Fundamenta Informaticae 47 (2001) 75–16
IOS Press

Discovering and Matching Elastic Rules from Sequence Databases

Sanghyun Park
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
shpark@cs.ucla.edu

Wesley W. Chu
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA
wwc@cs.ucla.edu

Abstract. This paper presents techniques for discovering and matching rules with *elastic patterns*. Elastic patterns are ordered lists of elements that can be stretched along the time axis. Elastic patterns are useful for discovering rules from data sequences with different sampling rates. For fast discovery of rules whose heads (left-hand sides) and bodies (right-hand sides) are elastic patterns, we construct a trimmed suffix tree from succinct forms of data sequences and keep the tree as a compact representation of rules. The trimmed suffix tree is also used as an index structure for finding rules matched to a target head sequence. When matched rules cannot be found, the concept of *rule relaxation* is introduced. Using a cluster hierarchy and relaxation error as a new distance function, we find the least relaxed rules that provide the most specific information on a target head sequence. Experiments on synthetic data sequences reveal the effectiveness of our proposed approach.

Keywords: Knowledge Discovery and Data Mining, Elastic Patterns, Sequence Databases, Type Abstraction Hierarchy, Rule Matching, Rule Relaxation

1. Introduction

Rule discovery from sequential data is a data mining technique for trend prediction [4, 9]. There have been several approaches [1, 8, 11, 16] to discover useful rules from such data. Most rules from data sequences have the format '$\mu[t1] \rightarrow \nu[t2]$' where μ and ν are patterns and $t1$ and $t2$ are time intervals.

They are interpreted as "if the pattern μ occurs within the time interval $t1$, then the pattern ν will follow within the time interval $t2$".

A pattern is defined as a partially ordered collection of elements. According to the constraints on the arrangement of elements, patterns can be classified as: serial patterns, parallel patterns, and non-serial and non-parallel patterns [10, 11]. Serial patterns require the occurrence of elements in a specified order while parallel patterns require the occurrence of elements without imposing any order. $\langle A, B\rangle$ and $\langle A, C, C, B\rangle$ are instances of a serial pattern AB while $\langle B, A\rangle$ is not. $\langle A\rangle$, $\langle B, A\rangle$ and $\langle B, C, C\rangle$ are instances of a parallel pattern $A|B$, but $\langle C\rangle$ is not. The patterns having both serial and parallel patterns are called non-serial and non-parallel patterns.

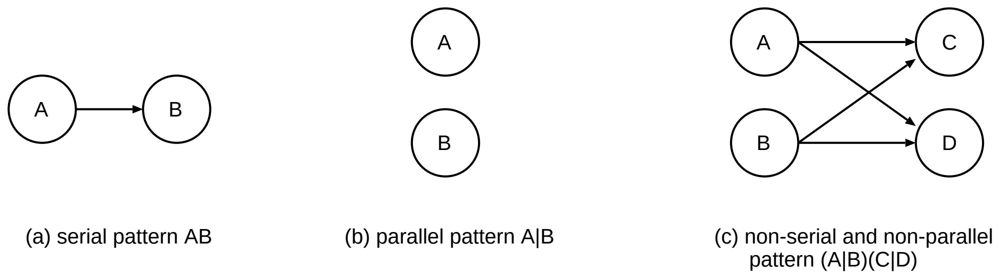


Figure 1. Graphical representations of a serial pattern AB, a parallel pattern $A|B$, and a non-serial and non-parallel pattern $(A|B)(C|D)$

As a subset of serial patterns, we can think of *elastic patterns* where elements can be stretched along the time axis by replicating themselves. Elastic patterns AB and ABC are interpreted as A^+B^+ and $A^+B^+C^+$, respectively, using the notation of a regular expression. $\langle A, B\rangle$ and $\langle A, A, B, B, B\rangle$ are instances of an elastic pattern AB while $\langle A, C, B\rangle$ is not. Elastic patterns are useful for discovering rules from data sequences whose sampling rates may vary. For example, consider medical data sequences that record the body temperatures of patients. Some data sequences may have temperature values taken every day while others may have values taken every week. Furthermore, even within a single data sequence, time intervals between neighboring temperature values can vary non-linearly. These sequences cannot be compared directly without considering stretches or compressions of elements along the time axis.

The rules whose heads (left-hand sides) and bodies (right-hand sides) are elastic patterns are called *elastic rules*. Given elastic patterns α and β, elastic rules have the format '$\alpha \rightarrow \beta$' that is interpreted as "if there occurs a sequence which is an instance of α, then it will be followed by a sequence which is an instance of β". Time intervals are not associated with elastic patterns because these patterns are flexible on the time axis.

There are many techniques [1, 8, 11, 16] to discover rules with serial patterns. Many of them use the relationship between patterns and their sub-patterns. Given the serial patterns AB occurring 200 times and $ABAC$ occurring 150 times, they extract the rule '$AB \rightarrow AC$ with confidence $\frac{150}{200}$(= 0.75)'. Infrequent patterns whose numbers of occurrences are below a threshold are ignored because infrequent patterns are considered insignificant. To find frequent patterns, they first find short frequent patterns and then combine them to generate longer candidate patterns. Candidate patterns are checked whether they are frequent or not. These combining and checking steps are repeated until all frequent patterns are found. Therefore, repeated readings of data sequences are unavoidable. However, if we focus on discovering elastic rules, better algorithms that require fewer database accesses can be developed.

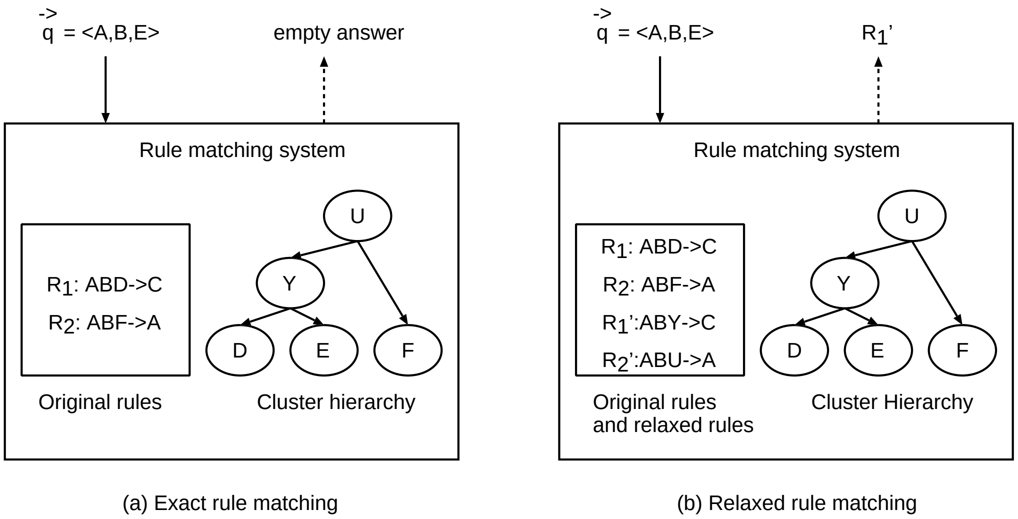


Figure 2. An example of rule matching. Without relaxation, neither rule can be matched to $\vec{q}$. However, both R_1' and R_2' cover $\vec{q}$ after relaxation. The least relaxed one (R_1') is returned to a user.

Once rules are discovered from data sequences, they may be used to predict the future trend of a target head sequence $\vec{q}$ via the process of rule matching. We say that a rule is matched to $\vec{q}$ when each element of the rule head is equal to the corresponding element of $\vec{q}$. However, if there are large number of rules, it is not a trivial task to find rules efficiently that are matched to $\vec{q}$. The fast finding of matched rules has not been well addressed regardless of its importance in real applications.

There are some occasions when we fail to find rules matched to a target head sequence $\vec{q}$. This failure often occurs when $\vec{q}$ is not a frequent sequence. For those infrequent target head sequences, we can introduce the concept of *rule relaxation*. Based on a cluster hierarchy, a rule R is relaxed to R' by replacing some elements of R with elements denoting more general concepts or broader range values. Given a target head sequence $\vec{q}$ and a rule R that is not matched to $\vec{q}$, we can relax R to R' so that R' *covers* $\vec{q}$. We say that a rule covers $\vec{q}$ when each element of the rule head represents the same range as or broader range than the one represented by the corresponding element of $\vec{q}$. In a cluster hierarchy, an ancestor node represents higher concept or broader range than the one represented by its descendant. For instance, consider the cluster hierarchy and the relaxed rule '$R_1' : ABY \rightarrow C$' shown in Figure 2. R_1' covers $\vec{q} = \langle A, B, E\rangle$ because the first two head elements (=AB) of R_1' are same as their corresponding elements of $\vec{q}$ and the third head element (=Y) of R_1' is the parent of the corresponding element (=E) in the cluster hierarchy.

Among many relaxed rules that can cover $\vec{q}$, we are interested in finding the least relaxed rules since they describe $\vec{q}$ more accurately than the other relaxed rules do. As an example, consider the rule matching system shown in Figure 2. Neither '$R_1 : ABD \rightarrow C$' nor '$R_2 : ABF \rightarrow A$' is matched to $\vec{q} = \langle A, B, E\rangle$. However, both rules can be relaxed to cover $\vec{q}$. Since D and F may by relaxed to Y and U respectively, we obtain the relaxed rules '$R_1' : ABY \rightarrow C$' and '$R_2' : ABU \rightarrow A$'. Now, both relaxed rules cover $\vec{q}$. Because R_1' is less relaxed than R_2', R_1' is chosen as an answer. We call the above process *relaxed rule matching*.

When the length of a rule head is not equal to that of $\vec{q}$, elements of the rule head may be replicated and then relaxed to cover $\vec{q}$. For instance, consider the target head sequence $\vec{q} = \langle A, B, D, E\rangle$ and the rule head ABD. The rule head can be converted to $ABDD$ by replicating the third element (=D) and then relaxed to $ABDY$ to cover $\vec{q}$. During the process of relaxed rule matching, we read each rule and compute the degree of relaxation needed for the rule to cover $\vec{q}$. Relaxed rule matching is time-consuming because the rules requiring the least relaxation can be found after processing all rules.

In this paper, we propose a method to discover elastic rules from sequence databases. We also present efficient techniques to find matched rules and to derive the least relaxed rules. We assume that data sequences consist of elements having univariate numeric values. The main data structure we are using is a suffix tree [15]. For fast discovery of elastic rules, we construct a trimmed suffix tree from succinct forms of data sequences. The trimmed suffix tree is used for both a compact representation of rules and an index structure for rule matching. Rules matched to a target head sequence are found by an exact matching algorithm, and the least relaxed rules are obtained by a similarity matching algorithm that uses a distance function based on relaxation error [7]. We use a type abstraction hierarchy (TAH) [5, 7] to acquire the symbolized representations of element values and to derive the relaxed rules.

The paper is organized as follows. Backgrounds on a suffix tree and a type abstraction hierarchy are described in Section 2. In Section 3, the method to discover elastic rules from data sequences is explained. The techniques to find matched rules and to derive the least relaxed rules are presented in Section 4 and Section 5, respectively. The experimental results are shown in Section 6. Finally, we conclude this paper with the summary and future extensions in Section 7.

2. Background

2.1. Suffix tree

A suffix tree [15] is an index structure that has been proposed as a fast access method to locate substrings (or subsequences) that are exactly matched to a target string (or a target sequence). A suffix is a subsequence that ends with the last element of a sequence, and a prefix is a subsequence that starts with the first element of a sequence. In a sequence whose length is L, there are L suffixes and L prefixes. We use the notation $\vec{x}[p : -]$ for the suffix starting from the p^{th} element of a sequence $\vec{x}$.

The suffix tree structure is based on *tries* and *suffix tries*. A trie is an indexing structure used for indexing sets of keywords of varying sizes. A suffix trie is a trie whose set of keywords comprises the suffixes of sequences. Nodes of a suffix trie with a single outgoing edge can be collapsed, yielding a suffix tree. Each suffix of a sequence is represented by a leaf node. The concatenation of the edge labels on the path from the root of the tree to the internal node N represents the longest common prefix of the suffixes represented by the leaf nodes under N. There are several algorithms [15] to build a suffix tree with computation complexity $O(M\bar{L})$ where M is the number of data sequences and $\bar{L}$ is the average length of data sequences. We use the notation PN for the parent node of N, and the notation $label(N_i, N_j)$ for the labels on the path connecting nodes N_i and N_j.

Figure 3 shows the suffix tree generated from two sequences $\vec{x} = \langle A, B, A\rangle$ and $\vec{y} = \langle B, A\rangle$. For simplicity, we merge all leaf nodes representing the same suffix. The node N stores the support value that is defined as the number of suffixes having $label(rootNode, N)$ as their prefixes. For example, N_2 stores the support value 3 because there are 3 suffixes ($\vec{x}[1 : -], \vec{x}[3 : -], \vec{y}[2 : -]$) that have $label(N_1, N_2)$ (=A) as their prefixes. \$ is used as an end marker of a suffix.

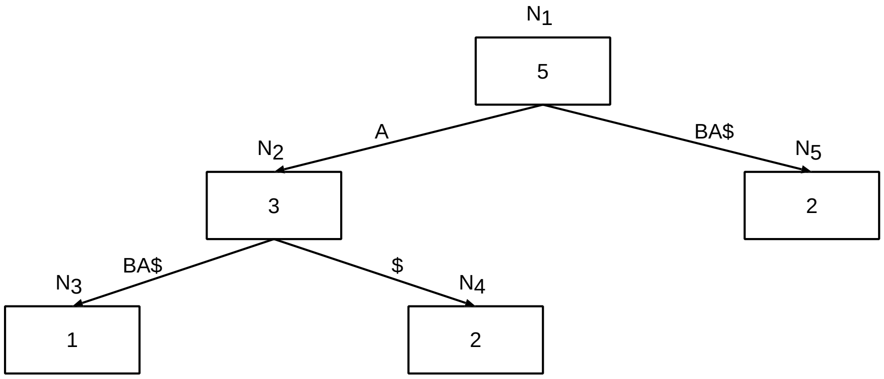


Figure 3. A suffix tree generated from $\vec{x} = \langle A, B, A\rangle$ and $\vec{y} = \langle B, A\rangle$. All leaf nodes representing the same suffix are merged. The support values are stored at the nodes.

2.2. Type abstraction hierarchy

Type Abstraction Hierarchy (TAH) [5, 7] is a data-driven multi-level cluster hierarchy that uses relaxation error as a goodness measure for clusters. For a cluster $C = \{x_1, x_2, ..., x_n\}$ having n elements that may or may not be unique, the relaxation error of C is defined as $RE(C) = \sum_{i=1}^{n}\sum_{j=1}^{n} \mid x_i - x_j \mid$. We can also think of the relaxation error of a partition. For a partition $P = \{C_1, C_2, ..., C_k\}$ having k clusters, the relaxation error of P is defined as $RE(P) = \sum_{i=1}^{k} P(C_i)RE(C_i)$ where $P(C_i)$ is the occurring probability of C_i. The TAH generation algorithm builds a TAH by partitioning clusters into sub-clusters until the number of elements in a cluster exceed the threshold T_{max}. The goodness measure for partitioning a cluster C into k sub-clusters $\{C_1, C_2, ..., C_k\}$ is given by the category utility, a relaxation error reduction per cluster, $CU = (RE(C) - \sum_{i=1}^{k} P(C_i)RE(C_i))/k$. The algorithms for generating binary and n-ary TAHs are given in [7]. TAH has the following benefits: 1) The algorithm considers both value and frequency distributions, thus generating more accurate clusters than equal-length interval clustering methods, and 2) TAH is easier to implement than maximum-entropy clustering methods. Figure 4 shows an example TAH built from data sequences whose elements take values within the range of $[0, 7.0)$. The relaxation error and the value range are stored in each node, and the nodes are labeled with the unique symbols.

3. Rule discovery

In this section, we propose an efficient method to discover elastic rules from data sequences via a suffix tree. We assume that the TAH has been generated from data sequences and distinct symbols have been assigned to the TAH nodes.

The support value of the pattern α is defined as the number of suffixes having α as their prefixes. SUP_{min} is the minimum support value that is used to filter out infrequent patterns. We also define the relative support value of the pattern α as $RSUP(\alpha)$ = (the number of suffixes having α as their prefixes) / (the total number of suffixes). For the applications where the total number of data sequences and their lengths may vary, the relative support is better than the (absolute) support.

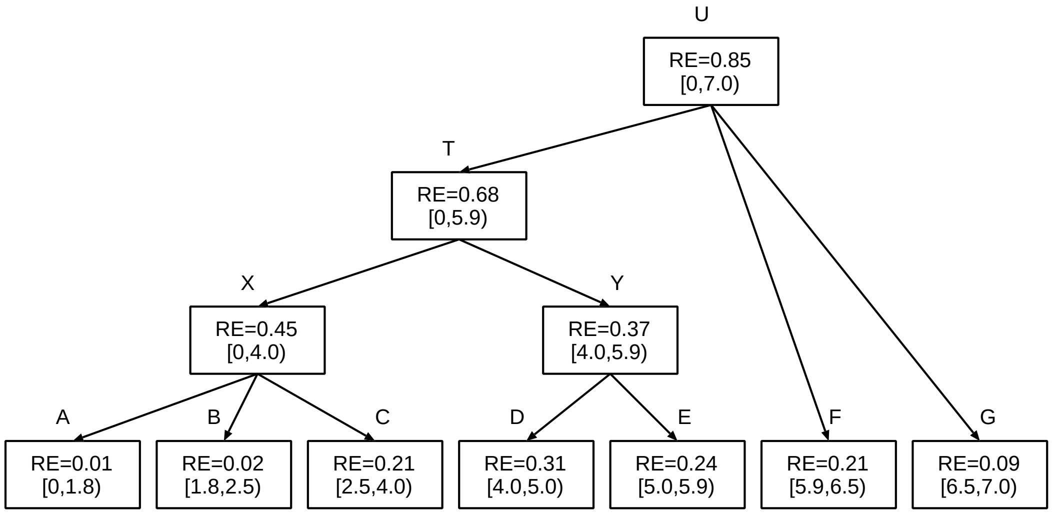


Figure 4. A TAH example. Each node is labeled with a unique symbol. The value range and the corresponding relaxation error are stored at each node.

The problem of elastic rule discovery is defined as follows: *Given a database with M sequences $\vec{x}_1, \vec{x}_2, ..., \vec{x}_M$ and the minimum support value SUP_{min}, discover rules composed of elastic patterns whose supports are at least SUP_{min}.* Our solution to the problem of elastic rule discovery consists of five steps as shown in Figure 5 : converting numeric elements to symbol elements, Compaction, suffix tree construction, trimming infrequent nodes, rule extraction.

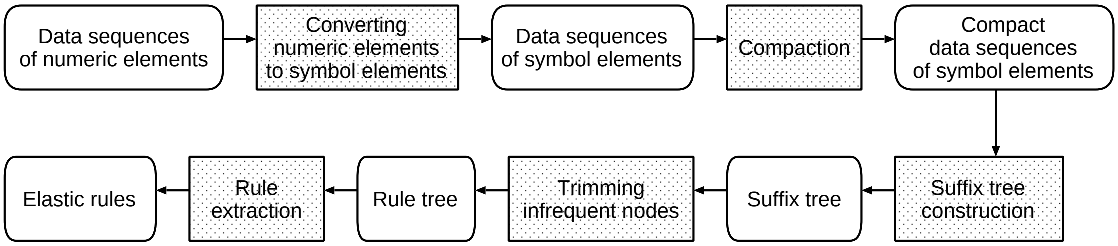


Figure 5. Five steps for elastic rule discovery.

Step 1. Converting numeric elements to symbol elements: We convert each numeric element of data sequences into the symbol of the corresponding leaf node of the TAH. The symbolized representation of $\vec{x}$ is denoted as $S(\vec{x})$. For example, based on the TAH in Figure 4, a data sequence $\vec{x} = \langle 3.4, 3.0, 3.7, 2.3, 2.1, 4.3 \rangle$ is converted to $S(\vec{x}) = \langle C, C, C, B, B, D \rangle$.

Step 2. Compaction: We convert the symbolized data sequence $S(\vec{x})$ into the compact representation $C(S(\vec{x}))$ by replacing consecutive elements that have the same value with a single element of that value. This step is for considering the property of elastic patterns. For example, $S(\vec{x}) = \langle C, C, C, B, B, D \rangle$ is converted to $C(S(\vec{x})) = \langle C, B, D \rangle$. We use the notation $\vec{X}$ for $C(S(\vec{x}))$.

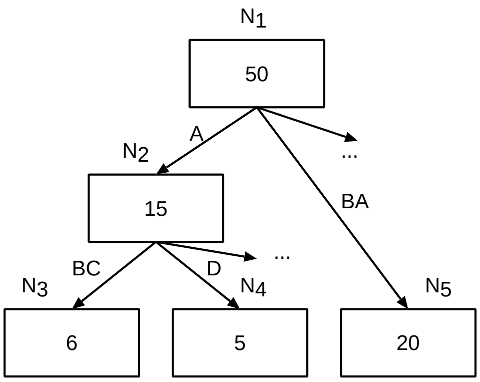
Step 3. Suffix tree construction: From the set of M converted data sequences $\vec{X}_1, ..., \vec{X}_M$, we build a suffix tree using either McCreight's algorithm [12] or incremental disk-based algorithm [3].

Step 4. Trimming: We compute the support values of the nodes and trim out the nodes whose support values are less than SUP_{min}. The support values of internal nodes are obtained by summing up the support values of their children nodes. The support values of the leaf nodes are the same as the number of suffixes represented by the leaf nodes. The trimmed suffix tree is called the *rule tree*.

Step 5. Rule extraction: We compute the confidence values of nodes and then extract rules. The expression for computing the confidence value of the node N is $confidence(N) = Support(N)/Support(PN)$ where PN is the parent node of N. If the number of labels on the path from PN to N is L, we extract L rules as shown in the following:

$R_1 : label(rootNode, PN) \rightarrow label(PN, N)$
$R_2 : label(rootNode, PN) \bullet (label(PN, N)[1:1]) \rightarrow label(PN, N)[2:L]$
$R_3 : label(rootNode, PN) \bullet (label(PN, N)[1:2]) \rightarrow label(PN, N)[3:L]$
...
$R_L : label(rootNode, PN) \bullet (label(PN, N)[1:L-1]) \rightarrow label(PN, N)[L:L]$

where $label(PN, N)[i:j]$ is the subsequence of $label(PN, N)$ including elements in positions i through j ($i \leq j \leq L$), and '$\bullet$' is the binary operator for concatenating two sequences. If N is the root node, then $label(rootNode, PN)$ becomes the empty sequence $\langle\rangle$. The confidence of R_1 is the same as confidence(N) while the confidences of R_2, R_3, ..., and R_L are 1. Figure 6 shows a part of a rule tree and the rules extracted from the tree. The values in the nodes represent their support values. Instead of storing extracted rules separately, we keep the rule tree as a compact representation of rules.



(a) A part of a rule tree

```
<> -> A  with confidence 15/50
<> -> BA with confidence 20/50
A  -> BC with confidence 6/15
A  -> D  with confidence 5/15
AB -> C  with confidence 1
B  -> A  with confidence 1
```

(b) Rules extracted from a rule tree

Figure 6. An example of a rule tree and the corresponding rules.

4. Exact rule matching

Rules discovered from data sequences may be used to predict the future trend of a target head sequence $\vec{q}$ via the process of rule matching. However, if there are large number of rules, it is not a trivial task to find rules that are matched to $\vec{q}$.

In this section, we present an efficient technique for exact rule matching. Exact rule matching is defined as follows: *Given a rule tree, a type abstraction hierarchy, and a target head sequence $\vec{q}$, find the rules matched to $\vec{q}$.* A rule is matched to $\vec{q}$ when each element of the rule head is identical to the corresponding element of $\vec{q}$.

This problem can be solved by conducting a sequential search over all the stored rules with computation complexity $O(N_R|\vec{q}|)$ where N_R is the number of rules and $|\vec{q}|$ is the length of $\vec{q}$. However, the search time is reduced if there is an index over the heads of rules. Instead of building a new index, we use the rule tree as an index structure for exact rule matching. As a result, matched rules can be found from the rule tree with complexity $O(|\vec{q}|)$. Our approach for exact rule matching consists of two steps, the search step and the rule extraction step, as discussed in the following subsections.

4.1. Search for exactly matched rule head

Using the rule tree as an index structure, we can find the rule head $\vec{h}$ that is exactly matched to a target head sequence. Algorithm 1 shows the exact matching algorithm RTI-E (Rule Tree Index for Exact matching). Note that $\vec{q}$ is converted to the compact representation $C(S(\vec{q}))$ before beginning the search process. We use the notation $\vec{Q}$ for $C(S(\vec{q}))$. The algorithm traverses the rule tree and returns a pair (N, p) that represents the matched rule head $\vec{h}$ = $label(rootNode, PN) \bullet (label(PN, N)[1:p])$. The first call to the algorithm has two arguments: rootNode and $\vec{Q}$.

Algorithm 1: Exact matching algorithm RTI-E

Input : node N, target head sequence $\vec{Q}$
Output : child node CN, length of matched prefix

Visit the node N;
Select the child node, CN, where $label(N, CN)$ is matched to the prefix of $\vec{Q}$;
Remove the matched prefix from $\vec{Q}$;
if $\vec{Q}$ *becomes empty* **then**
 return a pair (CN, the length of a matched prefix);
else
 call RTI-E(CN, $\vec{Q}$);

Example 4.1. Consider the TAH in Figure 4, the rule tree in Figure 6, and two target head sequences $\vec{q_1} = \langle 1.2, 1.5\rangle$ and $\vec{q_2} = \langle 1.2, 1.7, 2.3\rangle$. We first convert target head sequences into abbreviated symbolized representations, $\vec{Q_1} = \langle A\rangle$ and $\vec{Q_2} = \langle A, B\rangle$. The algorithm RTI-E returns (N_2, 1) for $\vec{Q_1}$ and (N_3, 1) for $\vec{Q_2}$.

4.2. Rule extraction from exactly matched rule head

Using the relationship between the exactly matched rule head and its following subsequences, we extract the rules. Let us assume that RTI-E has returned the pair (N, p) and the length of $label(PN, N)$ is L. If $p < L$, then the matched rule is '$label(rootNode, PN) \bullet (label(PN, N)[1 : p]) \rightarrow label(PN, N)[p+1 : L]$ with confidence 1'. Otherwise, the number of matched rules is the same as the number of children of N. For each child node CN of N, the matched rule is '$label(rootNode, N) \rightarrow label(N, CN)$ with confidence(CN)'.

Example 4.2. Let us continue the example. From the returned pair (N_2, 1), we extract two rules: '$A \rightarrow BC$ with confidence $6/15$' and '$A \rightarrow D$ with confidence $5/15$'. From the returned pair (N_3, 1), we extract the rule '$AB \rightarrow C$ with confidence 1'.

5. Relaxed rule matching

Relaxed rule matching is useful when we cannot find rules matched to a target head sequence $\vec{q}$. Instead of telling a user that we do not have any matched rules, it would be better to return the relaxed rules that cover $\vec{q}$. The rules become relaxed when some elements of rules are replaced with ancestors in the cluster hierarchy.

The problem of relaxed rule matching is defined as follows: *Given a rule tree, a type abstraction hierarchy, and a target head sequence $\vec{q}$, find the least relaxed rules that cover $\vec{q}$.* A rule covers $\vec{q}$ when each element of the rule head represents the same range or broader range than the one represented by the corresponding element of $\vec{q}$.

Among many relaxed rules covering $\vec{q}$, we focus on finding the least relaxed rules since they provide more specific information than others do. To find the least relaxed rules, we need a distance function that measures the degree of relaxation needed for the rule to cover $\vec{q}$. Note that the rule head whose length is not equal to $|\vec{q}|$ may be stretched and relaxed to cover $\vec{q}$. Therefore, the distance function has to consider stretches of elements along the time axis. Our approach for relaxed rule matching consists of two steps, the search step and the rule extraction step, which will be discussed in the following subsections.

5.1. Relaxation-based time warping distance function

To measure the degree of relaxation needed for the rule head $\vec{h}$ to cover $\vec{q}$, we propose a relaxation-error based time warping distance function $D_{RE}(\vec{h}, \vec{q})$, which is modified from an original time warping distance function [14] where elements are assumed to have numeric values and their distances are measured by Euclidean distance metric.

Definition 5.1. Given a rule head $\vec{h}$ and a target head sequence $\vec{q}$, a relaxation-error based time warping distance $D_{RE}(\vec{h}, \vec{q})$ is defined as follows:

$$
\begin{aligned}
D_{RE}(\langle\rangle, \langle\rangle) &= 0. \\
D_{RE}(\vec{h}, \langle\rangle) &= D_{RE}(\langle\rangle, \vec{q}) = \infty. \\
D_{RE}(\vec{h}, \vec{q}) &= d_{RE}(\vec{h}[1], \vec{q}[1]) + min \begin{cases} D_{RE}(\vec{h}, \vec{q}[2:-]) \\ D_{RE}(\vec{h}[2:-], \vec{q}) \\ D_{RE}(\vec{h}[2:-], \vec{q}[2:-]) \end{cases} \\
d_{RE}(\vec{h}[1], \vec{q}[1]) &= RE(ComNode(\vec{h}[1], \vec{q}[1])) - RE(Node(\vec{h}[1])).
\end{aligned}
$$

$RE(ComNode(\vec{h}[1], \vec{q}[1]))$ is the relaxation error of the lowest node containing both $\vec{h}[1]$ and $\vec{q}[1]$, and $RE(Node(\vec{h}[1]))$ is the relaxation error of the lowest node containing $\vec{h}[1]$. Therefore, $d_{RE}(\vec{h}[1], \vec{q}[1])$ is the relaxation error increase induced by relaxing $Node(\vec{h}[1])$ to $ComNode(\vec{h}[1], \vec{q}[1])$.

For example, let us consider the TAH shown in Figure 4. Because $RE(Node(A))$ is 0.01 and $RE(ComNode(A, D))$ (= $RE(T)$) is 0.68, $d_{RE}(A, D)$ = 0.68 − 0.01 = 0.67. $D_{RE}(\vec{h}, \vec{q})$ can be calculated efficiently by the dynamic programming technique [2] based on the recurrence relation $r(x, y)$ ($x = 1, 2, ..., |\vec{h}|, y = 1, 2, ..., |\vec{q}|$).

Definition 5.2. Given a rule head $\vec{h}$ and a target head sequence $\vec{q}$, the recurrence relation $r(x, y)$ ($x = 1, 2, ..., |\vec{h}|, y = 1, 2, ..., |\vec{q}|$) that calculates $D_{RE}(\vec{h}, \vec{q})$ is defined as follows:

$$
\begin{aligned}
r(0, 0) &= 0. \\
r(x, 0) &= r(0, y) = \infty. \\
r(x, y) &= d_{RE}(\vec{h}[x], \vec{q}[x]) + min(r(x, y-1), r(x-1, y), r(x-1, y-1)).
\end{aligned}
$$

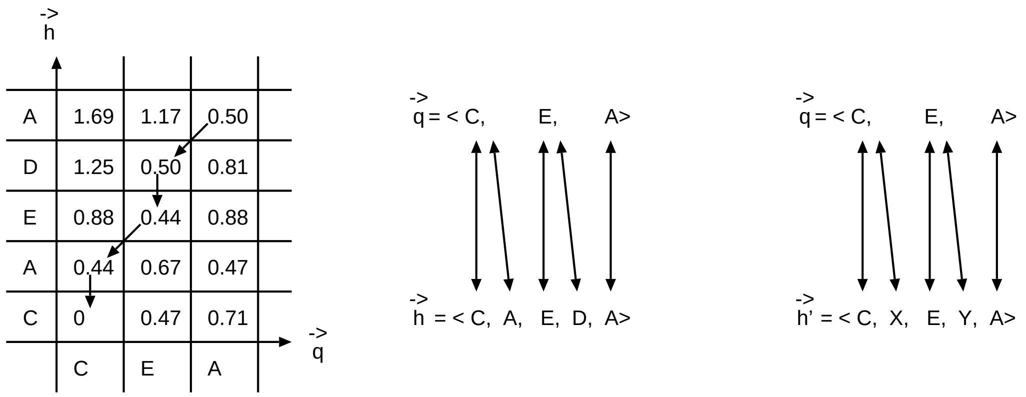
The recurrence relation $r(x, y)$ fills in the table of cumulative distances as the computation proceeds. The final cumulative distance, $r(|\vec{h}|, |\vec{q}|)$, is the degree of relaxation needed for $\vec{h}$ to cover $\vec{q}$. The mapping of elements that generates the minimum distance can be traced backward in the table - choosing the previous cells with the lowest cumulative distance. The recurrence relation $r(x, y)$ for calculating $D_{RE}(\vec{h}, \vec{q})$ has computation complexity $O(|\vec{h}||\vec{q}|)$.

Example 5.1. Figure 7 shows the cumulative distance table for computing $D_{RE}(\vec{h} = \langle C, A, E, D, A\rangle, \vec{q} = \langle C, E, A\rangle)$ and the best mapping of elements. $\vec{h}$ is converted to $\vec{h}'$ by relaxing the second and the fourth elements to X and Y, respectively, and $\vec{q}$ is converted $\vec{q}'$ by replicating the first and the second elements. Now, $\vec{q}'$ is covered by $\vec{h}'$. The relaxation error increase for converting $\vec{h}$ to $\vec{h}'$ is 0.50 and, therefore $D_{RE}(\vec{h}, \vec{q})$ = 0.50.

To find the rule head $\vec{h}$ that is most similar to $\vec{q}$, we traverse the rule tree using a depth-first downward approach. To speed-up the search process, we use the branch-pruning method based on Theorem 5.1.

Theorem 5.1. If all columns of the last row of the cumulative distance table have values greater than or equal to $MinDist$, adding more rows on this table does not yield the new values smaller than $MinDist$.

Proof:
The proof is given in [13]. □



(a) Cumulative distance table T for D_{RE}(<C,A,E,D,A>,<C,E,A>)

(b) Mapping of elements after time warping. The first and the third element of query sequence are replicated.

(c) Mapping of elements after relaxation. The second element (=A) and the fourth element (=D) of head sequence are relaxed to X and Y, respectively.

Figure 7. Cumulative distance table for $D_{RE}(\vec{h} = \langle C, A, E, D, A\rangle, \vec{q} = \langle C, E, A\rangle)$ and the mapping of elements that generates the minimum distance.

5.2. Search for the nearest rule head

To generate the least relaxed rules, we first traverse the rule tree to find the rule head $\vec{h}$ that requires the least relaxation to cover a target head sequence $\vec{q}$. The similarity matching algorithm RTI-S (Rule Tree Index for Similarity matching) is given in Algorithm 2. Note that a target head sequence $\vec{q}$ is converted to the compact representation $\vec{Q}$ before beginning the search process. The algorithm maintains three global variables during its execution: $\vec{Q}$, the nearest rule head $\vec{h}$ found so far, and its distance $MinDist$ from $\vec{Q}$. The first call to the algorithm has two arguments: rootNode and emptyTable.

Algorithm 2: Similarity matching algorithm RTI-S

Input : node N, cumulative distance table T

Visit the node N;
for *each child node* CN **do**
- Build a new cumulative distance table $newT$, by adding rows corresponding to $label(N, CN)$ on T;
- Find a nearer rule head $\vec{h}$ from $newT$ and update $MinDist$;
- **if** *further traverse-down the tree is necessary* **then**
  - call RTI-S(CN, $newT$);

The algorithm starts from the root node. When it visits a node, it inspects each child node to find a nearer rule head and to determine if further going-down the tree is necessary. Let us assume that the search algorithm examines the child CN of the node N. The first step is to build a cumulative distance table for $label(N, CN)$ (located on Y-axis) and $\vec{Q}$ (located on X-axis). If N is the root node, the cumulative distance table is built from the bottom. Otherwise, it is built by augmenting new rows on the top of the cumulative distance table that has been accumulated from the root node to N. The next step is to examine the last columns of newly added rows to find a nearer rule head. If there is a column having the distance value less than $MinDist$, the nearest rule head $\vec{h}$ is changed and $MinDist$ is also updated accordingly. Remember that $\vec{h}$ and $MinDist$ are global variables. The final step is to check all columns of the last row to determine whether or not further going-down the tree is needed. If at least one column of the last row has a value less than $MinDist$, we continue traverse down the tree. Otherwise, the search moves to the next child of N.

The least relaxed rules can also be found by a sequential scanning over all the rules with computation complexity $O(N_R \bar{H} |\vec{Q}|)$ where N_R is the number of rules and $\bar{H}$ is the average length of rule heads. RTI-S has the same complexity but it reduces the search time by applying the branch-pruning approach [13] and by allowing the cumulative distance table to be shared by rule heads that have the same prefix.

5.3. Rule extraction from the nearest rule head

After finding the rule head $\vec{h}$ most similar to $\vec{Q}$, we generate the least relaxed rules from $\vec{h}$ and its following subsequences. This step begins with extracting the rules from $\vec{h}$ using the method explained in Section 4.2. Then, we convert symbols of rule heads and bodies into their relaxed symbols according to the mapping of $\vec{h}$ and $\vec{Q}$, and get the compact representations of rules. Finally, the rules having the same head and body are merged and their confidence values are recomputed.

Example 5.2. Let us assume that we extract three rules from $\vec{h}$: '$ABD \rightarrow BG$ with confidence 20/50', '$ABD \rightarrow AG$ with confidence 10/50', and '$ABD \rightarrow C$ with confidence 10/50'. If both A and B are to be relaxed to X, the rules are converted as: '$XD \rightarrow XG$ with confidence 20/50', '$XD \rightarrow XG$ with confidence 10/50', and '$XD \rightarrow C$ with confidence 10/50'. After merging the rules that have the same head and body, we obtain the final rules: '$XD \rightarrow XG$ with confidence 30/50' and '$XD \rightarrow C$ with confidence 10/50'.

6. Experiments

To study the effectiveness of our proposed methods, we performed several experiments on the S&P 500 stock data sequences (http://biz.swcp.com/stocks) and on synthetic data sequences. The stock data were based on their daily closing prices. A total of 545 stock sequences was used with an average length of 232. The expression for generating the data sequence was defined as $\vec{x}_i[p] = \vec{x}_i[p-1] + Z_p$ where Z_p (p = 1,2,...) are independent, identically distributed random variables. The number of and the average length of synthetic data sequences vary according to each experiment. We used the relative minimum support value $RSUP_{min}$ to control the number of discovered rules.

6.1. Rule discovery

We used the total elapsed time as a performance measure of our rule discovery algorithm. First, we increased the number of sequences from 100 to 10,000 while keeping their average length constant at 200. Then, we changed the average length of sequences from 100 to 1,000 while maintaining the number of sequences at 500. As shown in Figures 8 and 9, the total elapsed times increase linearly as the number

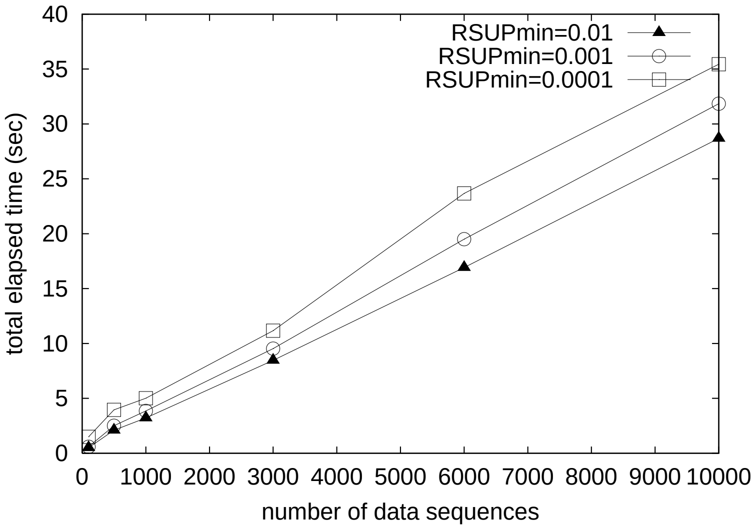


Figure 8. Total elapsed time for discovering elastic rules with selected numbers of data sequences. The average length of data sequences is kept at 200.

of and the average length of data sequences grow. The figures also show that the linearity is maintained with different $RSUP_{min}$ values.

6.2. Rule matching

To evaluate the efficiency of the proposed rule matching algorithms, RTI-E and RTI-S, we compared their execution times with that of sequential scanning. For our experiments, we generated a binary TAH from 500 data sequences whose average length is 400. With the threshold T_{max} set at 6,000, which is the maximum number of elements that can be contained in a leaf node, the TAH has 44 internal nodes and 45 leaf nodes.

Sequential scanning reads the rules sequentially and compares their heads with the target head sequence $\vec{Q}$. For a fast sequential search, only a subset of rules whose heads have the length of $|\vec{Q}|$ are compared with $\vec{Q}$. Figure 10 shows the average search times of RTI-E and SS(Sequential- Scanning)-based exact matching algorithm for a different number of rules. The search times of SS-based exact matching algorithm increase linearly with the number of rules while the search times of RTI-E remain relatively constant.

Since the most expensive step in relaxed rule matching is to find the rule head that is most similar to $\vec{Q}$, we have compared our similarity matching algorithm RTI-S with sequential scanning. Sequential scanning reads the rules sequentially and builds the cumulative distance table for each rule head. The average search times of RTI-S and sequential scanning are shown in Figure 11. The performance gain of our approach increases as the number of rules increases.

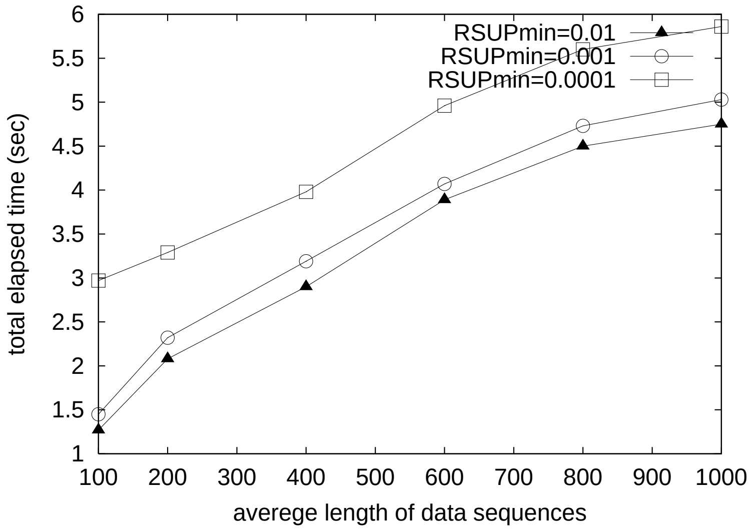


Figure 9. Total elapsed time for discovering elastic rules with selected average length of data sequences. The total number of data sequences is kept at 500.

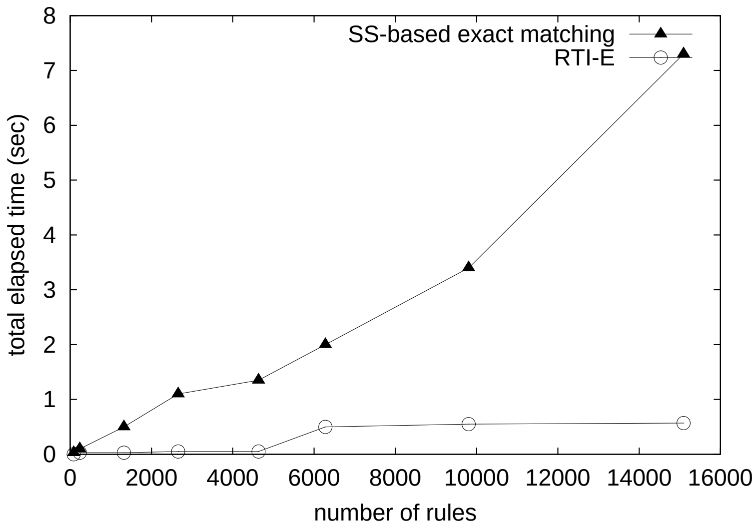


Figure 10. Performance comparison between sequential scanning and RTI-E for exact rule matching.

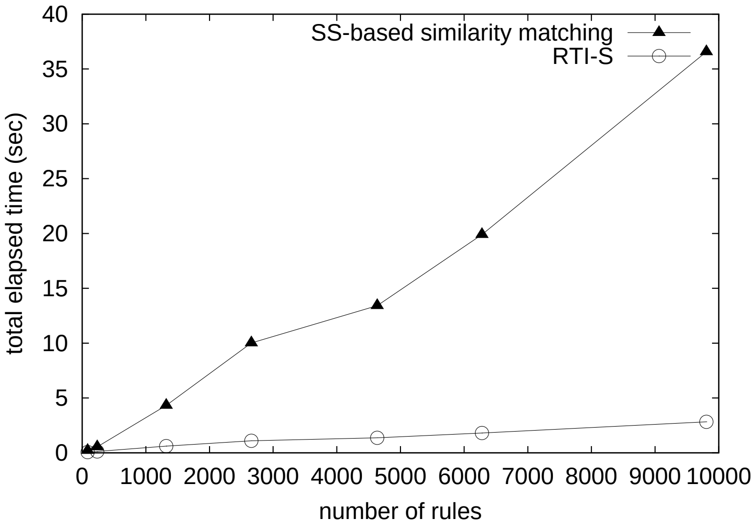


Figure 11. Performance comparison between sequential scanning and RTI-S for relaxed rule matching.

7. Conclusion

In this paper, we proposed a method to discover elastic rules from sequence databases. We also presented efficient techniques to find matched rules and to derive the least relaxed rules. We focused on data sequences consisting of univariate numeric values. If elements are non-numeric, we employ an encoding scheme that converts non-numeric elements to numeric elements.

The contributions of our work are: 1) proposing the concepts of elastic patterns and elastic rules for data sequences with different sampling rates, 2) presenting an efficient rule discovering algorithm which converts data sequences into succinct forms and then builds a rule tree, 3) introducing the idea of rule relaxation and suggesting the relaxation-error based time warping distance, and 4) presenting effective algorithms for exact and relaxed rule matchings.

Experiments on synthetic data sequences revealed that: 1) our rule discovering algorithm is linear to both the total number of and the average length of data sequences, and 2) our exact and relaxed rule matching algorithms are a few orders of magnitude faster than sequential scanning.

There are many useful applications of elastic rules. In the medical domain, they can be used for predicting the disease evolution patterns of a patient. In the web-environment, based on a user's web-site visiting history, the system may retrieve the next URL in advance while the user is browsing the current site.

The problems of discovering and matching elastic rules are more complicated if we deal with data sequences of multivariate real values. The Multi-Attribute TAH (MTAH) [7] may be used to acquire the symbolized representations of multi-variate elements and to get the relaxed rules. We are currently working toward this direction using medical image sequences [5, 6].

References

[1] Agrawal, R., Srikant, R.: Mining Sequential Patterns, *Proc. IEEE International Conference on Data Engineering (ICDE)*, 1995.

[2] Berndt, D. J., Clifford, J.: Finding Patterns in Time Series: A Dynamic Programming Approach, in: *Advances in Knowledge Discovery and Data Mining*, AAAI/MIT, 1996, 229–248.

[3] Bieganski, P., Riedl, J., Carlis, J. V.: Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation, *Proc. Hawaii International Conference on System Sciences*, 1994.

[4] Bradley, P. S., Fayyad, U. M., Mangasarian, O. L.: *Data Mining: Overview and Optimization Opportunities*, Technical Report MSR-TR-98-04, Microsoft Research Lab, 1998.

[5] Chu, W. W., Cardenas, A. F., Taira, R. K.: KMeD: a Knowledge-based Multimedia Medical Distributed Database System, *Information Systems*, **20**(2), 1995, 75–96.

[6] Chu, W. W., Cardenas, A. F., Taira, R. K.: Knowledge-Based Image Retrieval with Spatial and Temporal Constructs, *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, **10**(6), 1998, 872–888.

[7] Chu, W. W., Chiang, K.: Abstraction of High Level Concepts from Numerical Values in Databases, *Proc. AAAI Workshop on Knowledge Discovery in Databases*, 1994.

[8] Das, G., Lin, K., Mannila, H., Renganathan, G., Smyth, P.: Rule Discovery from Time Series, *Proc. International Conference on Knowledge Discovery and Data Mining (KDD)*, 1998.

[9] Fayyad, U. M.: Mining Databases: Toward Algorithms for Knowledge Discovery, *Data Engineering Bulletin*, **21**(1), 1998.

[10] Mannila, H., Toivonen, H.: Discovering Generalized Episodes using Minimal Occurrences, *Proc. International Conference on Knowledge Discovery and Data Mining (KDD)*, 1996.

[11] Mannila, H., Toivonen, H., Verkamo, A. I.: Discovering Frequent Episodes in Sequences, *Proc. International Conference on Knowledge Discovery and Data Mining (KDD)*, 1995.

[12] McCreight, E. M.: A Space-Economical Suffix Tree Construction Algorithm, *Journal of ACM*, **23**(2), 1998, 262–272.

[13] Park, S., Chu, W. W., Yoon, J., Hsu, C.: Efficient Searches for Similar Subsequences of Different Lengths in Sequence Databases, *Proc. IEEE International Conference on Data Engineering (ICDE)*, 2000.

[14] Rabinar, L., Juang, B.-H.: *Fundamentals of Speech Recognition*, Prentice Hall, 1993.

[15] Stephen, G. A.: *String Searching Algorithms*, World Scientific Publishing, 1994.

[16] Wang, J. T., Chirn, G., Marr, T. G., Shapiro, B., Shasha, D., Zhang, K.: Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results, *Proc. ACM SIGMOD*, 1994.