield for their willingness to offer opinions and suggestions for the improvement of my knowledge and experience.

I would like to thank Mr. Michael G. Carrel for kindly providing web log data to support my thesis work.

Without a doubt, my family members have been the largest supporters throughout my academic career. I would like to thank my parents, Yongjiang Wang and Qiaoru Sun, my brother Guoxin Wang and his wife Yue Sun for their continuous love and words of encouragement.

Last but not least, I would like to thank my wife, Weixiu Kong for her encouragement, understanding, and love since the first day we met.

TABLE OF CONTENTS

1. INTRODUCTION 1
1.1 Transaction Database Mining 5
1.2 Sequence Data Mining 9
2. COMPLEX EPISODE AND PARALLEL-OF-SERIAL EPISODE 15
2.1 Complex Episode 15
2.2 Parallel-of-Serial Episode 17
3. ALGORITHMS 19
3.1 Generation of Candidate Parallel-of-Serial Episodes 19
3.2 Recognition of Frequent Parallel-of-Serial Episodes 21
4. EXPERIMENTS 26
4.1 A Simple Example on Parallel-of-Serial Episode Mining 26
4.2 Parallel-of-Serial Episode Mining on a Large Web Log Data Set 29
5. CONCLUSION 38
REFERENCES 40
APPENDIX 41

LIST OF FIGURES

Figure 1-1. Serial episode α, parallel episode β, and complex episode γ 10

Figure 1-2. Four consecutive windows along the time axis in the episode-recognizing algorithm for parallel episode α. 13

Figure 2-1. Complex episode γ and a serial combination of three episodes. 16

Figure 2-2. A PoS-episode γ 17

Figure 2-3. Decomposing a PoS-episode γ into two serial subepisodes γ' and γ'' 17

Figure 3-1. Construct two deterministic finite automata M_1 and M_2 for PoS-episode α . 21

Figure 4-1. Structure of a three-page website. 26

Figure 4-2. The web log of a three-page website. 27

Figure 4-3. Condensing of web log data. 32

Figure 4-4. Ratio of candidate and frequent episode numbers under different window size. 33

Figure 4-5. A linear relation between window sizes and minimum frequency ratios when F_{fresh} reaches its maximum value.. 36

LIST OF TABLES

Table 3-1. Description of the last transitions of automata pair (M_1, M_2). 23

Table 4-1. Serial episode mining result of a three-page website. 28

Table 4-2. Candidate serial episodes of size 4. 28

Table 4-3. The PoS-episode mining result of a three-page website. 29

Table 4-4. Basic information of the experiment data set. 30

Table 4-5. Frequent PoS-episodes number in different combination of window width and frequency ratio. 32

Table 4-6. Detailed result of PoS-episode mining. 34

Table 4-7. Evaluate the discreteness of this web log episode mining 37

1. INTRODUCTION

Data mining is the task that extracts or "mines" knowledge from large amounts of data. In recent years, data mining has attracted a great deal of attention in the information industry due to the wide availability of huge amounts of data and the imminent need for turning such data into useful information and knowledge. The obtained knowledge can be used for applications such as science exploration, engineering design, production control, business management, and market analysis.

Data mining is the result of the natural evolution of information technology and can be traced in the database industry in the development of the following functionalities: data collection and database creation, data management (including data storage and retrieval, and database transaction processing), and data analysis and understanding (involving data warehousing and data mining). As an example, data collection and database creation mechanisms acted as a prerequisite for later development of effective mechanisms for data storage and retrieval, and query and transaction processing. With the blooming of database systems that have query and transaction processing as common practice, data analysis and understanding has turn out to be the next target.

Data mining is an essential step in the process of knowledge discovery in databases. Knowledge discovery as a process consists of an iterative sequence of the following seven steps: data cleaning, data integration, data selection, data transformation, data mining, pattern evaluation, and knowledge presentation. The data mining step may interact with the user or a knowledge base. The interesting patterns are presented to the user, and may be stored as new knowledge in the knowledge base.

Data mining can be applicable to different kind of information repository such as relational databases, data warehouses, transactional databases, advanced database systems, flat files, and the World Wide Web (WWW). Advanced database systems include object-oriented and object-relational databases, and specific application-oriented databases, such as spatial databases, time-series databases, text databases, and multimedia databases. Each of the repository systems has its own challenges and techniques of data mining.

In general, data mining tasks can be divided into two groups: descriptive mining and predictive mining. Descriptive mining tasks extract the general characteristics of data from the database while predictive mining tasks emphasize predicating future activities based on the analysis of current data.

In some cases, users do not know what kinds of patterns in their data may be interesting, and may search for several different kinds of patterns in parallel. This demands the data mining system to mine multiple kinds of patterns for different user expectations or applications. Furthermore, data mining systems should be able to discover patterns at various granularities. On the other hand, users should have the flexibility to specify hints to guide or focus the search for interesting patterns. Since some patterns may not hold for all of the data in the database, a measure of certainty or "trustworthiness" is usually associated with each discovered pattern.

A data mining system may generate thousands or even millions of patterns, or rules. In general, only a small fraction of the generated patterns would actually be of interest to a given user. A pattern is interesting if it (1) could be easily understood by humans, (2) is valid on new or test data with some degree of certainty, (3) is potentially useful, and (4) is novel. A pattern is also interesting if it confirms an assumption. An interesting pattern represents knowledge.

There are two kinds of measures of pattern interestingness: objective interestingness measures and subjective interestingness measures [4]. Objective interestingness measures are based on the structure of discovered patterns and the statistics underlying them. For two patterns X and Y constrained on transactions, an association rule denoted by $X \Rightarrow Y$ indicates the conditional consideration of Y under the hypothesis of X. One objective measure for association rules of the form $X \Rightarrow Y$ is the rule support, representing the percentage of transactions from a transaction database that the given rule satisfies. This is taken to be the probability Pr(a transition satisfying both X and Y). Another objective measure for association rules is confidence, which assesses the degree of certainty of the detected association. This is taken to be the conditional probability Pr(a transition satisfying Y | a transition satisfying X), that is, the probability that a transaction containing X also contains Y. More formally, support and confidence are defined as:

support $(X \Rightarrow Y) = Pr$(a transition satisfying both X and Y),

confidence$(X \Rightarrow Y) = Pr$(a transition satisfying Y | a transition satisfying X).

Subjective interestingness measures are based on user beliefs in the data. These measures find patterns interesting if they are unexpected (contradicting a user's belief) or offer strategic information on which the user can act.

Expecting data mining systems to generate all of the possible patterns is unrealistic and inefficient. Instead, constraints and interestingness measures provided by user should be used to guide the search. This is often sufficient to ensure the completeness of the algorithm. As an example, constraints and interestingness measures can be applied to association-rule mining to ensure that all qualified patterns and rules can be found out in the mining. It will be much more efficient for users and data mining systems if data mining can generate only interesting patterns. However, such optimization remains a challenging issue in data mining technology.

Efficiently selecting valuable patterns to the given user from large amounts of potential patterns is essential for data mining task. This can be done after the data mining step by ranking the discovered patterns according to their interestingness, filtering out the uninteresting ones. More importantly, this step can be pushed into the discovery process of data mining and thus improve the search efficiency by pruning away subsets of the pattern space that do not satisfy pre-specified interestingness constraints [7] [8].

1.1 Transaction Database Mining

With massive amounts of data continuously being collected and stored, many industries are becoming interested in mining association rules from their databases. Association-rule mining finds interesting association or correlation relationships among a large set of data items. The discovery of interesting association relationships among huge amounts of business transaction records can help in many business decision making processes, such as catalog design, cross-marketing, and loss-leader analysis.

A typical example of association-rule mining is the market basket analysis [1]. The transaction database of the merchant is analyzed between the different items that customers place in their "shopping baskets" by data mining process to find out customer buying habits. Retailers can develop marketing strategies from the data mining result by gaining insight into which items are frequently purchased together by customers.

A Boolean variable of an item represents the presence or absence of that item when we think of the universe as the set of items available at the store. Each basket can then be represented by a Boolean vector of values assigned to these variables. All the present items in a Boolean vector consist of an item set. Conventionally, we use the term *k*-itemset to describe an item set with cardinality *k*. Through analyzing the Boolean vectors, one can get the buying patterns that reflect items that are frequently associated or purchased together. These patterns can be represented in the form of association rules.

Apriori is an influential algorithm [2] for mining frequent itemsets for Boolean association rules. The name of the algorithm is based on the fact that the algorithm uses prior knowledge of frequent itemset properties. Apriori employs an iterative approach known as a level-wise search, where k-itemsets are used to explore $(k + 1)$-itemsets. First, the set of frequent 1-itemsets is found. This set is denoted by L_1. The set L_1 is used to find L_2, the set of frequent 2-itemsets, which is used to find L_3, and so on, until no more frequent k-itemsets can be found. The finding of each L_k requires one full scan of the database.

To improve the efficiency of the level-wise generation of frequent itemsets, an important property called the *Apriori property*, is used to reduce the search space. Apriori property is known as that all nonempty subsets of a frequent itemset must also be frequent. This property belongs to a special category of properties called anti-monotone, in the sense that if a set cannot pass a test, all of its supersets will fail the same test as well.

A typical Apriori algorithm and its related procedures are shown as follows:

Algorithm Apriori. Find frequent itemsets using an iterative level-wise approach based on candidate generation.
Input: Database, D, of transactions; minimum support threshold, *min_sup*.
Output: L, frequent itemsets in D.
Method:

```
(1)  L_1 = find_frequent_1-itemsets(D);
(2)  for(k=2; $L_{k-1} \neq \phi$ ; k++) {
(3)      $C_k$ = apriori_gen ($L_{k-1}$, min_sup);
(4)      for each transaction $t \in D$ { // scan D for counts
(5)          $C_t$ = subset($C_k$, t); // get the subsets of t that are candidates
(6)          for each candidate $c \in C_t$
(7)              c.count++;
(8)          }
(9)          $L_k = \{c \in C_k \mid c.count \geq min\_sup\}$
(10) }
(11) return $L = \cup_k L_k$;
```

```
procedure apriori_gen($L_{k-1}$; frequent (k-1)-itemsets; min_sup; minimum support threshold)
(1)  for each itemset $l_1 \in L_{k-1}$
(2)      for each itemset $l_2 \in L_{k-1}$
(3)          if $(l_1[1] = l_2[1]) \wedge (l_1[2] = l_2[2]) \wedge \cdots \wedge (l_1[k-2] = l_2[k-2]) \wedge (l_1[k-1] < l_2[k-1])$ then {
(4)              $c = l_1 \bowtie l_2$; // join step: generate candidates
(5)              if has_infrequerrt.subset(c, $L_{k-1}$) then
(6)                  delete c; // prune step; remove unfruitful candidate
(7)              else add c to $C_k$;
(8)              }
(9)  return $C_k$;

procedure has_infrequent_subset(c: candidate k-itemset; $L_{k-1}$:frequent (k-1)-itemsets);
// use prior knowledge
(1)  for each(k-1)-subset s of c
(2)      if $s \notin L_{k-1}$ then
(3)          return TRUE;
(4)  return FALSE;
```

The apriori_gen procedure generates the candidates and then uses the Apriori property to eliminate those having a subset that is not frequent. The apriori_gen procedure performs two kinds of actions, namely, join and prune. By convention, we assume that items within a transaction or itemset have been sorted in lexicographic order before applying Apriori algorithm. In the join component (steps 1-4), a new candidate k-itemset c is generated as $(l_1[1], l_1[2], \ldots, l_1[k-1], l_2[k-1])$ in step 4 if the condition in step 3 is satisfied. The prune component (steps 5-7) employs the Apriori property to remove candidates that have a subset that is not frequent.

Once the frequent itemsets from transactions in a database D have been found, it is straightforward to generate strong association rules from them, where strong association rules satisfy both minimum support and minimum confidence. This can be done using the following equation for confidence, where the conditional probability is expressed in terms of itemset support count:

$$\text{confidence}(A \Rightarrow B)$$

$$= Pr(\text{a transaction containing itemset } B \mid \text{a transaction containing itemset } A)$$

$$= \frac{\text{support\_count}(A \cup B)}{\text{support\_count}(A)}$$

where support_count ($A \cup B$) is the number of transactions containing the itemset $A \cup B$, and support_count (A) is the number of transactions containing the itemset A. Based on this equation, association rules can be generated as follows:

- For each frequent itemset *l*, generate all nonempty subsets of *l*.
- For every nonempty subset s of *l*, output the rule "$s \Rightarrow l$" if

  $$\frac{\text{support\_count}(l)}{\text{support\_count}(s)} \geq \min\_conf,$$

  where *min_conf* is the minimum confidence threshold.

Since the rules are generated from frequent itemsets, each one automatically satisfies minimum support.

In many cases the Apriori candidate generate-and-test method reduces the size of candidate sets significantly and leads to good performance gain. However, it may suffer from two nontrivial costs:

- It may need to generate a huge number of candidate sets.
- It may need to repeatedly scan the database and check a large set of candidates by pattern matching. This is especially the case for mining long patterns.

Frequent pattern (FP) [6] growth is a method of mining frequent itemsets without candidate generation. It constructs a highly compact data structure (an FP-tree) to compress the original transaction database. Rather than employing the generate-and-test strategy of Apriori-like methods, it focuses on frequent pattern (fragment) growth, which avoids costly candidate generation, resulting in greater efficiency.

1.2 Sequence Data Mining

Recent years, besides mining on relational databases, transactional databases, vast amounts of data in various complex forms (e.g., structured and unstructured, hypertext and multimedia) have been growing explosively owing to the rapid progress of data collection tools, advanced database system technologies, and WWW technologies. Therefore, an increasingly important task in data mining is to mine complex types of data, including time-series data, sequence data, text data, and WWW.

In our proposed work, we will focus on sequence data mining. Examples of such data are alarms in a telecommunication network, user interface actions, crimes committed by a person, occurrences of recurrent illnesses, etc. Abstractly, such data can be viewed as a sequence of events, where each event has an associated time of occurrence.

Given a set E of event types, an event is a pair (A, t) where $A \in E$ is an event type and t is an integer, the occurrence time of the event. We assume that all events will last one time unit. An event sequence on E is a triple (s, T_s, T_e), where $s = <(A_1, t_1), (A_2, t_2), \ldots, (A_n, t_n)>$ is an ordered sequence of events such that $A_i \in E$ for all $i = 1, 2, \ldots, n$, and $t_i \leq t_{i+1}$ for all $i = 1, 2, \ldots, n-1$. Further on, T_s and T_e are integers: T_s is called the starting time and T_e the ending time, and $T_s \leq t_i < T_e$ for all $i = 1, 2, \ldots, n$.

A window on an event sequence (s, T_s, T_e) is an event sequence (w, t_s, t_e), where $t_s < T_e$ and $t_e > T_s$, and w consists of those pairs (A, t) from s where $t_s \le t < t_e$. We abbreviate an event sequence (s, T_s, T_e) and a window (w, t_s, t_e) by s and w, respectively. The time span t_e - t_s is called the width of the window w, and it is denoted by width(w). Given an event sequence s and an integer w_width, we denote by W(s, w_width) the set of all windows w on s such that width(w) = w_width.

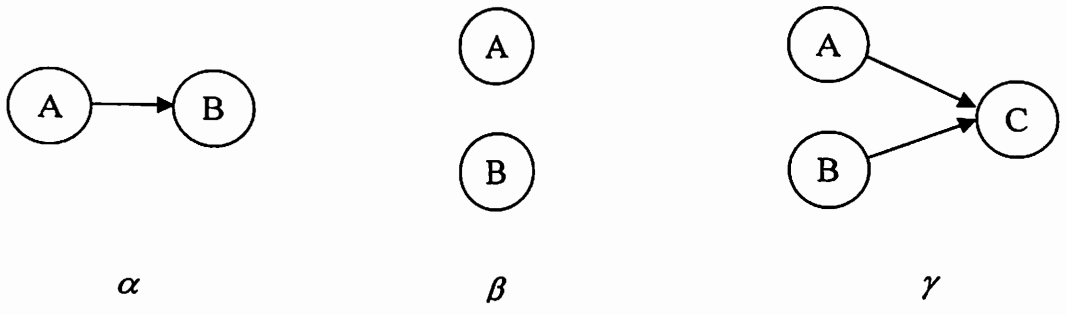

Figure 1-1. Serial episode α , parallel episode β , and complex episode γ .

An episode can be understood as an acyclic directed graph with node-set V and edge-set $\le$. An episode α is a triple $(V, \le, g)$, where V is a set of nodes, $\le$ is a partial order on V, and $g : V \to E$ is a mapping associating each node with an event type. We define the size of episode α , $|\alpha|$ as $|V|$. Depending on the graph structure, we can divide episodes into three categories: serial episodes, parallel episodes, and complex episodes as shown in Figure 1-1. An episode α is parallel if the partial order $\le$ is trivial (i.e., $x \not\le y$ for all x, $y \in V$ such that $x \neq y$). An episode α is serial if the relation $\le$ is a total order (i.e., $x \le y$ or $y \le x$ for all x, $y \in V$). An episode α is complex if it is not serial or parallel. A complex episode can be reduced to the recognition of a hierarchical combination of serial and parallel episodes.

An episode $\beta = (V', \leq', g')$ is a subepisode of $\alpha = (V, \leq, g)$, denoted by $\beta \preceq \alpha$, if there exists an injective mapping $f: V' \to V$ such that $g'(v) = g(f(v))$ for all $v \in V'$, and for all $v, w \in V'$, if $v \leq' w$ then $f(v) \leq f(w)$. An episode α is a superepisode of β if and only if $\beta \preceq \alpha$. We write $\beta \prec \alpha$ if $\beta \preceq \alpha$ and $\alpha \not\preceq \beta$.

An episode $\alpha = (V, \leq, g)$ occurs in an event sequence s = (<(A_1, t_1), (A_2, t_2), ... ,(A_n, t_n)>, T_s, T_e) if there exists an injective mapping $h: V \to \{1, \ldots, n\}$ from node-set of α to the set of all event-indices of s such that $g(v)=A_{h(v)}$ for all $v \in V$, and for all v, $w \in V$ with $v \neq w$ and $v \leq w$, we have $t_{h(v)} < t_{h(w)}$.

The frequency of an episode is the fraction of windows in which the episode occurs out of all possible windows. Given an event sequence s = (s, T_s, T_e) and a window width *w_width*, the frequency of an episode α in s is:

$$fr(a, s, w\_width) = \frac{|\{w \in W(s, w\_width) \mid \alpha \text{ occurs in } W\}|}{|W(s, w\_width)|}.$$

We say that α is frequent if $fr(a, s, w\_width) \geq min\_fr$, where *min_fr* is a frequency threshold given by the user. The collection of all frequent episodes is denoted by $F(s, w\_width, min\_fr)$ with respect to the given s, *w_width*, and *min_fr*.

Once we find all frequent episodes, we can use them to generate rules that describe the relationship between episodes. An episode rule is an expression $\beta \Rightarrow \gamma$ where β is a subepisode of γ, i.e., $\beta \preceq \gamma$. The confidence of the episode rule is:

$$\text{confidence}(\beta \Rightarrow \gamma) = \frac{fr(\gamma, s, w\_width)}{fr(\beta, s, w\_width)}.$$

The algorithm that discovers all frequent episodes [3] [5] is an Apriori-like algorithm that has been described in Section 1.1 Transaction Database Mining. It performs a level-wise search. From the episode set with only one event, i.e., set of all size-1 episodes, the search algorithm first computes a collection of candidate episodes, then checks the frequencies by scanning the event sequence. The episodes with frequency of at least *min_fr* form the frequent episode set of current episode size. The algorithm then computes the collection of candidate episodes for next episode size and generates the frequent episode set of next episode size accordingly, until no frequent episodes are generated at certain episode size.

The candidate-generation algorithms for parallel episodes and serial episodes are slightly different, but both of them follow the same logic as the *apriori_gen*() procedure. Since the partial order $\leq$ is trivial for parallel episodes, the number of parallel episode candidates is smaller than that of serial episodes. In the candidate-generation algorithm, a parallel episode is represented as a lexicographically sorted array of event types. Parallel and serial episodes in their episode collections are also sorted by lexicographical order. Candidates can be generated by arranging appropriate combinations of two episodes of size l that share the first l-1 common events.

Comparing with the candidate-generation algorithms, the episode-recognizing algorithms for parallel episodes and serial episodes are in different approaches. For parallel episode recognition, each candidate parallel episode α is associated with an event counter, namely $\alpha.event\_count$, to indicate the number of events of α in the window. If at some point when we slide the window, the $\alpha.event\_count = |\alpha|$, it means that α fully enters in the window, as W_2 shown in Figure 1-2. While α remains in the window at the last timestamp, if we observe that $\alpha.event\_count < |\alpha|$ at the current timestamp, it means that from this point α is no longer entirely in the window. We maintain $\alpha.freq\_count$ to hold the number of windows between the enter point and exit point. At the end of the algorithm, $\alpha.freq\_count$ contains the total number of windows in which α occurs.

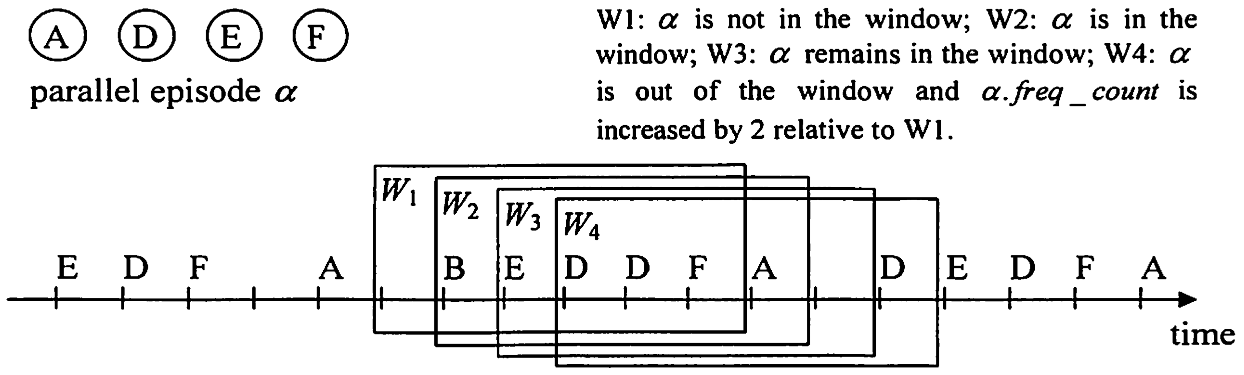
A
D
E
F
parallel episode α
W1: α is not in the window; W2: α is in the window; W3: α remains in the window; W4: α is out of the window and $\alpha.freq\_count$ is increased by 2 relative to W1.
W_1
W_2
W_3
W_4
E D F A B E D D F A D E D F A
time

Figure 1-2. Four consecutive windows along the time axis in the episode-recognizing algorithm for parallel episode α.

Serial episodes can be recognized by using automata. The algorithm constructs an automaton for each serial episode α every time the first event A_{first} of α comes into the window. The active state of an automaton reflects a prefix of α in the window. When the same state A_{first} of α leaves the window, the corresponding automaton is removed. When an automaton reaches its accepting state at time t, it means that the corresponding episode is entirely in the window. Since there could be multiple instances of the same automaton existing in a window, the starting time t is saved in $\alpha.inwindow$ if no other automata for

α are in the accepting state. When an automaton for α in the accepting state is removed and no other automata for α in the accepting state, the counter $\alpha.freq\_count$ is increased by the number of windows that α has been remained in the window.

2. COMPLEX EPISODE AND PARALLEL-OF-SERIAL EPISODE

2.1 Complex Episode

Using the algorithms described in Section 1.2: Sequence Data Mining, one can recognize all the frequent parallel episodes and serial episodes from a given event sequence. All parallel episode rules and all serial episode rules can be generated easily with certain confidence. However, the real world is much more complex, and representing groups of events with parallel-only episodes or serial-only episodes cannot always work for complex sequence data mining tasks.

In parallel episode mining, the partial order is trivial. A frequent parallel episode α means that a group of events happen together with respect to the frequency $fr(\alpha, s, w\_width)$ for a given event sequence s with window width w_width. Parallel episode mining can find more frequent episodes and discover more episode rules than serial episode mining. However, in sequence data mining, people usually want to find rules in which the relative event sequence is considered. The result of parallel episode mining cannot satisfy this requirement.

Serial episode puts rigid restriction on the episode configuration. The nodes in the graph have to be totally ordered. Using the rules obtained from serial episode mining, people can predict the coming events as well as the coming sequence accurately. Unfortunately, only a few of real world scenarios can be represented as serial episodes. Most of them are complex episodes and can be hierarchically divided into parallel episodes and serial episodes.

Complex episode can provide the power to represent more general real-world scenarios. For example, suppose that course A is the prerequisite of courses B and C, and both course B and C are the prerequisite of course D. There is no constraint between courses B and C. The partial order of the four courses can be explained by Figure 2-1.

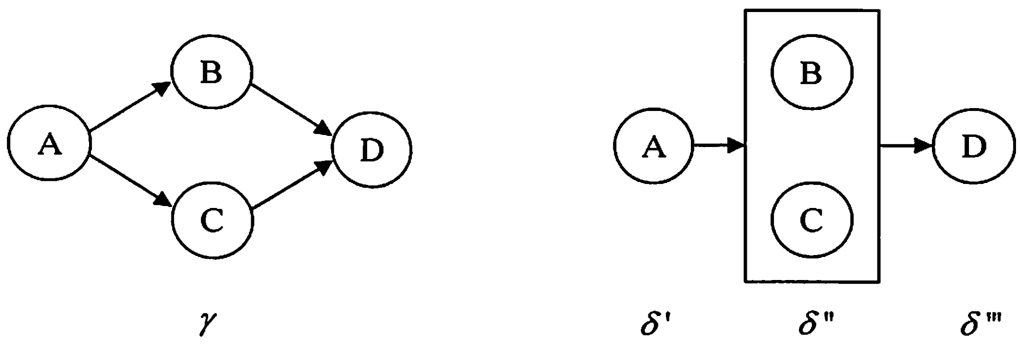

Figure 2-1. Complex episode γ and a serial combination of three episodes.

This is a complex episode and cannot be mined by parallel or serial episode data mining algorithm. The recognition of complex episodes can be reduced to the recognition of a hierarchical combination of serial and parallel episodes. For the example in Figure 2-1, episode γ can be considered a serial combination of three episodes: an episode δ' consisting of A alone, a parallel episode δ'' consisting of B and C, and an episode δ''' consisting of D alone.

2.2 Parallel-of-Serial Episode

Mining complex episodes is a nontrivial task; and no known algorithms are in the literature. A complex episode is an acyclic directed graph, which can be arbitrarily complex. Among these complex episodes, one type of complex episode is particularly useful to data mining users. This special complex episode can be defined as: an episode consists of two serial subepisodes that share the common start event and the end event, as shown in Figure 2-2.

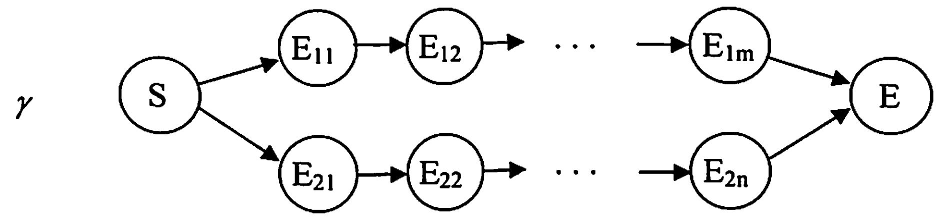

Figure 2-2. A PoS-episode γ.

We call this type of complex episode PoS-episode for it is a parallel connection of two serial episodes that connects two events together. A PoS-episode has *m*+*n*+2 nodes with $m \geq 1$, $n \geq 1$, and has at least four nodes when $m = n = 1$. We can divide the PoS-episode into two serial subepisodes, which are γ' and γ'', as depicted in Figure 2-3.

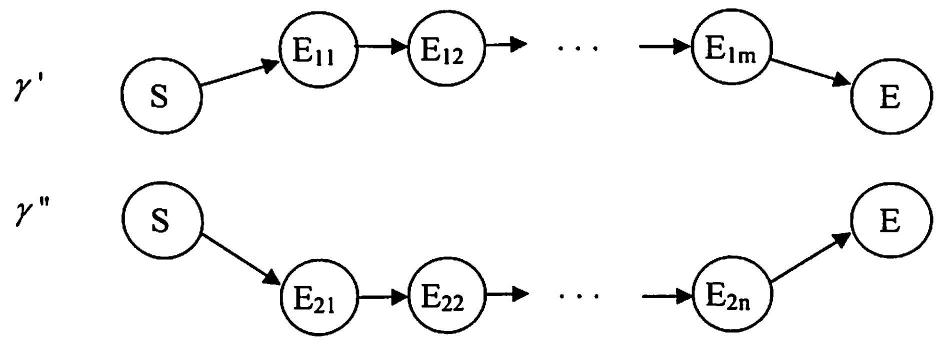

Figure 2-3. Decomposing a PoS-episode γ into two serial subepisodes γ' and γ''.

This work aims to solve the PoS-episode mining problem by developing algorithms that extend the previous work on parallel episode mining and serial episode mining.

3. ALGORITHMS

The PoS-episode mining algorithms consist of two parts, generation of candidate PoS-episodes and recognition of frequent PoS-episodes.

3.1 Generation of Candidate Parallel-of-Serial Episodes

As the output of the serial episode mining algorithm described in Section 1.2: Sequence Data Mining, the frequent serial episode array, *FSE*, is used in this work as the input of the PoS-episode candidate-generation algorithm.

The frequent serial episode array *FSE* is the array of all frequent serial episode sets: *FSE*[*i*] is the set of all frequent serial episodes that have the same size *i*, i.e., have the same number of events. The number of frequent serial episode sets in *FSE* and the number of episodes in *FSE*[*i*] are denoted by |*FSE*| and |*FSE*[*i*]|, respectively. The serial episodes of size *i* in *FSE*[*i*] are sorted in the lexicographical order, and *FSE*[*i*][*j*] gives the *j*th serial episode of size *i* in *FSE*[*i*].

A PoS-episode α is a parallel connection of two serial episodes that connects two events together. We label these two serial episodes as α_1 and α_2. Since we employ state-transition automata to recognize episodes, we use the term state interchangeable with the term event when the context is clear. The sizes of α_1 and α_2 can be different but each is at least 3. Since α_1 and α_2 share the same beginning state and the ending state, the size of α is α.size = α_1.size + α_2.size – 2, and the size of a PoS-episode is at least 4. A PoS-episode α remains the same if we interchange the serial episodes α_1 and α_2 in it. If we have $\alpha_1 = \alpha_2$, then α becomes a serial episode.

Algorithm 1 generates all possible candidate PoS-episodes α consisting of two serial episodes α_1 and α_2 with $\alpha_1 < \alpha_2$ in term of the lexicographic order. We assume that all the event types have been mapped into contiguous integer numbers beginning from 1. Thus, we use the event type as the index of arrays in our algorithm.

Algorithm 1. Generating all the possible PoS-episode using the frequent serial episode set.
Input: An array *FSE* of frequent serial episode set.
Output: An array *C* of all candidate PoS-episodes.
Method:

```
C := $\phi$;
for i := 3 to |FSE| do
   for j := 1 to |FSE[i]| do
      $\alpha$ := FSE[i][j];
      for m := i to |FSE| do                    //since $\alpha < \beta$
         for n := 1 to |FSE[m]| do
            $\beta$ := FSE[m][n];
            if( $\beta \leq \alpha$ ) then
               continue;
            if( $\alpha[1] = \beta[1]$ and $\alpha[\alpha.size] = \beta[\beta.size]$ ) then
               construct candidate PoS-episode $\gamma$ based on $\alpha$ and $\beta$;
               C := C $\cup$ { $\gamma$ };
output C.
```

According to the Apriori property that all nonempty subsets of a frequent set must also be frequent, the two subepisodes α_1 and α_2 must also be frequent serial episodes for any given α that is frequent.

Since *FSE* is the complete set of frequent serial episodes, Algorithm 1 can explore all possible combinations of the serial episode pairs and obtain the complete candidate PoS-episode set. Also, because *FSE* consists of only frequent serial episodes, the candidate PoS-episode set generated by Algorithm 1 is a small set since it utilizes the known frequent serial episode information. This can greatly reduce the search space when detecting the frequent PoS-episodes.

3.2 Recognition of Frequent Parallel-of-Serial Episodes

Frequent PoS-episodes can be recognized by constructing two deterministic finite automata M_1 and M_2 for each candidate PoS-episode α as shown in Figure 3-1: M_1 and M_2 correspond to α_1 and α_2 of the candidate PoS-episode α.

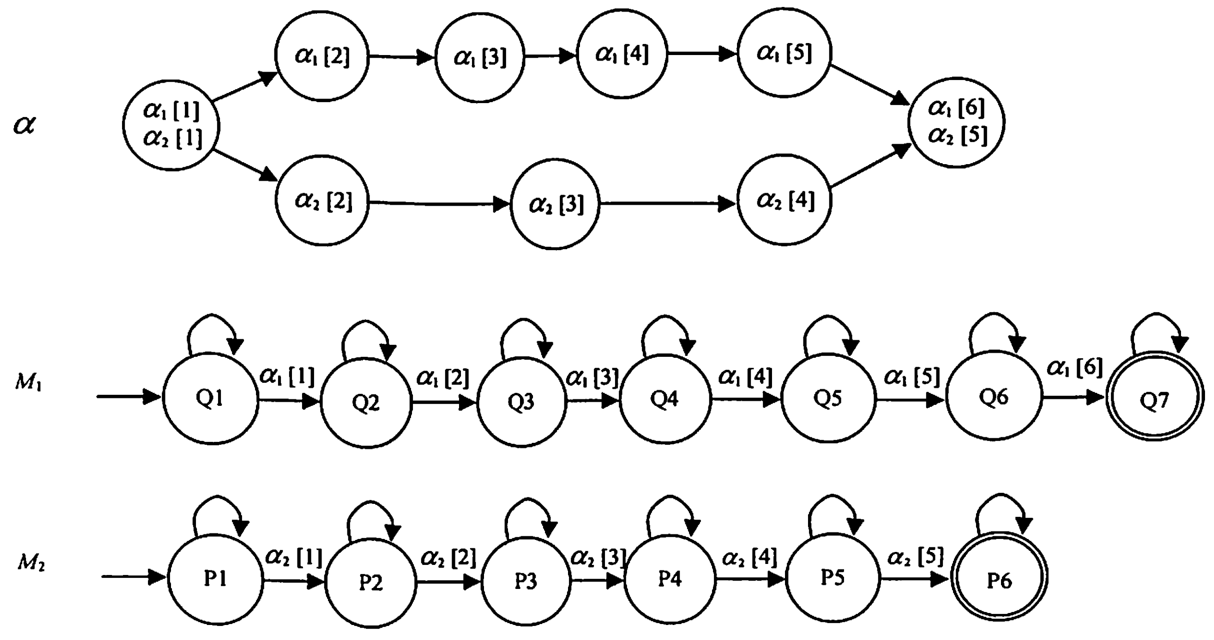

Figure 3-1. Construct two deterministic finite automata M_1 and M_2 for PoS-episode α .

When symbols, i.e., event types are fed in, both M_1 and M_2 will do state transitions according to the input event types. A PoS-episode α is recognized when both M_1 and M_2 are in the accepting states. During the recognition, a sliding window with width *w_width* is advancing along the time axis. For the event sequence $<(A_1, t_1), (A_2, t_2), \ldots, (A_n, t_n)>$, the timestamp of the first sliding window is t_1 and the timestamp of the last sliding window is $t_n + w\_width - 1$. Each moving represents the passing of one time unit and will bring in zero or one event (A, t), where t is the timestamp of current window.

Algorithm 2 recognizes all of the frequent PoS-episodes from the given event sequence, window size, and the minimum frequency threshold. This algorithm initializes a pair of instances of automata (M_1, M_2) for each PoS-episode α whose first event type is the same as the event that comes into the sliding window. The automata M_1 and M_2 are equivalent to the subepisodes α_1 and α_2 of PoS-episode α. The instances of automata pair (M_1, M_2) will be removed when the same event falls out of the sliding window. A PoS-episode α is completely enclosed in the window if both M_1 and M_2 of α reached their accepting states. We call the earliest time that α is enclosed in the sliding window the entrance time of α. The entrance time is saved into variable α *.inwindow*. When an automata pair in the accepting states is removed, we calculate the window number that the PoS-episode α stayed in the sliding window and save it into α *.freq_count*.

There may be multiple instances of automata of PoS-episode α that exist in the sliding window. One or more of them may be in the accepting state. However, according to the frequency definition in Chapter 1, only the number of windows that contain the PoS-episode will be counted. In other words, for a certain window, it is important that PoS-episode α is present or not, while the number of α is not important. Based on this analysis, the entrance time of α is defined as the time that the instance of automata of α reaches the accepting states and no other instances of α in the window are in the accepting states. The removal time of automata of α in the accepting states is defined as the time that the automata of α are removed and no other instance of the automata in the accepting states.

We use a two-dimensional array *waits*[*i*][*j*] to keep track of the automata pair (M_1, M_2) that accept event type *i* in M_1 and *j* in M_2. In the algorithm, symbols α_1 and α_2 are used interchangeably with M_1 and M_2. The automata pair (M_1, M_2) is preserved in the linked list. The cell *waits*[*i*][*j*] is the head of the linked list. The automata pair (M_1, M_2) is represented in the form (α, *i*, *j*, *T*, *w*) where α is the PoS-episode and the data structure of α preserves the states and transition information, *i* and *j* are the next states in M_1 and M_2, *T* is the time when the first state of α enters the window, and *w* is the description of the last transitions of the automata pair (M_1, M_2) as shown in Table 3-1.

Table 3-1. Description of the last transitions of automata pair (M_1, M_2).

w	0	1	2	3
Description of the last transitions of automata pair (M_1, M_2).	M_1 moved forward one state; M_2 moved forward one state.	M_1 moved forward one state; M_2 remained in the same state.	M_2 moved forward one state; M_1 remained in the same state.	Both M_1 and M_2 remained in the same states.

All the automata pairs that are initialized at time *T* are linked at *beginset*[*T*]. If the automata pair is removed, then it is removed from this linked list. Unlike the automata pair stored in the *waits*[*i*][*j*] that *i* and *j* are the next states for transitions, the automata pair (α , i, j, T, w) stored in *beginset*[T] give the states i and j on which transitions just happened.

For each input symbol, all the transitions are organized in a linked list *transitions*. Automata stored in *transitions* are in the form of (α , i, j, T, A) where α , i, j, and T have the same meanings as *beginset*[T] and A is the input symbol, i.e., current processing event.

Algorithm 2. Recognizing all of the frequent PoS-episodes from the given event sequence, window size, and the minimum frequency threshold.
Input: number of event type *ET*, candidate PoS-episode array *C*, an event sequence $s = (s, T_s, T_e)$, a window width *w_width*, and a frequency threshold *min_fr*.
Output: a set of frequent PoS-episodes *FPoS*.
Method:

```
// initialization
for i := 1 to ET
   for j := 1 to ET
      $waits[i][j] := \phi$;
for each $\alpha \in C$ do
   $waits[\alpha_1[1]][\alpha_2[1]] := waits[\alpha_1[1]][\alpha_2[1]] \cup \{(\alpha, 1, 1, -1, -1)\}$;
   $\alpha.freq\_count := 0$;
   $\alpha.inwindow := -\infty$;
// recognition
for start := $T_s$ - w_width +1 to $T_e$ do
   t := start + w_width - 1;
   $beginset[t] := \phi$;
   $transitions := \phi$;
   for all events $(A, t') \in s$ such that $t' = t$ do
      for all $(\alpha, i, j, T, w) \in waits[A][*] \cup waits[*][A]$ do
         if $i = |\alpha_1|$ and $j = |\alpha_2|$ and $\alpha.inwindow = -\infty$ then
            $\alpha.inwindow = start$;
         if $i = j = 1$ then
            $transitions := transitions \cup \{(\alpha, 1, 1, t, A)\}$;
         else
            $transitions := transitions \cup \{(\alpha, i, j, T, A)\}$;
            if $w = 0$ then
               $beginset[T] := beginset[T] \setminus \{(\alpha, i\text{-}1, j\text{-}1)\}$;
            else if $w = 1$ then
               $beginset[T] := beginset[T] \setminus \{(\alpha, i\text{-}1, j)\}$;
            else if $w = 2$ then
               $beginset[T] := beginset[T] \setminus \{(\alpha, i, j\text{-}1)\}$;
            else if $w = 3$ then
               $beginset[T] := beginset[T] \setminus \{(\alpha, i, j)\}$;
            $waits[\alpha_1[i]][\alpha_2[j]] := waits[\alpha_1[i]][\alpha_2[j]] \setminus \{(\alpha, i, j, T, w)\}$;
```

```
    for all ($\alpha$ , $i, j, T, w$) $\in$ transitions do
      if $w = \alpha_1[i]$ and $w = \alpha_2[j]$ then
        if $i< \alpha_1$.size and $j< \alpha_2$.size then
          waits[$\alpha_1[i+1]$] [$\alpha_2[j+1]$] := waits[$\alpha_1[i+1]$] [$\alpha_2[j+1]$] $\cup$ {($\alpha$ , $i+1, j+1, T$, 0)}; $w'$:=0;
        else if $i< \alpha_1$.size then
          waits[$\alpha_1[i+1]$] [$\alpha_2[j]$] := waits[$\alpha_1[i+1]$] [$\alpha_2[j]$] $\cup$ {($\alpha$ , $i+1, j, T$, 1)}; $w'$:=1;
        else if $j< \alpha_2$.size then
          waits[$\alpha_1[i]$] [$\alpha_2[j+1]$] := waits[$\alpha_1[i]$] [$\alpha_2[j+1]$] $\cup$ {($\alpha$ , $i+1, j, T$, 2)}; $w'$:=2;
      else if $w = \alpha_1[i]$ then
        if $i< \alpha_1$.size then
          waits[$\alpha_1[i+1]$] [$\alpha_2[j]$] := waits[$\alpha_1[i+1]$] [$\alpha_2[j]$] $\cup$ {($\alpha$ , $i+1, j, T$, 1)}; $w'$:=1;
        else
          waits[$\alpha_1[i]$] [$\alpha_2[j]$] := waits[$\alpha_1[i]$] [$\alpha_2[j]$] $\cup$ {($\alpha$ , $i, j, T$, 3)}; $w'$:=3;
      else if $w = \alpha_2[j]$ then
        if $j< \alpha_2$.size then
          waits[$\alpha_1[i]$] [$\alpha_2[j+1]$] := waits[$\alpha_1[i]$] [$\alpha_2[j+1]$] $\cup$ {($\alpha$ , $i, j+1, T$, 2)}; $w'$:=2;
        else
          waits[$\alpha_1[i]$] [$\alpha_2[j]$] := waits[$\alpha_1[i]$] [$\alpha_2[j]$] $\cup$ {($\alpha$ , $i, j, T$, 3)}; $w'$:=3;
      beginset[$T$] := beginset[$T$] $\cup$ {($\alpha$ , $i, j, T, w'$)};
    for all ($\alpha$ , $i, j, T, w$) $\in$ beginset[start-1] do
      if $i=\alpha_1$.size and $j=\alpha_2$.size then
        if no other $\alpha$ in current sliding window in the accepting state then
          $\alpha$ .freq_count = $\alpha$ .freq_count - $\alpha$ .inwindow + start;
          $\alpha$ .inwindow = $-\infty$ ;
        else
          continue;
      else if $w$=0 then
        waits[$\alpha_1[i+1]$] [$\alpha_2[j+1]$] := waits[$\alpha_1[i+1]$] [$\alpha_2[j+1]$] \ {($\alpha$ , $i+1, j+1, T$)};
      else if $w$=1 then
        waits[$\alpha_1[i+1]$] [$\alpha_2[j]$] := waits[$\alpha_1[i+1]$] [$\alpha_2[j]$] \ {($\alpha$ , $i+1, j, T$)};
      else if $w$=2 then
        waits[$\alpha_1[i]$] [$\alpha_2[j+1]$] := waits[$\alpha_1[i]$] [$\alpha_2[j+1]$] \ {($\alpha$ , $i, j+1, T$)};
      else if $w$=3 then
        waits[$\alpha_1[i]$] [$\alpha_2[j]$] := waits[$\alpha_1[i]$] [$\alpha_2[j]$] \ {($\alpha$ , $i, j, T$)};
  FPoS := $\phi$ ;
  for all PoS-episode s $\alpha$ $\in C$ do
    if $\alpha$ .freq_count / (Te – Ts + w_width) $\geq$ min_fr then
      FPoS := FPoS $\cup$ {$\alpha$ };
```

4. EXPERIMENTS

In this chapter, the PoS-episode mining algorithms are studied by carrying out web page traversal pattern mining using web server log data set. When a user is visiting a website, each visit of a web page can be viewed as an event, in which the web page name is the event type and the visiting time is the event time. Thus, a website access event E has the form of (web page name, visiting time). All the events in a web server log file constitute of an event sequence s.

4.1 A Simple Example on Parallel-of-Serial Episode Mining

To show how the PoS-episode mining algorithms work and explain that it is a useful extension of serial episode mining, a simple example is given below.

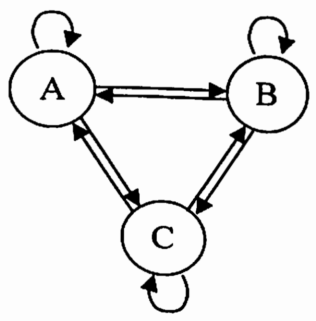

Figure 4-1. Structure of a three-page website.

Suppose that we have a website that has only three web pages, namely, A, B, and C. Each page has links to other two pages and itself. Figure 4-1 illustrates the structure of the website. The default entrance of the website is page A. From page A, a user can browse any pages in any sequence.

While a user is visiting the website, the web server records the online activities in a log file. Each entry of the log file includes a page name and the time of the request. Usually, the user activities are different from websites to websites. A website like google.com serves users all over the world and has the workload equally distributed in a day. While a website that serves only the local users may have the peak load time interlaced with a long silence period usually happened during the night. Figure 4-2 shows the web server log of the three-page website.

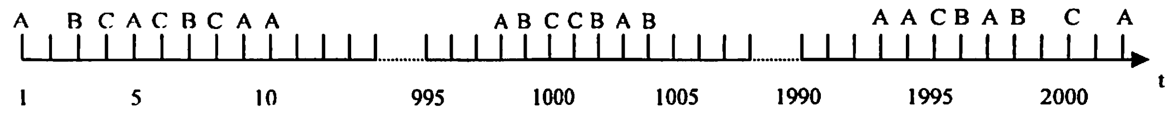

Figure 4-2. The web log of a three-page website.

To find the user visiting patterns, we can run our PoS-episode mining algorithm as well as serial episode mining algorithm on the web log data set. We set the sliding window size to 10 and the minimum frequency ratio to 1%. Table 4-1 gives the mining result of the serial episode mining.

There is no frequent serial episode with size greater than 3 that can be found using the serial episode mining algorithm. From the frequent serial episodes given in the Table 4-1, we can get the candidate serial episodes of size 4. Table 4-2 shows the candidate serial episodes of size 4 and the frequency.

Table 4-1. Serial episode mining result of a three-page website.

Frequent Serial Episode	Frequency Count	Frequency Ratio	Frequent Serial Episode	Frequency Count	Frequency Ratio
A	53	0.026	B	41	0.020
C	40	0.019	AA	32	0.015
AB	37	0.018	AC	31	0.015
BA	32	0.015	BB	24	0.011
BC	30	0.014	CA	34	0.016
CB	27	0.013	CC	24	0.011
ABA	25	0.012	ABC	26	0.012
ACA	25	0.012	ACB	22	0.010
BAB	22	0.010	BCA	24	0.011
CAB	21	0.010	CBA	23	0.011

Table 4-2. Candidate serial episodes of size 4.

Serial Episode	Frequency Count	Frequency Ratio	Serial Episode	Frequency Count	Frequency Ratio
ABCA	20	0.0099	ACBA	18	0.0089

Because the minimum frequency ratio was set at 1%, these two serial episodes are eliminated from the final mining output. Then we run our PoS-episode mining algorithm on the result of the serial episode mining. The candidate PoS-episodes are generated from the frequent serial episodes by selecting frequent serial episode pair (α , β) that has: α *.size*$\geq$3, β *.size*$\geq$3; α *.firstevent* = β *.firstevent*; *and* α *.lastevent* = β *. lastevent.*

The only candidate PoS-episode generated by the PoS mining algorithm is $A\overset{B}{\underset{C}{\_}}A$ and it is the combination of the frequent serial episodes ABA and ACA. The PoS-episode mining result is listed in Table 4-3.

Although no frequent serial episode can be found for size 4, we discover one frequent PoS-episode of size 4. This PoS-episode can also provide the sequence information of events as the serial episodes do. From the result, we can get such information on the web page visiting pattern as: if the current visiting page is A, then in the next 10 time units, we can predict that page sequence B, C and A, or page sequence C, B and A will be visited in the order in terms of the minimum frequency.

Table 4-3. The PoS-episode mining result of a three-page website.

Frequent PoS-episode	Frequency Count	Frequency Ratio
$A\overset{B}{\underset{C}{\_}}A$	25	0.012

4.2 Parallel-of-Serial Episode Mining on a Large Web Log Data Set

The experiment data set is the website (cybermath.okstate.edu) access log data. This website is a remote education website which belongs to College of Arts and Sciences of Oklahoma State University. It provides college-level calculus courses to high school students all over the country. Students use this website to view lessons, take quizzes, and communicate with other users. The basic information of the website and log data set is shown in Table 4-4.

Table 4-4. Basic information of the experiment data set.

Website Name	cybermath.okstate.edu
Individual Web Pages Number	304
Log Starting Time	08/Jan/2003:09:09:03
Log Ending Time	28/Feb/2003:19:18:22
Total Log Entries	44955
Effective Web Pages Entries	19710
Average Visiting Length	493 seconds
Average Web Page Visiting Length	44 seconds

In this experiment, the purpose of data mining is to discover the web page traversal patterns. In the website, each web page has more than one links that point to other web pages and may even have links point back to itself. If we treat each web page as a node and each link as an edge, then we can view the web pages in a website as a complex graph. From one node to another node, there could be many ways to move along. To improve the website design or to efficiently put advertisements on the right web pages, the frequent web page traversal patterns are needed.

Before we input the log data to the data mining algorithm, data need to be cleaned first. The web log includes two categories of entries, effective web page entries and associated web entries. An effective web page is the web page that is opened by users with intent. Usually an effective web page name has extensions like html, htm, asp, and php. An associated web entry records the object that is opened by the effective web page while it is visited. Most of the associated web entries are the images embedded in the web pages or the cascading style sheet files. The associated web entries are not helpful to the mining of website usage patterns and need to be cut off before data mining.

To distinguish the frequent episodes, i.e., web page traversal patterns from the low frequent ones, users need to give criteria to decide whether an episode is frequent or not. Two parameters can be used for this purpose, one is frequency count and the other is frequency ratio. Frequency count is the number of the sliding windows that contain the target episode. It would be more visualized if we know the total sliding windows in advance. However this count can be changed from case to case and is not convenient for comparing the results of data mining with each other. Frequency ratio is the fraction of the number of the sliding windows that contain the target episode over the total sliding windows. Frequency ratio is easier to use in episode mining than frequency count and the results can be compared among different cases. We use frequency ratio in this work.

One of the characters of the web log data is the sparseness of the visit events compared to the total available timestamps. During the night, there are long gaps of silence without users' visit. This leads to a very small frequency ratio to be set. On the other hand, the algorithms iterate on every sliding window. In the web server log, there could be an event in any second. So, the step for the sliding windows is one second. The sparseness of the event can slow down the computation time dramatically. Thus, condensing of web log data is needed. The data condensing process should not introduce new relationship among any events. This means that any two adjacent events with a gap width $w_{gap} > w\_width+1$ should be moved together and leave a gap width of $w\_width+1$ between them. All other relative distances between events are preserved during the condensing process. Figure 4-3 shows the condensed result of Figure 4-2.

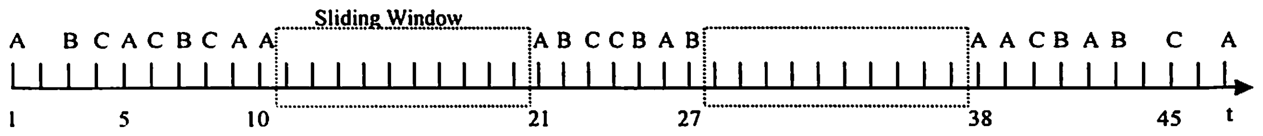

Figure 4-3. Condensing of web log data.

We can see from Table 4-4 that the average website visit length and the average web page visit length are 493 seconds and 44 seconds respectively. The experimental window sizes are chosen based on these values and are listed in the Table 4-5. The frequency ratios for each window size are decided by making the algorithms generate suitable number of output frequent PoS-episodes since too many or too few output of frequent PoS-episodes are not desirable. In this work, we use the output frequent PoS-episodes number range of 10 – 1000 to decide the frequency ratios. In the experiment, we first guessed a frequency ratio and used it to get the frequent PoS-episodes number under this ratio. Then we gradually increased or decreased this ratio to let the output frequent PoS-episodes number fall into the desired range. The frequency ratio and the frequent PoS-episodes numbers obtained from the experiments are listed in Figure 4-5.

Table 4-5. Frequent PoS-episodes number in different combination of window width and frequency ratio.

Window Width (seconds)	Frequency Ratio (%)															
	0.8	1	1.5	2	3	4	5	6	7	8	9	10	12	15	18	20
30	887	639	280	171	45		4									
50				812	311	144	38	13	6							
100						813	532	214	136	13	12	10				
200								870	668	534	259	143	36	8		
300										877		552	170	30		3
400												717	544	133	22	8
500												1059	638	179	44	19

By running the PoS algorithms on the parameters described in Table 4-5, we can get all the candidate episode data and the frequent episode data for both serial episodes and PoS-episodes. For each category, the candidate episodes are generated first and then the frequent episodes are recognized by the database passes. In Table 4-6, these values are listed out.

The ratio of the candidate episode number to the frequent episode number can provide how efficient the candidate-generation algorithms are. The larger the ratio is, the more efficient the algorithms are. From Figure 4-4, we can see that the PoS candidate-generation algorithm has a high ratio and the efficiency is high. Only about 10% of the candidates turn out to be not frequent.

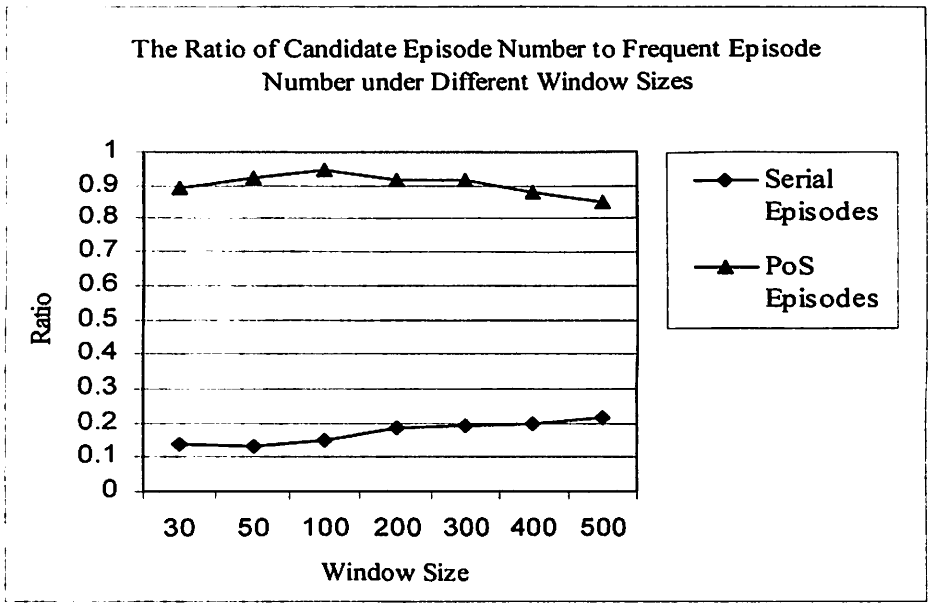

Figure 4-4. Ratio of candidate and frequent episode numbers under different window size.

Among the discovered frequent PoS-episodes, there are two categories of PoS-episodes. We name one the fresh PoS-episode and the other the associating PoS-episode. A PoS-episode α is an associating PoS-episode if there is at least one input string on event type set that makes the automata pair (M_1, M_2) for α to reach their accepting states and also makes at least one frequent serial episode based automaton M_s reach the accepting state. We say that these frequent serial episodes are associated with the PoS-episode α .

Table 4-6. Detailed result of PoS-episode mining.

Win. Size	Min Freq. Ratio	Candi. PoS-episode #	Freq. PoS-episode #	Fresh PoS-episode #	F_{fresh} (%)	Win. Size	Min Freq. Ratio	Candi. Serial Episode #	Freq. Serial Episode #
30	0.008	1004	887	27	3.0	30	0.008	2222	373
30	0.01	723	639	73	11.4	30	0.01	1896	302
30	0.015	301	280	4	1.4	30	0.015	1383	201
30	0.02	186	171	20	11.7	30	0.02	1321	144
30	0.03	49	45	3	6.7	30	0.03	992	91
30	0.05	5	4	1		30	0.05	462	31
50	0.02	847	812	10	1.2	50	0.02	1440	260
50	0.03	354	311	4	1.3	50	0.03	1232	171
50	0.04	165	144	25	17.4	50	0.04	964	110
50	0.05	45	38	3	7.9	50	0.05	759	76
50	0.06	16	13	1	7.7	50	0.06	483	44
50	0.07	10	6	1	16.7	50	0.07	449	35
100	0.04	832	813	5	0.6	100	0.04	1281	218
100	0.05	559	532	61	11.5	100	0.05	1020	172
100	0.06	236	214	21	9.8	100	0.06	890	131
100	0.07	159	136	20	14.7	100	0.07	521	88
100	0.1	14	10	1	10.0	100	0.08	478	66
100	0.08	48	42	0	0.0	100	0.09	442	49
100	0.09	25	25	0	0.0	100	0.1	433	38
200	0.06	952	870	22	2.5	200	0.06	1078	219
200	0.07	726	668	54	8.1	200	0.07	991	183
200	0.08	559	534	63	11.8	200	0.08	658	152
200	0.09	291	259	33	12.7	200	0.09	557	126
200	0.1	168	143	23	16.1	200	0.1	511	95
200	0.12	42	36	3	8.3	200	0.12	456	61
200	0.15	12	8	1	12.5	200	0.15	395	34
300	0.08	967	877	27	3.1	300	0.08	892	204
300	0.1	577	552	68	12.3	300	0.1	608	152
300	0.12	201	170	28	16.5	300	0.12	496	103
300	0.15	33	30	3	10.0	300	0.15	449	55
300	0.2	4	3	0	0.0	300	0.2	370	24
400	0.1	848	717	71	9.9	400	0.1	705	186
400	0.12	567	544	65	11.9	400	0.12	562	150
400	0.15	166	133	23	17.3	400	0.15	480	86
400	0.18	23	22	2	9.1	400	0.18	405	45
400	0.2	12	8	1	12.5	400	0.2	393	33
500	0.1	1212	1059	53	5.0	500	0.1	797	219
500	0.12	774	638	69	10.8	500	0.12	634	169
500	0.15	213	179	26	14.5	500	0.15	503	108
500	0.18	58	44	6	13.6	500	0.18	443	65
500	0.2	20	19	6	31.6	500	0.2	405	44

Apparently, there could be one or more serial episodes associated with a PoS-episode. A PoS-episode is a fresh PoS-episode if it is not an associating PoS-episode. An associating PoS-episode α represents a class of serial episodes that accept the same inputs with α. A fresh PoS-episode is a new discovery of the PoS-episode mining algorithm and no frequent serial episode is associated with it. All the associated serial episodes of a fresh PoS-episode are eliminated from the output of serial episode mining because their frequencies are lower than the minimum frequency threshold.

We can use the fraction of fresh PoS-episode F_{fresh} and the fraction of associating PoS-episode $F_{\text{associating}}$ to evaluate the performance of the parameter settings for serial episode mining algorithm and PoS-episode mining:

$$F_{\text{fresh}} = \frac{\text{number of fresh PoS-episodes}}{\text{number of all freqnent PoS-episodes}},$$

$$F_{\text{associating}} = \frac{\text{number of associating PoS-episodes}}{\text{number of all freqnent PoS-episodes}}.$$

A high fraction of fresh PoS-episode F_{fresh} means that the PoS-episode mining algorithm picks up a large number of frequent PoS-episodes that have been missed by the serial episode mining algorithm. Table 4-6 lists the numbers of fresh PoS-episode and the values of F_{fresh}. We can see from the Table 4-6 that for a specific window size, the number of frequent PoS-episodes decreases while the minimum frequency ratio increases. However, the F_{fresh} increases first, then reaches a maximum value, and then begins to decrease. We can easily discover that when F_{fresh} reaches its maximum value, the number of fresh PoS-episodes and the number of frequent serial episodes reach reasonable values for data mining users who will eventually examine the result manually. If we plot out these window sizes and minimum frequency ratios that corresponding to the maximum values of F_{fresh}, we can find that they are almost in a linear relation as shown in Figure 4-5. This can give the PoS-episode mining a guide on how to choose the mining parameters.

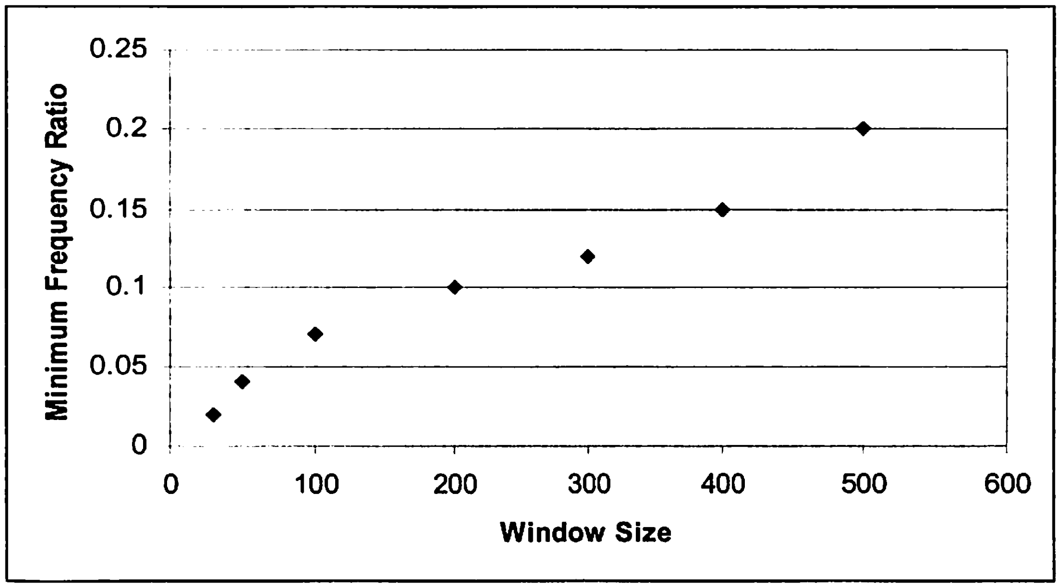

Figure 4-5. A linear relation between window sizes and minimum frequency ratios when F_{fresh} reaches its maximum value.

For an associating PoS-episode, its sliding window count $C_{\text{associating}}$ has such relationship with the count of the associated serial episodes $C_{\text{associated}}$ as the following equation:

$$\sum_i C^i_{\text{associated}} \geq C_{\text{associating}} \geq \max_i(C^i_{\text{associated}}).$$

We define the variable $R_{\text{associating}}$ as the ratio of the maximum value of the $C_{\text{associated}}$ to the $C_{\text{associating}}$:

$$R_{\text{associating}} = \frac{\max_i(C^i_{\text{associated}})}{C_{\text{associating}}},$$

which can be used to evaluate the discreteness of the associated serial episodes of an associating PoS-episode . If $R_{\text{associating}} = 1.0$, it means that there is only one serial episode associated with the PoS-episode. A low value of $R_{\text{associating}}$ gives a hint that there could be several associated serial episodes and each of them has a low frequency. The data for evaluating the discreteness of this web log episode mining are listed in Table 4-7.

Table 4-7. Evaluate the discreteness of this web log episode mining.

Window size	Sum of the maximum value of the $C_{\text{associated}}$	Sum of the $C_{\text{associating}}$	$R_{\text{associating}}$
30	7202124	7414343	0.97
50	12711858	13043078	0.97
100	43591203	44569178	0.98
200	142462501	145425962	0.98
300	143962251	147034334	0.98
400	171119403	174538600	0.98
500	286804882	292511138	0.98

We can see that the values of $R_{\text{associating}}$ for all the window sizes are almost 1.0. For this reason, the PoS-episode mining tasks can ignore the associating PoS-episodes and focus on the fresh PoS-episodes mining only while dealing with the similar data that is analyzed in this work.

5. CONCLUSION

This work extended the existing parallel and serial episode mining algorithms by developing the PoS-episode mining algorithms. The existing parallel and serial episodes either lack the ability to preserve the sequence information or are too rigid to express more general real world scenarios. The PoS-episodes can model more general situations and preserve the sequence information as well. The PoS-episode mining algorithms can provide episode mining users a powerful mining tool and make the episode mining more flexible.

The PoS-episode mining algorithms use the serial episode mining output as the input. There are two parts of the PoS-episode mining algorithms, candidate PoS-episode generation algorithm and frequent PoS-episode recognition algorithm. Candidate PoS-episodes are generated by analyzing the frequent serial episode set. Frequent PoS-episode recognition is carried out by using automata. While sliding the window along the time axis, the algorithm generates and maintains multiple instances for each possible automaton to recognize and count each candidate PoS-episode. After users input the algorithm a suitable window width and a minimum frequency ratio, they will get frequent PoS-episode set as the output.

To use the PoS-episode mining algorithm, users need to decide reasonable parameters like window width and minimum frequency ratio. As examples of how to analyze the real PoS-episode mining case and how to decide reasonable parameters, experiments are performed in this work by studying the web server log data set and mining the web page traversal patterns. Concepts and methods are provided for this specific mining case to evaluate the mining process. A linear relation between window sizes and minimum frequency ratios is found and can be used as a guide when further PoS-episode mining tasks are carried out for the similar cases.

REFERENCES

[1] Agrawal, R., Imielinski, T., and Swami, A. Mining association rules between sets of items in large databases. *In Proceedings of the 1993 Association for Computing Machinery Special Interest Group on Management Of Data International Conference on Management of Data. Washington, D.C., pages 207-216, 1993.*

[2] Agrawal, R. and Srikant, R. Fast algorithms for mining association rules. *In Proceedings of 20th International Conference on Very Large Data Bases, pages 487-499, 1994.*

[3] Agrawal, R. and Srikant, R. Mining sequential patterns. *In Proceedings of the Eleventh International Conference on Data Engineering, Taipei, Taiwan, pages 3–14, 1995.*

[4] Silberschatz, A. and Tuzhilin, A. What makes patterns interesting in knowledge discovery systems. *IEEE Transactions on Knowledge and Data Engineering, volume 8, number 6, pages 970-974, 1996.*

[5] Mannila, H., Toivonen, H., and Verkamo, A. I. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery, volume 1, number 3, pages 259-289, 1997.*

[6] Han, J., Pei, J., and Yin, Y. Mining frequent patterns without candidate generation. *In Proceedings of Association for Computing Machinery Special Interest Group on Management of Data, pages 1-12, 2000.*

[7] Pei, J. and Han, J. Can we push more constraints into frequent pattern mining? *In Proceedings of the Sixth Association for Computing Machinery Special Interest Group on Knowledge Discovery and Data mining International Conference on Knowledge Discovery and Data Mining, pages 350-354, 2000.*

[8] Pei, J., Han, J., and Wang, W. Mining sequential patterns with constraints in large databases. *The Association for Computing Machinery Conference on Information and Knowledge Management 2002, pages 18-25.*

APPENDIX

The programming language used to implement the PoS episode mining algorithms and the serial episode mining algorithms is java. The program runs on the csa machine at the Computer Science Department of Oklahoma State University. There are totally 21 java classes with the Thesis.java as the main program. The source codes are listed below with the Thesis.java at the leading position and others in the lexicographical order according to the class names. Also, a screen copy of the file list is provided for reference.

```
$ ls -al *.java
-rw-r--r--  1 guohuan  guohuan   2279 Mar 16 16:34 PoSECG.java
-rw-r--r--  1 guohuan  guohuan   6456 Mar 29 22:54 PoSER.java
-rw-r--r--  1 guohuan  guohuan   2646 Mar 19 07:43 PoSEpisode.java
-rw-r--r--  1 guohuan  guohuan   2432 Mar 15 23:12 PoSEpisodePool.java
-rw-r--r--  1 guohuan  guohuan   3574 Mar 29 22:59 SECG.java
-rw-r--r--  1 guohuan  guohuan   6098 Mar 29 22:56 SER.java
-rw-r--r--  1 guohuan  guohuan   5394 Apr  1 15:17 Thesis.java
-rw-r--r--  1 guohuan  guohuan   2640 Mar 29 22:05 analysis.java
-rw-r--r--  1 guohuan  guohuan   4386 Mar 29 20:27 beginsetPoSE.java
-rw-r--r--  1 guohuan  guohuan   4385 Mar 29 22:09 beginsetSE.java
-rw-r--r--  1 guohuan  guohuan   1332 Mar  4 00:06 eventPool.java
-rw-r--r--  1 guohuan  guohuan   1575 Mar 12 16:02 eventSequence.java
-rw-r--r--  1 guohuan  guohuan   1691 Mar  4 16:20 frequentSE.java
-rw-r--r--  1 guohuan  guohuan   2074 Mar 19 07:38 instancePoSEpisode.java
-rw-r--r--  1 guohuan  guohuan   1504 Mar 11 17:13 instanceSerialEpisode.java
-rw-r--r--  1 guohuan  guohuan   2553 Mar  9 16:14 serialEpisode.java
-rw-r--r--  1 guohuan  guohuan   4126 Mar  6 15:53 serialEpisodePool.java
-rw-r--r--  1 guohuan  guohuan    862 Mar  4 11:40 theEvent.java
-rw-r--r--  1 guohuan  guohuan   2039 Mar 17 15:00 transitionsPoSE.java
-rw-r--r--  1 guohuan  guohuan   1624 Mar 12 11:11 transitionsSE.java
-rw-r--r--  1 guohuan  guohuan   5087 Mar 22 05:18 waitsPoS.java
```

```
//-----------------------------------------------------------
//Name:Guohuan Wang
//Course:Master Thesis
//Instructor:Dr. Dai
//-----------------------------------------------------------
//To compile: javac Thesis.java
//To run: java Thesis input_file_name w_width min_fr >output_file_name
//-----------------------------------------------------------
```

```
//----------------------------------------------------------
//Class name:Thesis
//Purpose: main program of Guohuan Wang's Master Thesis.
//----------------------------------------------------------

import java.io.*;
import java.util.StringTokenizer;
import java.text.DecimalFormat;
public class Thesis
{
 public static void main(String [] args)
  {
   File inFiletmp = new File (args[0]);    //check if the input file exist.
   BufferedReader inFile;
   PrintWriter outfileBF = null;
   StringTokenizer tokenizer;
   DecimalFormat myFormatter = new DecimalFormat("#0.000");
   eventPool EvnPool = new eventPool(306);  //maximum 30 different event types
   eventSequence EvtSeq = null;

   //--------------------------------------------------------------------//
   //Task 1: Open input file, that is the output of class ESG, fill Event   //
   //Type Pool(EvnPool: class eventPool) and form Event Sequence           //
   //(EvtSeq: class eventSequence).                         //
   //--------------------------------------------------------------------//
   if(!inFiletmp.exists())
   {
    System.out.println("Input file "+args[0]+" is missing!");
    return;
   }

   try
   {
    //open input file, read data and initial variables.
    inFile = new BufferedReader (new FileReader (args[0]));
                    String line;
                    line = inFile.readLine();

                    //generate event Pool, using input file data
                    while(line != null && line.compareTo("Event Type")!=0)   //throw away the leading
lines until line "Event Type"
                     line = inFile.readLine();
                    line = inFile.readLine(); //event type line
                    if(line != null)
                    {
                     tokenizer = new StringTokenizer (line);
     while(tokenizer.hasMoreTokens())
                     EvnPool.setEventType(tokenizer.nextToken());
    }
//     EvnPool.print();

    //generate event sequence, using input file data
    line = inFile.readLine();
                    while(line != null && line.compareTo("Event Sequence")!=0)   //throw away the leading
lines until line "Event Sequence"
```

```
                    line = inFile.readLine();
                   line = inFile.readLine(); //total events #
                   EvtSeq = new  eventSequence(Integer.parseInt(line));
                   line = inFile.readLine(); //first line of event data
                   while(line != null)
                   {
                    tokenizer = new StringTokenizer (line);
                    int tmp_e,tmp_t;
                    tmp_t = Integer.parseInt(tokenizer.nextToken());
                    tmp_e = EvnPool.getEvent(tokenizer.nextToken());
                    theEvent MyEvent = new theEvent(tmp_e,tmp_t);
                    EvtSeq.setEvent(MyEvent);
      line = inFile.readLine();
     }
     //EvtSeq.print(EvnPool);
    }
    catch (IOException e)          //handle error.
    {
     System.out.println( e );
    }
    //-----------------------------------------------------------------------//
    //End Of Task 1                                                  //
    //-----------------------------------------------------------------------//

    frequentSE FSE = new frequentSE();
    SER mySER=null;
    SECG secg;
    int w_width, round = 1;
    double min_fr;      //minimum frequency
    int EvnPoolSize = EvnPool.pointer;

    w_width = Integer.parseInt(args[1]);
    min_fr = 1.0*Integer.parseInt(args[2])/1000;
    System.out.println("w_width = "+w_width+"\t\tmin_fr = "+min_fr+"\n\n");
    System.out.println("--------------------Serial Episode Generation and Recognition--------------------");
    while(round == 1 || (FSE.pointer>0 && FSE.SEParr[FSE.pointer-1].pointer>0))
    {
     secg = new SECG(EvnPool, EvtSeq, FSE);
     secg.nextSCS();
//    secg.print();
     System.out.println("Size = "+secg.candidateSE.episodeSize+"\t\tCandidate Count =
"+secg.candidateSE.pointer);
     mySER = new SER(secg.candidateSE, EvtSeq, w_width, min_fr, FSE, EvnPoolSize);
     mySER.run();
     if(FSE.pointer>0)
     {
      System.out.println("Frequent Episode Count = "+FSE.SEParr[FSE.pointer-1].pointer);
//     System.out.println("\n------------- Frequent Serial Episode  -------------\n");
//     FSE.SEParr[FSE.pointer-1].print();
//     System.out.println("\n------------- END of Frequent Serial Episode  -------------\n");
     }
     round++;
    }
    mySER.beginset=null;
    mySER=null;
    System.out.println("--------------------PoS Episode Generation and Recognition--------------------");
    PoSECG PoSEcg = new PoSECG(FSE);
    PoSEcg.run();
    System.out.println("PoS Candidate Count = "+PoSEcg.candidatePoS.pointer);
//   PoSEcg.print();

    PoSER PosEr = new PoSER(PoSEcg.candidatePoS, EvtSeq, w_width, min_fr, EvnPoolSize);
    PosEr.run();
    System.out.println("Frequent PoS Count = "+PosEr.frequentPoSE.pointer);
//   System.out.println("\n------------ Frequent PoS Episode ------------\n");
//   PosEr.frequentPoSE.print();
//   System.out.println("\n------------ END of Frequent PoS Episode -------------\n");
    analysis anlz = new analysis(FSE,PosEr.frequentPoSE);
    anlz.run();
  }
}

class analysis
{
  frequentSE FSE;
  PoSEpisodePool frequentPoSE;

  analysis(frequentSE FSE,PoSEpisodePool frequentPoSE)
  {
    this.FSE = FSE;
    this.frequentPoSE = frequentPoSE;
  }

  public void run()
  {
    int i,j,k,m,n,p,q,counter,flag;
    PoSEpisode alpha;
    serialEpisode beta;
    System.out.println("PoS Size\t\tPoS Freq_count\t\tMax Serial Freq_count");
    for(i=0;i<frequentPoSE.size;i++)       //for each frequent PoS Episode
    {
      if(frequentPoSE.PoSPool[i]==null)  //skip the null ones
        continue;
      alpha = frequentPoSE.PoSPool[i];
      counter = 0;                    //sum of the counts of all possible serial episodes of the corresponding PoS
      for(j=0;j<FSE.size;j++)          //for each frequent serial episodes set
      {
        if(FSE.SEParr[j]==null)
          continue;
        if(FSE.SEParr[j].episodeSize<alpha.size_2 || FSE.SEParr[j].episodeSize>(alpha.size_1+alpha.size_2-
2))
          continue;
        for(k=0;k<FSE.SEParr[j].size;k++) //for each frequent serial episodes in one set
        {
          if(FSE.SEParr[j].SEPool[k]==null)
            continue;
          beta = FSE.SEParr[j].SEPool[k];
          n=p=flag=0;                 //alpha 's states; n:alpha1; p:alpha2
          for(m=0;m<beta.size;m++)       //for each state in serial automaton
          {
            q = n+p;
            if(n<alpha.size_1 && alpha.eventType_1[n]==beta.eventType[m])
          n++;
         if(p<alpha.size_2 && alpha.eventType_2[p]==beta.eventType[m])
          p++;
         if(q ==n+p)      //beta is not matching alpha
         {
          flag = 1;
          break;
         }
        }
        if(flag == 0 && n == alpha.size_1 && p == alpha.size_2)
        {
         counter = beta.freq_count>counter? beta.freq_count:counter;
//        beta.print();
//        System.out.println("beta.freq_count = "+beta.freq_count+"\n");
        }
       }
      }
//    alpha.print();
     System.out.println((alpha.size_1+alpha.size_2-2)+"\t\t"+alpha.freq_count+"\t\t"+counter);
    }
   }
}

class beginsetPoSE
{
  instancePoSEpisode iPoSE[];//each cell corresponds to a time, and is the head of link list, dummy
  int size;
  int beginpoint;
  int firstindex;

  beginsetPoSE(int size)
  {
   this.size = size;
   this.iPoSE = new instancePoSEpisode[1000];
   this.beginpoint=0;
   this.firstindex=0;
  }
  public int convertindex(int w,int fst, int bgpit)  //in w from 1;
  {
   int diff;
   w=w-1;      //input w from 1 to ...
   if(w<fst || w>=fst+1000)  //range of handle
    return w-fst;
   diff = w-fst;
   if(diff+bgpit<1000)
    return diff+bgpit;
   else
    return (diff+bgpit)-1000;
  }
  public instancePoSEpisode getiPoSE(int index)   //input index from 1
  {
   int index1 = convertindex( index, firstindex, beginpoint);
   if(index1<0 || index1>999)  //range of handle
    return null;
   else if(iPoSE[index1]==null || iPoSE[index1].w!= -1*index)
    return null;
  else
    return iPoSE[index1];
 }
 public void insert(PoSEpisode PoSE, int activeState_1, int activeState_2, int initTime, int status)
 {
  if(PoSE == null)
    return;
  int index = convertindex(initTime,firstindex,beginpoint);
  if(index>=0 && index<=999)
  {
   if(iPoSE[index]==null || iPoSE[index].w!= -1*initTime)
     iPoSE[index] = new instancePoSEpisode(null,-1,-1,0,-1*initTime);
   instancePoSEpisode tmpiiSE = new
instancePoSEpisode(PoSE,activeState_1,activeState_2,initTime,status); // copy
   tmpiiSE.insert(iPoSE[index]);
  }
  else if(index>999)
  {
   int move = index-999;
   if(beginpoint+(move%1000)<1000)
      beginpoint = beginpoint+(move%1000);
     else
      beginpoint = beginpoint+(move%1000)-1000;
   firstindex = firstindex+move;
   index = convertindex(initTime,firstindex,beginpoint);
   if(iPoSE[index]==null || iPoSE[index].w!= -1*initTime)
     iPoSE[index] = new instancePoSEpisode(null,-1,-1,0,-1*initTime);
   instancePoSEpisode tmpiiSE = new
instancePoSEpisode(PoSE,activeState_1,activeState_2,initTime,status); // copy
   tmpiiSE.insert(iPoSE[index]);
  }
 }
 public void delete(PoSEpisode PoSE, int activeState_1, int activeState_2,int initTime)
 {
  if(PoSE == null)
    return;
  int index = convertindex(initTime,firstindex,beginpoint);
  if(index>=0 && index<=999)
  {
   if(iPoSE[index]==null)
     return;
   instancePoSEpisode tmpiiSE = iPoSE[index].next; //first real node in the link list
   while(tmpiiSE!=null)
   {
     if(tmpiiSE.Alpha == PoSE && tmpiiSE.activeState_1 == activeState_1 && tmpiiSE.activeState_2 ==
activeState_2)
       tmpiiSE.delete();
     tmpiiSE = tmpiiSE.next;
   }
  }
 }
 //check no other (alpha, |alpha1|, |alpha2|) in window range start - (start+w_width-1), i.e.
 //no same episodes in current window in the accept state.
 public boolean checkDuplicate(PoSEpisode alpha, int start, int w_width)
 {
  instancePoSEpisode tmpiSE;
```

```
   for(int i=start;i<start+w_width;i++)
   {
    if(i<1 || i>size-1)
      continue;
    int index = convertindex(i,firstindex,beginpoint);
    if(index>=0 && index<=999)
    {
     if(iPoSE[index]==null)      //head is null or not
       continue;
     tmpiSE = iPoSE[index].next;
     while(tmpiSE!=null && iPoSE[index].w== -1*i)
     {
      if(tmpiSE.Alpha==alpha && tmpiSE.activeState_1==(alpha.size_1-1) &&
tmpiSE.activeState_2==(alpha.size_2-1))
        return true;
      tmpiSE = tmpiSE.next;
     }
    }
   }
   return false;
  }
  public void print()
  {
   System.out.println("\n--------  BeginSet PoSE  ------------\n");
   for(int i =0;i<1000;i++)
    if(iPoSE[i]!=null)
    {
     System.out.println("\n--------  i = "+i+"  ------------\n");
     iPoSE[i].print();
    }
   System.out.println("\n--------  END OF BeginSet PoSE  ------------\n");
  }

}


class beginsetSE
{
  instanceSerialEpisode iSE[];//each cell corresponds to a time, and is the head of link list, dummy
  int size;
  int beginpoint;
  int endpoint;
  int firstindex;

  beginsetSE(int size)
  {
   this.size = size;
   this.iSE = new instanceSerialEpisode[1000]; //size  index used from 0
   this.beginpoint=0;
   this.firstindex=0;
  }

  public int convertindex(int w,int fst, int bgpit)  //in w from 1;
  {
   int diff;
   w=w-1;     //input w from 1 to ...
  if(w<fst || w>=fst+1000)  //range of handle
    return w-fst;
  diff = w-fst;
  if(diff+bgpit<1000)
    return diff+bgpit;
  else
    return (diff+bgpit)-1000;
 }
 public instanceSerialEpisode getiSE(int index)    //input index from 1
 {
  int index1 = convertindex( index, firstindex, beginpoint);
  if(index1<0 || index1>999)  //range of handle
    return null;
  else if(iSE[index1]==null || iSE[index1].activeState!= -1*index)
    return null;
  else
    return iSE[index1];
 }
 public void insert(serialEpisode SE, int activeState, int initTime)
 {
  if(SE == null)
    return;
  int index = convertindex(initTime,firstindex,beginpoint);
//System.out.println("index="+index+"  initTime="+initTime+"  firstindex="+firstindex+"
beginpoint="+beginpoint);

  if(index>=0 && index<=999)
  {
    if(iSE[index]==null || iSE[index].activeState!= -1*initTime)
      iSE[index] = new instanceSerialEpisode(null,-1*initTime);
    instanceSerialEpisode tmpiiSE = new instanceSerialEpisode(SE,activeState); // copy and get a new iiSE
    tmpiiSE.insert(iSE[index]);
  }
  else if(index>999)
  {
    int move = index-999;
    if(beginpoint+(move%1000)<1000)
      beginpoint = beginpoint+(move%1000);
    else
      beginpoint = beginpoint+(move%1000)-1000;
    firstindex = firstindex+move;
    index = convertindex(initTime,firstindex,beginpoint);

    if(iSE[index]==null || iSE[index].activeState!= -1*initTime)
      iSE[index] = new instanceSerialEpisode(null,-1*initTime);
    instanceSerialEpisode tmpiiSE = new instanceSerialEpisode(SE,activeState); // copy and get a new iiSE
    tmpiiSE.insert(iSE[index]);
  }
 }
 public void delete(serialEpisode SE, int activeState, int initTime)
 {
  if(SE == null)
    return;
  int index = convertindex(initTime,firstindex,beginpoint);
  if(index>=0 && index<=999)
  {
    if(iSE[index]==null)
      return;
    instanceSerialEpisode tmpiiSE = iSE[index].next; //first real node in the link list
    while(tmpiiSE!=null)
    {
      if(tmpiiSE.SE == SE && tmpiiSE.activeState == activeState)
        tmpiiSE.delete();
      tmpiiSE = tmpiiSE.next;
    }
  }
}
//check no other (alpha, |alpha|) in window range start - (start+w_width-1), i.e.
//on same episodes in current window in the accept state.
public boolean checkDuplicate(serialEpisode alpha, int start, int w_width)
{
  instanceSerialEpisode tmpiSE;
  for(int i=start;i<start+w_width;i++)
  {
    if(i<1 || i>size-1)
      continue;
    int index = convertindex(i,firstindex,beginpoint);
    if(index>=0 && index<=999)
    {
      if(iSE[index]==null)      //head is null or not
        continue;
      tmpiSE = iSE[index].next;
      while(tmpiSE!=null && iSE[index].activeState== -1*i)
      {
        if(tmpiSE.SE==alpha && tmpiSE.activeState==(alpha.size-1))
          return true;
        tmpiSE = tmpiSE.next;
      }
    }
  }
  return false;
}
public void print()
{
  System.out.println("\n--------  BeginSet SE  ------------\n");
  for(int i =0;i<1000;i++)
  {
    if(iSE[i]!=null)
    {
      System.out.println("\n--------  i = "+i+"  ------------\n");
      iSE[i].print();
    }
  }
  System.out.println("\n--------  END OF BeginSet SE  ------------\n");
}

}

class eventPool
{
  String eventType [];
  int size;
int pointer;

eventPool(int size)
{
 this.size = size;
 eventType = new String[size];
 this.pointer = 0;
}
public void setEventType(String et)
{
 int flag = 0;
 for(int i=0;i<pointer;i++)
   if(eventType[i].compareTo(et)==0)
     flag = 1;
 if(pointer<size && flag == 0)
   eventType[pointer++] = et;
}
public String getEventType(int index)
{
 if(index<size)
   return eventType[index];
 else
   return "";
}
public int getEvent(String et)
{
 int i;
 for(i=0;i<size;i++)
   if(eventType[i].compareTo(et)==0)
     return i;
 return -1;    //not found
}
public void print()
{
 System.out.println("Event Pool");
 for(int i=0;i<pointer;i++)
   System.out.print(eventType[i]+"\t");
 System.out.println();
}
}


class eventSequence
{
 theEvent evt [];   //the Ts=1, Te=evt[pointer-1].theTime
 int size;
 int pointer;
 int mark;        //for output next event at time t

 eventSequence(int size)
 {
  this.size = size;
  evt = new theEvent[size];
  this.pointer = 0;
  this.mark = 0;
 }
 public void resetMark()
 {
  mark = 0;
 }
 public int getEvent(int t)
 {
  while(mark<size)
  {
   if(evt[mark].theTime == t)
     return evt[mark++].eventType;
   else if(evt[mark].theTime < t)
     mark++;
   else
     return -1;    //no event found at t
  }
  return -1;
 }

 public void setEvent(theEvent e)
 {
  evt[pointer++] = e;
 }
 public void print(eventPool EP)
 {
  System.out.println("Event Sequence");
  for(int i=0;i<pointer;i++)
    evt[i].print(EP);
  System.out.println();
 }
 public void print()
 {
  System.out.println("Event Sequence");
  for(int i=0;i<pointer;i++)
    System.out.println("event = "+evt[i].eventType+"   time = "+ evt[i].theTime);
  System.out.println();
 }
}


class frequentSE
{
 serialEpisodePool SEParr[];   //dynamic array, initial size 5, each time double size by self-adjusting
 int size;                  //capacity of SEParr
 int pointer;                //next free cell in SEParr, also the # of non-empty cells

 frequentSE()
 {
  this.size = 5;
  SEParr = new serialEpisodePool[size];
  this.pointer = 0;
 }

 //Insert a new serialEpisodePool in.
 //Each serialEpisodePool  should has difference size.
 public void setSEP(serialEpisodePool SEP)
 {
  if(pointer<size)
   insert(SEP);
  else    //dynamic increase the SEParr array
  {
   int i;
   serialEpisodePool tmp_SEParr[]= new serialEpisodePool[size*2];
   for(i=0;i<size;i++)
     tmp_SEParr[i] = SEParr[i];
   SEParr = tmp_SEParr;
   size = size*2;
   insert(SEP);
  }
 }
 public void insert(serialEpisodePool SEP)
 {
  SEParr[pointer++] = SEP;
 }

 public void print()
 {
  System.out.println("Frequent Serial Episode");
  for(int i=0;i<pointer;i++)
    SEParr[i].print();
 }
}


class instancePoSEpisode
{
 PoSEpisode Alpha;
 int activeState_1;     //0 - Alpha1.size-1
 int activeState_2;     //0 - Alpha2.size-1
 int T;   //time
 int w;   //0 diagonal,1 left,2 top;
 instancePoSEpisode pre = null;   //double linked list
 instancePoSEpisode next = null;

 instancePoSEpisode(PoSEpisode Alpha, int activeState_1, int activeState_2, int T, int w)
 {
  this.Alpha = Alpha;
  this.activeState_1 = activeState_1;
  this.activeState_2 = activeState_2;
  this.T = T;
  this.w = w;
 }

 //insert Alpha behind Beta
 public void insert(instancePoSEpisode Beta)
 {
  if(Beta == null)
    return;
  this.next = Beta.next;
  if(Beta.next != null)
    Beta.next.pre = this;
  Beta.next = this;
  this.pre = Beta;
 }

 //delete this itself
 public void delete()
 {
   this.pre.next = this.next;
   if(this.next != null)
     this.next.pre = this.pre;
 }

 public boolean compare(instancePoSEpisode Beta)
 {
   if(Beta.Alpha == this.Alpha && Beta.activeState_1 == this.activeState_1 && Beta.activeState_2 ==
this.activeState_2
      && Beta.T == this.T && Beta.w == this.w)
      return true;
   return false;
 }

 public void print()
 {

   if(Alpha!=null)
   {
     System.out.println("Instance PoS Episode");
     Alpha.print();
     System.out.println("i = "+activeState_1+" j = "+activeState_2+" T = "+T+" w = "+w);
   }
   if(next != null)
     next.print();
 }
}


class instanceSerialEpisode
{
  serialEpisode SE;
  int activeState;     //0 - SE.size-1
  instanceSerialEpisode pre = null;   //double linked list
  instanceSerialEpisode next = null;

  instanceSerialEpisode(serialEpisode SE, int activeState)
  {
    this.SE = SE;
    this.activeState = activeState;
  }

  //insert this behind iSE
  public void insert(instanceSerialEpisode iSE)
  {
    if(iSE == null)
      return;
    this.next = iSE.next;
    if(iSE.next != null)
      iSE.next.pre = this;
    iSE.next = this;
  this.pre = iSE;
 }

 //delete this itself
 public void delete()
 {
  this.pre.next = this.next;
  if(this.next != null)
    this.next.pre = this.pre;
 }

 public void print()
 {

  if(SE!=null)
  {
    System.out.println("Instance Serial Episode");
    SE.print();
    System.out.println("Active State  = "+activeState);
  }
  if(next != null)
    next.print();
 }
}

public class PoSECG     //PoS episode candidate generation
{
 PoSEpisodePool candidatePoS;
 frequentSE FSE;    //frequent serial episode set, all sizes

 PoSECG(frequentSE FSE)
 {
  this.FSE = FSE;
 }

 //generate all size of PoS episode candidate in one run
 public void run()
 {
  serialEpisode alpha, beta;
  candidatePoS = new PoSEpisodePool();
  for(int i=0;i<FSE.pointer;i++)
    for(int j=0;j<FSE.SEParr[i].pointer;j++)
    {
     alpha = FSE.SEParr[i].SEPool[j];
     if(alpha.size<3)
       continue;

     for(int m=i;m<FSE.pointer;m++)
       for(int n=0;n<FSE.SEParr[m].pointer;n++)
       {
        beta = FSE.SEParr[m].SEPool[n];
        if(beta.size<3)
          continue;
        if(alpha.compareTo(beta)>=0)    //alpha<beta
          continue;
        if(alpha.eventType[0] == beta.eventType[0] && alpha.eventType[alpha.size-1] ==
beta.eventType[beta.size-1])
        {
         PoSEpisode PoSE = new PoSEpisode(alpha.size,beta.size);
         for(int tmpi=0;tmpi<alpha.size;tmpi++)
           PoSE.setEventType(alpha.eventType[tmpi],1);  //1 means first episode alpha
         for(int tmpi=0;tmpi<beta.size;tmpi++)
           PoSE.setEventType(beta.eventType[tmpi],2);  //2 means second episode beta
         candidatePoS.setPoSEpisode(PoSE);     //form and add one PoS candidate
        }
       }
     }
  }

  public void print()
  {
   System.out.println("\n------------- PoS Candidate -------------\n");
   candidatePoS.print();
   System.out.println("\n------------- END of Candidate -------------\n");
  }
}

class PoSEpisode
{
  int eventType_1 [];
  int eventType_2 [];
  int size_1;
  int size_2;
  int pointer_1;   //this variable is used when initialize the PoS Episode, then reset to 0.
  int pointer_2;   //when do state transition,
  int freq_count;
  int inwindow;
  PoSEpisode(int size_1,int size_2)
  {
   this.size_1 = size_1;
   this.size_2 = size_2;
   eventType_1 = new int[size_1];
   eventType_2 = new int[size_2];
   this.pointer_1 = 0;
   this.pointer_2 = 0;
   this.freq_count = 0;
   this.inwindow = -1000000;
  }
  public void setEventType(int et,int swap)  //swap = 1, for 1st serial episode, swap = 2, for 2nd serial
episode,
  {
   if(swap==1)
     if(pointer_1<size_1)
       eventType_1[pointer_1++] = et;
   if(swap==2)
     if(pointer_2<size_2)
       eventType_2[pointer_2++] = et;
  }
  //compare this episode with input episode
  //this=se = 0, otherwise = 1
  public int compareTo(PoSEpisode se)
```

```
 {
  int i;
  if(this.size_1!=se.size_1 || this.size_2!=se.size_2)
    return 1;
  for(i=0;i<size_1;i++)
    if(this.eventType_1[i]!=se.eventType_1[i])
      return 1;
  for(i=0;i<size_2;i++)
    if(this.eventType_2[i]!=se.eventType_2[i])
      return 1;
  return 0;
 }

 public void print()
 {
  int i;
  System.out.println("PoS Episode");
  for(i=0;i<size_1;i++)
    System.out.print(eventType_1[i]+" ");
  System.out.print("\n");
  for(i=0;i<size_2;i++)
    System.out.print(eventType_2[i]+" ");
  System.out.print("\n");
  System.out.println("freq_count="+freq_count+"   inwindow="+inwindow);
 }
}


class PoSEpisodePool
{
 PoSEpisode PoSPool[];      //dynamic array, initial size 5, each time double size by self-adjusting
 int size;                  //capacity of PoS episode # in the pool
 int pointer;               //next free cell in the pool, also the # of PoS episode in the pool

 PoSEpisodePool()
 {
  this.size = 5;
  PoSPool = new PoSEpisode[size];
  this.pointer = 0;
 }
 public void setPoSEpisode(PoSEpisode alpha)   //this method calls insertPoSEpisode()
 {
  if(pointer<size)
    insertPoSEpisode(alpha);
  else   //dynamic increase the PoSEpisode array
  {
    int i;
    PoSEpisode tmp_Pool[]= new PoSEpisode[size*2];
    for(i=0;i<size;i++)
      tmp_Pool[i] = PoSPool[i];
    PoSPool = tmp_Pool;
    size = size*2;
    insertPoSEpisode(alpha);
  }
 }
 public boolean isExist(PoSEpisode alpha)   //check if alpha in the pool
 {
  for(int i=0;i<pointer;i++)
  {
   if(PoSPool[i].compareTo(alpha)==0)
     return true;
  }
  return false;
 }
 public void insertPoSEpisode(PoSEpisode alpha)
 {
  PoSPool[pointer++] = alpha;
 }

 public void print()
 {
  int i;
  System.out.println("PoS Episode Pool\nSize = " + pointer+"\n");
  for(i=0;i<size;i++)
   if(PoSPool[i]!= null)
     PoSPool[i].print();
 }
}


public class PoSER      //PoS Episode Recognition
{
 PoSEpisodePool candidatePoSE;
 eventSequence EvtSeq;
 int w_width;
 PoSEpisodePool frequentPoSE;
 double min_fr;      //minimum frequency
 int EvnPoolSize;
 waitsPoS waits; //each entry i is a likned list of iSE that next state is eventType[i].
 beginsetPoSE beginset;

 PoSER(PoSEpisodePool candidatePoSE, eventSequence EvtSeq, int w_width, double min_fr, int
EvnPoolSize)
  {
   this.candidatePoSE = candidatePoSE;
   this.EvtSeq = EvtSeq;
   this.w_width = w_width;
   this.min_fr = min_fr;
   this.EvnPoolSize = EvnPoolSize;
   waits = new waitsPoS(EvnPoolSize);
   for(int i=0;i<candidatePoSE.pointer;i++)
   {
    //set all episodes in candidate set to wait in the first states
    waits.add(candidatePoSE.PoSPool[i], 0,0,-1,-1);
   }
   //waits.print();
   this.beginset = new beginsetPoSE(EvtSeq.evt[EvtSeq.pointer-1].theTime+1);
  }

 public void run()
  {
```

```
int start,Ts, Te, event_A,t,T,i,j,w,episode_size_1,episode_size_2;
transitionsPoSE tranPoSE;
PoSEpisode alpha;

Ts = 1;
Te = EvtSeq.evt[EvtSeq.pointer-1].theTime;
EvtSeq.resetMark();
for(start=Ts-w_width+1; start<=Te+1; start++)
{
 t = start+w_width-1;

  instancePoSEpisode tmpiSE = null;
  if(start>1)
  {
    tmpiSE = beginset.getiPoSE(start-1);
  }
  while(start>1 && tmpiSE != null && tmpiSE.next != null)
  {
   alpha = tmpiSE.next.Alpha;         // alpha.print();
   i = tmpiSE.next.activeState_1;
   j = tmpiSE.next.activeState_2;
   w = tmpiSE.next.w;
   T = tmpiSE.next.T;

   if(alpha.size_1-1 == i && alpha.size_2-1 == j)    //if remove an automaton in the accepting state
   {
    if(!(beginset.checkDuplicate(alpha, start, w_width))) //if no the same alpha in window in  the
accepting state
    {
     alpha.freq_count = alpha.freq_count - alpha.inwindow + start;
     alpha.inwindow=-1000000;    //reset episode status
    }
    else
    {
     tmpiSE = tmpiSE.next;
     continue;    //avoid doing 'alpha.initialized[L] = -1;'
    }
   }

   else if(w==0)  waits.delete(alpha, i+1, j+1,T);
   else if(w==1)  waits.delete(alpha, i+1, j,T);
   else if(w==2)  waits.delete(alpha, i, j+1,T);
   else if(w==3)  waits.delete(alpha, i, j,T);

   tmpiSE = tmpiSE.next;
  }

  tranPoSE = new transitionsPoSE(null,-1,-1,0,-1); //just as head of link list
  event_A = EvtSeq.getEvent(t);

  while(event_A!=-1)    //for all events(event_A,t) in EvtSeq such that t = start+w_width-1
  {
   waits.reset(event_A);
   instancePoSEpisode alpha_j = waits.get(event_A);

   while(alpha_j!=null)    //for all (alpha,i,j) in waits(event_A)
  {
   i = alpha_j.activeState_1;
   j = alpha_j.activeState_2;
   T = alpha_j.T;
   w = alpha_j.w;
   alpha = alpha_j.Alpha;
   episode_size_1 = alpha.size_1;
   episode_size_2 = alpha.size_2;

   if(i == (episode_size_1-1) && j == (episode_size_2-1) && alpha.inwindow==-1000000)
   {
    alpha.inwindow = start;
   }
   if(i==j && j == 0)
   {
    tranPoSE.insert(alpha,0,0,t,event_A);  //0,0 - first states;  0 same states
   }
   else
   {
    tranPoSE.insert(alpha,i,j,T,event_A);
    if(w==0)
     beginset.delete(alpha, i-1,j-1,T);
    else if(w==1)
     beginset.delete(alpha, i-1,j,T);
    else if(w==2)
     beginset.delete(alpha, i,j-1,T);
    else if(w==3)
     beginset.delete(alpha, i,j,T);

    waits.delete(alpha, i, j, T, w);
   }
   alpha_j = waits.get(event_A);
  }
  event_A = EvtSeq.getEvent(t);
 }

 while(tranPoSE.next!= null)   //for all (alpha, i, j, t, w) in transitions
 {
  alpha = tranPoSE.next.PoSE;
  i = tranPoSE.next.activeState_1;
  j = tranPoSE.next.activeState_2;
  T = tranPoSE.next.trantime;
  w = tranPoSE.next.status;
  int ww=0;

  if(w==alpha.eventType_1[i] && w==alpha.eventType_2[j])   //if both alpha1,alpha2 changed states
  {
   if(i<alpha.size_1-1 && j<alpha.size_2-1)
     {waits.insert(alpha,i+1,j+1,T,0);ww=0;}
   else if(i<alpha.size_1-1)
     {waits.insert(alpha,i+1,j,T,1);ww=1;}   //alpha2 already in accepting state
   else if(j<alpha.size_2-1)
     {waits.insert(alpha,i,j+1,T,2);ww=2;}   //alpha1 already in accepting state
  }
  else if(w==alpha.eventType_1[i])   //if only alpha1 changed state and hasnot reached the last state
  {
        if(i<alpha.size_1-1)
          {waits.insert(alpha,i+1,j,T,1);ww=1;}
        else
          {waits.insert(alpha,i,j,T,3);ww=3;}
      }
      else if(w==alpha.eventType_2[j])
      {
        if(j<alpha.size_2-1)
          {waits.insert(alpha,i,j+1,T,2);ww=2;}
        else
          {waits.insert(alpha,i,j,T,3);ww=3;}
      }

      beginset.insert(alpha, i, j, T,ww);   //beginset(t) = beginset(t) Union {(alpha, i, j)}

      tranPoSE = tranPoSE.next;
    }
  }

  frequentPoSE = new PoSEpisodePool();
  double ratio;
  for(i=0;i<candidatePoSE.size && candidatePoSE.PoSPool[i]!=null;i++)
  {
    alpha = candidatePoSE.PoSPool[i];     //alpha.print();
    ratio = alpha.freq_count*1.0/(Te - 1 + w_width);
    if(ratio>=min_fr)
      frequentPoSE.setPoSEpisode(alpha);
  }
 }
}



public class SECG      //serial episode candidate generation
{
 eventPool EvnPool;
 eventSequence EvtSeq;
 serialEpisodePool candidateSE;
 frequentSE FSE;    //frequent serial episode set, all sizes

 SECG(eventPool EvnPool,eventSequence EvtSeq, frequentSE FSE)
 {
  this.EvnPool = EvnPool;
  this.EvtSeq = EvtSeq;
  this.FSE = FSE;
 }

 //generate next serial episode candidate set with size+1
 public void nextSCS()      //next Serial Candidate Set
 {
  //get next episode size to process.
  int nextSize;
  if(FSE.pointer==0)   //empty set so far
    nextSize = 1;
  else
    nextSize = FSE.SEParr[FSE.pointer-1].episodeSize + 1;
```

```
//set up a new candidate set for size = nextSize episodes
candidateSE = new serialEpisodePool(nextSize);

//when nextsize==1, i.e. all size 1 candidate serial episodes,
//use all event type as candidates
if(nextSize==1)
{
 for(int i=0;i<EvnPool.pointer;i++)
 {
  serialEpisode tmpSE = new serialEpisode(nextSize);
  tmpSE.setEventType(i);
  candidateSE.setserialEpisode(tmpSE);
 }
}
else
{
 int begin=0;
 //for all largest size frequent episodes in frequent set
 serialEpisodePool F_L;
 for(int i=0;i<FSE.SEParr[nextSize-2].pointer;i++)
 {
  boolean flag_j;
  //check if we entered a new block, and update block info if it is
  F_L = FSE.SEParr[nextSize-2];
  if(!(F_L.SEPool[i].isInSameBlock(F_L.SEPool[begin])))
   begin = i;     //reset the mark of beginning of next block
  //for eveny episodes in the block
  for(int j=begin;j<F_L.size && F_L.SEPool[j]!=null &&
F_L.SEPool[j].isInSameBlock(F_L.SEPool[i]);j++)
  {
   flag_j = false;
   serialEpisode alpha = new serialEpisode(nextSize);
   //build a potential candidate alpha
   for(int k=0;k<nextSize-1;k++)
    alpha.setEventType(F_L.SEPool[i].eventType[k]);
   alpha.setEventType(F_L.SEPool[j].eventType[nextSize-2]);
   //build and test all subepisodes beta of alpha
   for(int k=0;k<nextSize-2;k++)
   {
    serialEpisode beta = new serialEpisode(nextSize-1);
    for(int m=0;m<k;m++)
     beta.setEventType(alpha.eventType[m]);
    for(int m=k;m<nextSize-1;m++)
     beta.setEventType(alpha.eventType[m+1]);
    if(!(F_L.isExist(beta))) //find an infrequent subepisode
    {
     flag_j = true;
     break;
    }
   }
   if(flag_j == true)   //try next episode in the block
    continue;
   candidateSE.setserialEpisode(alpha);
  }
 }
}
```

```
    }
   public void print()
   {
    System.out.println("\n------------ Candidate ------------\n");
    candidateSE.print();
    System.out.println("\n------------ END of Candidate ------------\n");
   }
  }


public class SER     //Serial Episode Recognition
 {
  serialEpisodePool candidateSE;
  eventSequence EvtSeq;
  int w_width;
  frequentSE FSE;    //frequent serial episode set, all sizes
  double min_fr;     //minimum frequency
  int EvnPoolSize;
  instanceSerialEpisode waits[]; //each entry i is a likned list of iSE that next state is eventType[i].
  beginsetSE beginset;

  SER(serialEpisodePool candidateSE, eventSequence EvtSeq, int w_width, double min_fr, frequentSE
FSE, int EvnPoolSize)
   {
    this.candidateSE = candidateSE;
    this.EvtSeq = EvtSeq;
    this.w_width = w_width;
    this.min_fr = min_fr;
    this.FSE = FSE;
    this.EvnPoolSize = EvnPoolSize;
    this.waits = new instanceSerialEpisode[EvnPoolSize];
    for(int i=0;i<EvnPoolSize;i++)     //initialize heads
      this.waits[i] = new instanceSerialEpisode(null,-1);
    for(int i=0;i<candidateSE.pointer;i++)
    {
     //set all episodes in candidate set to wait in the first states
     instanceSerialEpisode iSE = new instanceSerialEpisode(candidateSE.SEPool[i],0); //2nd 0 means the
first state
     iSE.insert(this.waits[candidateSE.SEPool[i].eventType[0]]);
    }
    this.beginset = new beginsetSE(EvtSeq.evt[EvtSeq.pointer-1].theTime+1);
   }

  public void run()
   {
    int start,Ts, Te, event_A,t,j,episode_size;
    transitionsSE tranSE;
    serialEpisode alpha;

    episode_size = 0;
    Ts = 1;
    Te = EvtSeq.evt[EvtSeq.pointer-1].theTime;
    EvtSeq.resetMark();
    for(start=Ts-w_width+1; start<=Te+1; start++)
     {
      t = start+w_width-1;
    instanceSerialEpisode tmpiSE = null;
    if(start>1)
    {
      tmpiSE = beginset.getiSE(start-1);
    }
    while(start>1 && tmpiSE != null && tmpiSE.next != null)
    {
     alpha = tmpiSE.next.SE;
     int L = tmpiSE.next.activeState;
     if(alpha.size-1 == L)     //if remove an automaton in the accepting state
     {
       if(!(beginset.checkDuplicate(alpha, start, w_width))) //if no the same alpha in window in  the
accepting state
       {
        alpha.freq_count = alpha.freq_count - alpha.inwindow + start;
       }
       else
       {
        tmpiSE = tmpiSE.next;
        continue;     //avoid doing 'alpha.initialized[L] = -1;'
       }
     }
     else
       deleteWaits(alpha.eventType[L+1], alpha, L+1);
     alpha.initialized[L] = -1;
     tmpiSE = tmpiSE.next;
    }




    tranSE = new transitionsSE(null,-1,0); //just as head of link list
    event_A = EvtSeq.getEvent(t);
    while(event_A!=-1)    //for all events(event_A,t) in EvtSeq such that t = start+w_width-1
    {
      instanceSerialEpisode alpha_j = waits[event_A].next;
      while(alpha_j!=null)    //for all (alpha,j) in waits(event_A)
      {
       j = alpha_j.activeState;
       alpha = alpha_j.SE;
       episode_size = alpha.size;

       if(j == (alpha.size-1) && alpha.initialized[j]==-1)
       {
        alpha.inwindow = start;
       }
       if(j == 0)
       {
        tranSE.insert(alpha,0,t);
       }
       else
       {
        tranSE.insert(alpha,j,alpha.initialized[j-1]);
        beginset.delete(alpha, j-1, alpha.initialized[j-1]);
```

```
          alpha.initialized[j-1]=-1;
          deleteWaits(event_A, alpha, j);
         }
         alpha_j = alpha_j.next;
        }
        event_A = EvtSeq.getEvent(t);
       }
       while(tranSE.next!= null)  //for all (alpha, j, t) in transitions
       {
        alpha = tranSE.next.SE;
        j = tranSE.next.activeState;
        t = tranSE.next.trantime;
        alpha.initialized[j] = t;
        beginset.insert(alpha, j, t);  //beginset(t) = beginset(t) Union {(alpha,j)}
        if(j<alpha.size-1)    //if j < |alpha|, waits(alpha[j+1]) = waits(alpha[j+1]) Union {(alpha,j+1)}
        {
         instanceSerialEpisode iSE = new instanceSerialEpisode(alpha,j+1);
          tmpiSE = this.waits[alpha.eventType[j+1]].next;
         int flagiSE = 0;
         while(tmpiSE!=null)
         {
          if(tmpiSE.SE == alpha && tmpiSE.activeState == j+1)
          {
           flagiSE = 1;
           break;
          }
          tmpiSE = tmpiSE.next;
         }
         if(flagiSE == 0)
          iSE.insert(this.waits[alpha.eventType[j+1]]);
        }
        tranSE = tranSE.next;
       }


     }

     serialEpisodePool tmpsEP = new serialEpisodePool(episode_size);
     double ratio;
     for(int i=0;i<candidateSE.size && candidateSE.SEPool[i]!=null;i++)
     {
      alpha = candidateSE.SEPool[i];
      ratio = alpha.freq_count*1.0/(Te - 1 + w_width);
      if(ratio>=min_fr)
        tmpsEP.setserialEpisode(alpha);
     }
     FSE.setSEP(tmpsEP);
    }

    public void deleteWaits(int event_A, serialEpisode SE, int activeState)
    {
     instanceSerialEpisode tmpiSE = waits[event_A].next; //first real node
     while(tmpiSE!=null)
     {
      if(tmpiSE.SE == SE && tmpiSE.activeState == activeState)  //found
        tmpiSE.delete();
        tmpiSE = tmpiSE.next;
      }
    }

    public void printWaits(int event_A)
    {
     System.out.println("\n------  Waits  ------\n");
     System.out.println("\n------  Event "+event_A+"  ------\n");
     waits[event_A].print();
     System.out.println("\n------  END of Waits  ------\n");
    }
  }


class serialEpisode
{
  int eventType [];
  int size;
  int pointer;
  int initialized []; //cell i of this array stores the timestamp at which the automaton instance
                    //with active state eventType[i] initialized.
  int freq_count;
  int inwindow;
  serialEpisode(int size)
   {
    this.size = size;
    eventType = new int[size];
    this.pointer = 0;
    initialized = new int[size];
    for(int i=0;i<size;i++)
      initialized[i] = -1;      //means no legal value
    this.freq_count = 0;
    this.inwindow = 0;
   }
  public void setEventType(int et)
   {
    if(pointer<size)
      eventType[pointer++] = et;
   }
  //compare this episode with input episode using lexicographical order,
  //this<se = -1, this=se = 0, this>se = +1,
  public int compareTo(serialEpisode se)
   {
    int min,i;
    min = this.size<se.size?this.size:se.size;
    for(i=0;i<min;i++)
      if(this.eventType[i]<se.eventType[i])
        return -1;
      else if(this.eventType[i]>se.eventType[i])
        return 1;
    if(min == this.size && min == se.size)
      return 0;
    else if(min == this.size)
      return -1;
    else
      return 1;
  }
  //check if This is in the same block with se, i.e., the first size-1
  //event types are same
  public boolean isInSameBlock(serialEpisode se)
  {
    if(this.size!=se.size)
      return false;
    if(this.size == 1)
      return true;
    for(int i=0;i<this.size-1;i++)
      if(this.eventType[i] != se.eventType[i])
        return false;
    return true;
  }
  public void print()
  {
    int i;
    System.out.print("Serial Episode   Event Type:");
    for(i=0;i<size;i++)
      System.out.print(eventType[i]+" ");
    System.out.print("\n");
  }
  public static void main(String [] args)
  {
    serialEpisode SE = new serialEpisode(3);
    SE.setEventType(1);
    SE.setEventType(2);
    SE.setEventType(1);
    SE.print();
    serialEpisode SE1 = new serialEpisode(2);
    SE1.setEventType(1);
    SE1.setEventType(2);

    SE1.print();
    System.out.println(SE1.compareTo(SE));
  }
}


class serialEpisodePool
{
  serialEpisode SEPool[];    //dynamic array, initial size 5, each time double size by self-adjusting
  int episodeSize;           //event # in serial episode, must be the same
  int size;                  //capacity of serial episode # in the pool
  int pointer;               //next free cell in the pool, also the # of serial episode in the pool

  serialEpisodePool(int episodeSize)
  {
    this.episodeSize = episodeSize;
    this.size = 5;
    SEPool = new serialEpisode[size];
    this.pointer = 0;
  }
  public void setserialEpisode(serialEpisode SE)    //this method calls insertSerialEpisode()
  {
    if(SE.size != episodeSize)
    return;
  if(pointer<size)
    insertSerialEpisode(SE);
  else    //dynamic increase the serialEpisode array
  {
    int i;
    serialEpisode tmp_SEPool[]= new serialEpisode[size*2];
    for(i=0;i<size;i++)
      tmp_SEPool[i] = SEPool[i];
    SEPool = tmp_SEPool;
    size = size*2;
    insertSerialEpisode(SE);
  }
}
public boolean isExist(serialEpisode SE)   //check if SE in the pool
{
  if(SE.size != episodeSize)
    return false;
  for(int i=0;i<pointer;i++)
  {
    if(SEPool[i].compareTo(SE)==0)
      return true;
    else if(SEPool[i].compareTo(SE)==1)
      return false;
  }
  return false;
}
public void insertSerialEpisode(serialEpisode SE)   //keep the lexicographical order
{
  int tmp_pointer = 0;
  if(pointer==0)
    SEPool[pointer++] = SE;
  else if(SEPool[pointer-1].compareTo(SE)==-1)   //less than SE (newly insert to be)
    SEPool[pointer++] = SE;
  else if(SEPool[pointer-1].compareTo(SE)==0)    //equal to SE
    return;
  else if(SEPool[pointer-1].compareTo(SE)==1)    //great than SE
  {
    while(SEPool[tmp_pointer].compareTo(SE)==-1) //move up from beginning and stop at 1st one >=SE
      tmp_pointer++;
    if(SEPool[tmp_pointer].compareTo(SE)==0)
      return;
    else
    {
      for(int i=pointer;i>tmp_pointer;i--)   //move items up one step
        SEPool[i]=SEPool[i-1];
      SEPool[tmp_pointer] = SE;
      pointer++;
      return;
    }
  }
}

public void print()
{
  int i;
```

```
    System.out.println("Serial Episode Pool\nSize = " + pointer+"\n");
    for(i=0;i<size;i++)
      if(SEPool[i]!= null)
        SEPool[i].print();
  }
}


class theEvent
{
  int eventType;
  int theTime;

  theEvent(int eventType, int theTime)
  {
    this.eventType = eventType;
    this.theTime = theTime;
  }
  public void print(eventPool EP)
  {
    if(EP==null)
      System.out.println("Event: "+eventType+" "+theTime);
    else
      System.out.println("Event: "+EP.getEventType(eventType)+" "+theTime);
  }
}


class transitionsPoSE
{
  PoSEpisode PoSE;
  int activeState_1;     //0 - PoSE.size_1-1
  int activeState_2;     //0 - PoSE.size_2-1
  int trantime;          //begin from 1
  transitionsPoSE next = null;
  transitionsPoSE last = null;
  int status;  //if both automata change states, status=0; if 1st only, status=1; if 2nd only, status=2;

  transitionsPoSE(PoSEpisode PoSE, int activeState_1, int activeState_2,int trantime, int status)
  {
    this.PoSE = PoSE;
    this.activeState_1 = activeState_1;
    this.activeState_2 = activeState_2;
    this.trantime = trantime;
    this.status = status;
  }

  public void insert(PoSEpisode PoSE, int activeState_1, int activeState_2,int trantime, int status)
  {
    transitionsPoSE tranPoSE = new transitionsPoSE(PoSE, activeState_1, activeState_2,trantime, status);
    if(next == null)
    {
      this.next = tranPoSE;
      this.last = tranPoSE;
    }
    else
    {
     this.last.next = tranPoSE;
     this.last = tranPoSE;
    }
  }
  public void print(transitionsPoSE tmptranSE)
  {
   System.out.println("\n------------ Transition PoSE ------------\n");
   transitionsPoSE tranPoSE = tmptranSE;
   while(tranPoSE!=null)
   {
    tranPoSE = tranPoSE.next;
    if(tranPoSE!=null)
    {
     tranPoSE.PoSE.print();
     System.out.println("activeState_1 = "+tranPoSE.activeState_1+" activeState_2 =
"+tranPoSE.activeState_2+"  trantime = "+tranPoSE.trantime);
    }
   }
   System.out.println("\n------------ END of Transition PoSE ------------\n");
  }
}



class transitionsSE
{
 serialEpisode SE;
 int activeState;     //0 - SE.size-1
 int trantime;        //begin from 1
 transitionsSE next = null;
 transitionsSE last = null;

 transitionsSE(serialEpisode SE, int activeState, int trantime)
 {
  this.SE = SE;
  this.activeState = activeState;
  this.trantime = trantime;
 }

 public void insert(serialEpisode SE, int activeState, int trantime)
 {
  transitionsSE tranSE = new transitionsSE(SE,activeState,trantime);
  if(next == null)
  {
   this.next = tranSE;
   this.last = tranSE;
  }
  else
  {
   this.last.next = tranSE;
   this.last = tranSE;
  }
 }
 public void print(transitionsSE tmptranSE)
```

```
  {
   System.out.println("\n------------ Transition SE ------------\n");
   transitionsSE tranSE = tmptranSE;
   while(tranSE!=null)
   {
    tranSE = tranSE.next;
    if(tranSE!=null)
    {
     tranSE.SE.print();
     System.out.println("activeState = "+tranSE.activeState+"  trantime = "+tranSE.trantime);
    }
   }
   System.out.println("\n------------ END of Transition SE ------------\n");
  }
}


class waitsPoS
{
  instancePoSEpisode iPoSE[][];
  int size;     //# of event types
  int rw;  //row or column #. depends on direction
  instancePoSEpisode pointer; //use for search next element in waits[A][*] and waits[*][A]
  int direction;   //indicate current is waits[A][*]:0 or waits[*][A]:1

  waitsPoS(int size)
  {
   this.size = size;
   this.iPoSE = new instancePoSEpisode[size][size];
   for(int i=0;i<size;i++)
    for(int j=0;j<size;j++)
     this.iPoSE[i][j] = new instancePoSEpisode(null,-1,-1,-1,-1);
  }
  public void insert(PoSEpisode PoS, int i,int j,int T,int w)
  {
   instancePoSEpisode tmpPoS = iPoSE[PoS.eventType_1[i]][PoS.eventType_2[j]].next;
   while(tmpPoS!=null)
   {
    if(tmpPoS.Alpha==PoS && tmpPoS.activeState_1==i && tmpPoS.activeState_2==j)
    {
     if(tmpPoS.T<T)
     {
      tmpPoS.T = T;
      tmpPoS.w = w;
     }
     return;
    }
    tmpPoS=tmpPoS.next;
   }
   add(PoS, i,j, T, w);
  }

  public void add(PoSEpisode PoS, int i,int j,int t,int w)
  {
   instancePoSEpisode tmpiPoSE = new instancePoSEpisode(PoS,i,j,t,w);
   tmpiPoSE.insert(iPoSE[PoS.eventType_1[i]][PoS.eventType_2[j]]);
```

```
    //PoS.print();
    //System.out.println("i="+i+" j="+j+" t="+t+" w="+w+" j="+j+"
PoS.eventType_1[i]="+PoS.eventType_1[i]+" PoS.eventType_2[j]="+PoS.eventType_2[j]);
  }
  public void delete(PoSEpisode PoS, int i,int j,int t,int w)
  {
    instancePoSEpisode tmpiPoSE;
    tmpiPoSE = iPoSE[PoS.eventType_1[i]][PoS.eventType_2[j]].next;
    while(tmpiPoSE!=null)
    {
     if(tmpiPoSE.Alpha == PoS && tmpiPoSE.activeState_1 == i && tmpiPoSE.activeState_2 == j &&
        tmpiPoSE.T == t && tmpiPoSE.w == w)
      tmpiPoSE.delete();
     tmpiPoSE = tmpiPoSE.next;
    }
  }
  public void delete(PoSEpisode PoS, int i,int j,int T)
  {
    instancePoSEpisode tmpiPoSE;
    tmpiPoSE = iPoSE[PoS.eventType_1[i]][PoS.eventType_2[j]].next;
    while(tmpiPoSE!=null)
    {
     if(tmpiPoSE.Alpha == PoS && tmpiPoSE.activeState_1 == i && tmpiPoSE.activeState_2 == j &&
tmpiPoSE.T == T)
      tmpiPoSE.delete();
     tmpiPoSE = tmpiPoSE.next;
    }
  }
  public void reset(int A)
  {
    rw=0;
    pointer = iPoSE[A][0];
    direction = 0;
  }
  public instancePoSEpisode get(int A)   //search from iPoSE[A][*] then iPoSE[*][A]
  {
    if(pointer.next!=null)     //do all the linked list
     if(!(direction == 1 && rw==A))
     {
      pointer = pointer.next;
      return pointer;
     }

//System.out.println("get a POS >>>>>>>>>>>>>> 1 direction="+direction+" rw="+rw);
    if(direction == 0 && rw<size-1)
    {
     pointer = iPoSE[A][++rw];
     while(pointer.next==null && rw<size-1)
      pointer = iPoSE[A][++rw];
     if(pointer.next!=null)
     {
//      System.out.println("get a POS >>>>>>>>>>>>>> 2");
      pointer = pointer.next;
      return pointer;
     }       //end of horizontal search
     else
```

```
        {
//        System.out.println("get a POS >>>>>>>>>>>>>> 3");
         direction = 1;
         rw = 0;
         pointer = iPoSE[0][A];
         return get(A);
        }
      }
      else if(direction == 0 && rw==size-1)
      {
       direction = 1;
       rw = 0;
       pointer = iPoSE[0][A];
       return get(A);
      }
      else if(direction == 1 && rw<size-1)
      {
//      System.out.println("get a POS >>>>>>>>>>>>>> 4");
       pointer = iPoSE[++rw][A];
       while((pointer.next==null && rw<size-1)||(rw==A && rw<size-1))
        pointer = iPoSE[++rw][A];
       if(pointer.next!=null)
       {
//        System.out.println("get a POS >>>>>>>>>>>>>> 5");
        pointer = pointer.next;
        return pointer;
       }          //end of horizontal search
      }
     return null;
    }

   public void print()
   {
    System.out.println("\n------------- waits  --------------\n");
    instancePoSEpisode tmpPoS;
    for(int i=0;i<size;i++)
     for(int j=0;j<size;j++)
     {
      tmpPoS = iPoSE[i][j].next;
      if(tmpPoS!=null)
      {
       System.out.println("i="+i+"  j="+j);
       tmpPoS.print();
      }
     }
   System.out.println("\n============= END waits  --------------\n");
  }
}
```

VITA (1)

Guohuan Wang

Candidate for the Degree of

Master of Science

Thesis: ALGORITHMS FOR MINING POS-EPISODE

Major Field: Computer Science

Biographical:

Education: Received Bachelor of Science degree in Civil Engineering at Dalian University of Technology, July 1992, and Master of Science degree in Structural Engineering at Tianjin University, March 1997, China. Completed the requirements of the Master of Science degree with a major in Computer Science at Oklahoma State University in May 2004.

Experience: Employed as a civil engineer in Dalian University of Technology between July 1992 and September 1994. Employed by China Construction Bank as an engineer between March 1997 and January 2001. Employed as web application developer by Oklahoma State University, July 2001 to present.