Discovering and Matching Elastic Rules from Sequence Databases

Contact Author: **Sanghyun Park** (shpark@cs.ucla.edu)
Phone: (310) 206-4825
Fax: (310) 825-7578
Boelter Hall 4833
Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA

Author: **Wesley W. Chu** (wwc@cs.ucla.edu)
Department of Computer Science
University of California, Los Angeles

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. For example, $\langle A, A, B, B, B\rangle$ is an instance of an elastic pattern AB while $\langle A, C, B\rangle$ is not. 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 a new distance function based on relaxation error, we find the least relaxed rules that provide more specific information on a target head sequence than the other relaxed rules do. Experiments on synthetic data sequences shows the effectiveness of our proposed approach.

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

1 Introduction

Rule discovery from sequential data is one of the major areas in data mining for trend prediction [3, 8]. There have been several approaches [1, 7, 9, 10, 13, 15] 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 [9, 10]. Serial patterns require the occurrence of elements in a specified order while parallel patterns require the occurrence of elements without imposing any order. The patterns having both serial and parallel patterns are called non-serial and non-parallel patterns.

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, 7, 9, 10, 13, 15] 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.

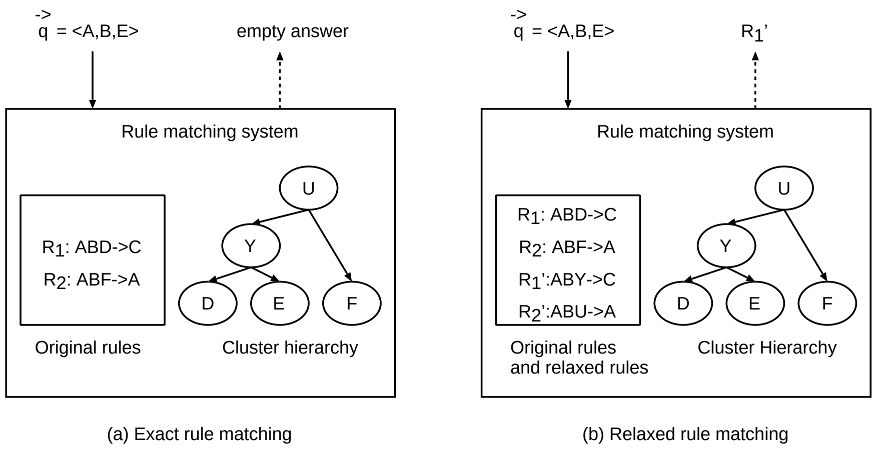


Fig. 1. An example of rule matching. Without relaxation, neither rule can be matched to $\vec{q}$. However, both R_1' and R_2' cover $\vec{q}$ after relaxation. The least relaxed one (R_1') is returned to a user.

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 $\vec{q}$.

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 higher concepts or broader ranges. 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' can *cover* $\vec{q}$. 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 1. 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 1. 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.*

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 [14]. 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 [5]. We use a type abstraction hierarchy (TAH) [5, 6] to acquire the symbolized representations of element values and to derive the relaxed rules.

2 Background

2.1 Suffix tree

A suffix tree [14] 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). 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. 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.

2.2 Type abstraction hierarchy

Type Abstraction Hierarchy (TAH) [5, 6] 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 $RE(C) = \sum_{i=1}^{n} \sum_{j=1}^{n} \mid x_i - x_j \mid$. The algorithms for generating binary and n-ary TAHs are given in [5]. 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 2 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.

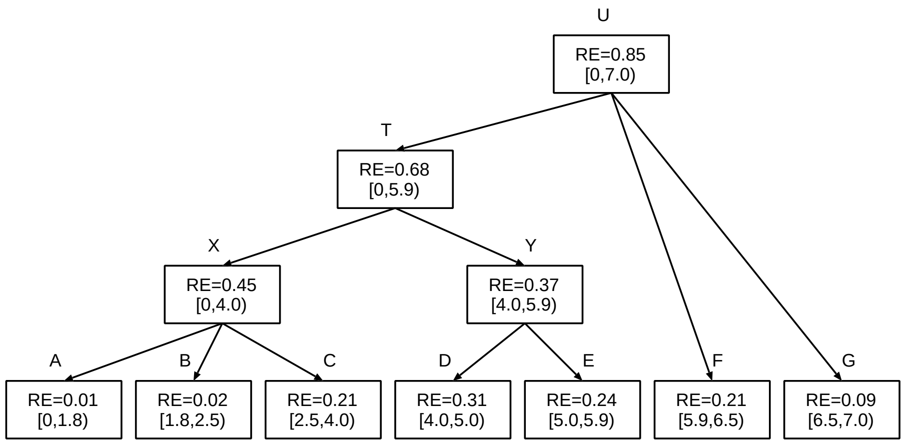


Fig. 2. A TAH example. Each node is labeled with a unique symbol. The value range and the corresponding relaxation error are stored at each node.

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 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}.

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 can 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). The relative support is better than the (absolute) support in applications where the total number of data sequences and their lengths may vary.

Our solution to the problem of elastic rule discovery consists of five steps as shown in Figure 3 : converting numeric elements to symbol elements, Compaction, suffix tree construction, trimming infrequent nodes, rule extraction.

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, a data sequence $\vec{x} = \langle 3.4, 3.0, 3.7, 2.3, 2.1, 4.3 \rangle$ is converted to $S(\vec{x}) = \langle C, C, C, B, B, D \rangle$ according to the TAH in Figure 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

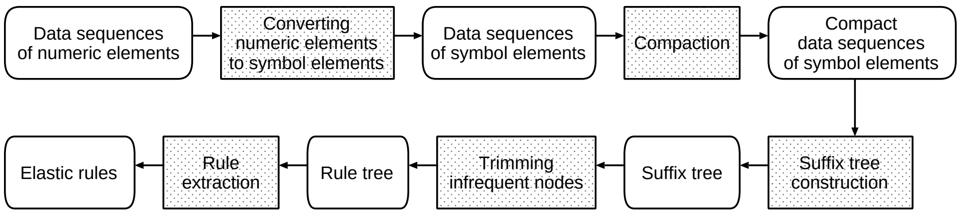

Data sequences of numeric elements
Converting numeric elements to symbol elements
Data sequences of symbol elements
Compaction
Compact data sequences of symbol elements
Suffix tree construction
Suffix tree
Trimming infrequent nodes
Rule tree
Rule extraction
Elastic rules

Fig. 3. Five steps for elastic rule discovery.

value with a single element of that value. This step is for considering the property of elastic patterns. For example, $S(\vec{x}) = \langle C, C, C, B, B, D, E, E \rangle$ is converted to $C(S(\vec{x})) = \langle C, B, D, E \rangle$. We use the notation $\vec{X}$ for $C(S(\vec{x}))$.

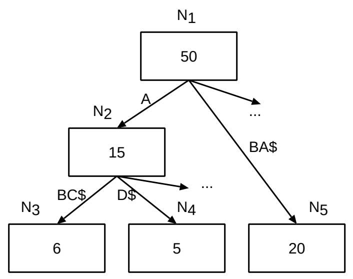
Suffix tree construction: From the set of M converted data sequences $\vec{X}_1, ..., \vec{X}_M$, we build a suffix tree using the incremental disk-based suffix tree construction algorithm [4]. Two suffix trees, representing two disjoint sets of data sequences, can be merged to produce a single suffix tree by pre-order traversal of both suffix trees and combining the paths corresponding to common subsequences. A suffix tree for a large set of data sequences can be constructed by performing a series of binary merges.

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*.

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. From the node N where the length of $label(PN, N)$ is L, we extract L rules.

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

where $label(PN, N)[p : q]$ is the subsequence of $label(PN, N)$ including elements in positions p through q, 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 4 shows the 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

Fig. 4. A part of a rule tree and rules extracted from a rule tree.

4 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. In this section, we present techniques to find matched rules and to derive the least relaxed rules.

4.1 Exact rule matching

The problem of 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 $\vec{q}$.

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.

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: RTI-E (node $N$, target head sequence $\vec{Q}$)
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($CN$, $\vec{Q}$);
```

Algorithm 1: Exact matching algorithm RTI-E

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)'.

4.2 Relaxed rule matching

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 $\vec{q}$.

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.

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 [12] where elements are assumed to have numeric values and their distances are measured by Euclidean distance metric.

Definition 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:*

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

$$d_{RE}(\vec{h}[1], \vec{q}[1]) = RE(ComNode(\vec{h}[1], \vec{q}[1])) - RE(Node(\vec{h}[1])).$$

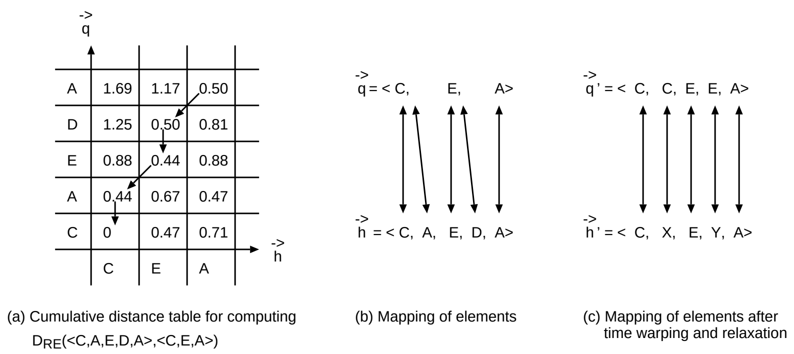
$RE(ComNode(\vec{h}[1], \vec{q}[1]))$ is the relaxation error of the lowest node containing both $\vec{h}[1]$ and $\vec{q}[1]$, and $RE(Node(\vec{h}[1]))$ is the relaxation error of the lowest node containing $\vec{h}[1]$. Therefore, $d_{RE}(\vec{h}[1], \vec{q}[1])$ is the relaxation error increase induced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

For example, let us consider the TAH shown in Figure 2. 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 $r(x, y)$ $(x = 1, 2, ..., |\vec{h}|, y = 1, 2, ..., |\vec{q}|)$.

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 $O(|\vec{h}||\vec{q}|)$. Figure 5 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 $D_{RE}(\vec{h}, \vec{q}) = 0.50$.

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 $\vec{Q}$ $(=C(S(\vec{q})))$ before beginning the search process. The algorithm maintains three global variables during its execution: the converted target head sequence $\vec{Q}$, 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.

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



(a) Cumulative distance table for computing D_RE(<C,A,E,D,A>,<C,E,A>)
(b) Mapping of elements
(c) Mapping of elements after time warping and relaxation
q = < C, E, A>
h = < C, A, E, D, A>
q' = < C, C, E, E, A>
h' = < C, X, E, Y, A>

A	1.69	1.17	0.50
D	1.25	0.50	0.81
E	0.88	0.44	0.88
A	0.44	0.67	0.47
C	0	0.47	0.71
	C	E	A

Fig. 5. 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.

```
Algorithm: RTI-S (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 from $newT$ and update $MinDist$;
    If further going-down the tree is necessary, call RTI-S($CN$, $newT$);
```

Algorithm 2: Similarity matching algorithm RTI-S

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 [11] and by allowing the cumulative distance table to be shared by rule heads that have the same prefix.

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 4.1. 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.

For example, 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'.

5 Experiments

To study the effectiveness of our proposed methods, we performed several experiments on the synthetic data sequences. 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. We used the relative minimum support value $RSUP_{min}$ to control the number of discovered rules.

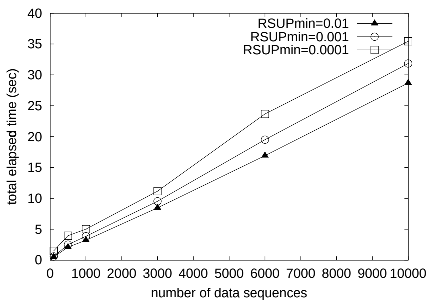


Fig. 6. Total elapsed time for discovering elastic rules with selected numbers of data sequences.

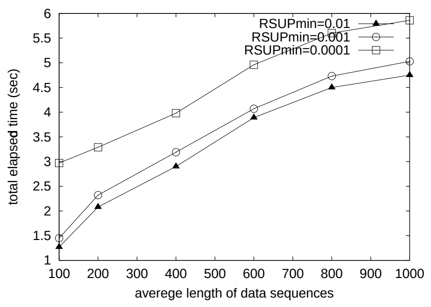


Fig. 7. Total elapsed time for discovering elastic rules with selected average length of data sequences.

5.1 Rule discovery

We used the total elapsed time required to discover rules as a performance measure of our rule discovery algorithm. First, we increased the number of data sequences from 100 to 10,000 while keeping the average length of data sequences 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 6 and 7, the total elapsed times increase linearly as the number of and the average length of data sequences grow. The figures also show that the linearity is maintained regardless of changing $RSUP_{min}$ values.

5.2 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.

Figure 8 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. Figure 9 shows the average search times of RTI-S and SS-based similarity matching algorithm. The performance gain of RTI-S increases as the number of rules increases.

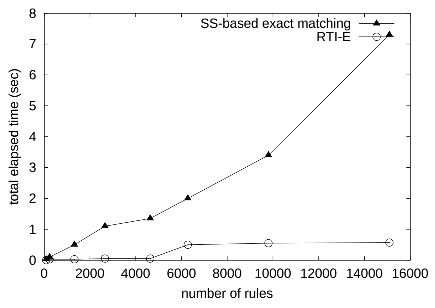


Fig. 8. Performance comparison between sequential scanning and RTI-E for exact rule matching.

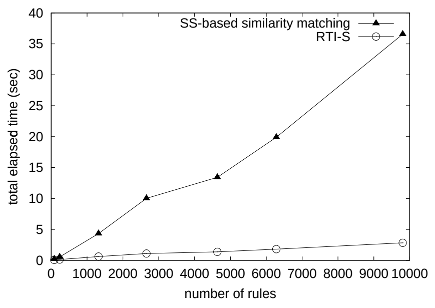


Fig. 9. Performance comparison between sequential scanning and RTI-S for relaxed rule matching.

6 Conclusion

In this paper, we have investigated the problems of discovering and matching elastic rules. The contributions of our works 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.

References

1. R. Agrawal, and R. Srikant, "Mining Sequential Patterns", *Proc. IEEE ICDE*, 1995.
2. D. J. Berndt, and J. Clifford, "Finding Patterns in Time Series: A Dynamic Programming Approach", *Advances in Knowledge Discovery and Data Mining*, AAAI/MIT, 1996.
3. P. S. Bradley, U. M. Fayyad, and O. L. Mangasarian, "Data Mining: Overview and Optimization Opportunities", *Microsoft Research Report MSR-TR-98-04*, 1998.
4. P. Bieganski, J. Riedl, and J. V. Carlis, "Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation", *Proc. Hawaii Int'l Conf. on System Sciences*, 1994.
5. W. W. Chu, and K. Chiang, "Abstraction of High Level Concepts from Numerical Values in Databases", *Proc. of AAAI Workshop on Knowledge Discovery in Databases*, 1994.
6. W. W. Chu, A. F. Cardenas, and R. K. Taira, "KMeD: a Knowledge-based Multimedia Medical Distributed Database System", *Information Systems, Vol.20, No.2*, Premagon-Press/Elsevier Science, 1995.
7. G. Das, K. Lin, H. Mannila, G. Renganathan, and P. Smyth, "Rule Discovery from Time Series", *Proc. International Conference on Knowledge Discovery and Data Mining*, 1998.
8. U. M. Fayyad, "Mining Databases: Toward Algorithms for Knowledge Discovery", *Data Engineering Bulletin 21(1)*, 1998.
9. H. Mannila, and H. Toivonen, "Discovering Generalized Episodes using Minimal Occurrences", *Proc. International Conference on Knowledge Discovery and Data Mining*, 1996.
10. H. Mannila, H. Toivonen, and A. I. Verkamo, "Discovering Frequent Episodes in Sequences", *Proc. International Conference on Knowledge Discovery and Data Mining*, 1995.
11. S. Park, W. W. Chu, J. Yoon, and C. Hsu, "Efficient Searches for Similar Subsequences of Different Lengths in Sequence Databases", *Proc. IEEE ICDE*, 2000.
12. L. Rabinar, and B. Juang. *Fundamentals of Speech Recognition*, Prentice Hall, 1993.
13. R. Srikant, and R. Agrawal, "Mining Sequential Patterns: Generalizations and Performance Improvements", *Proc. International Conference on Extending Database Technology*, 1996.
14. G. A. Stephen, *String Searching Algorithms*, World Scientific Publishing, 1994.
15. J. T.-L. Wang, G.-W. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang, "Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results", *Proc. ACM SIGMOD*, 1994.