Finding Best Patterns Practically

Ayumi Shinohara, Masayuki Takeda, Setsuo Arikawa,
Masahiro Hirao, Hiromasa Hoshino, and Shunsuke Inenaga

Department of Informatics, Kyushu University 33, Fukuoka 812-8581, JAPAN
{ayumi, hirao, hoshino, s-ine, takeda, arikawa}@i.kyushu-u.ac.jp

Abstract. Finding a pattern which separates two sets is a critical task in discovery. Given two sets of strings, consider the problem to find a subsequence that is common to one set but never appears in the other set. The problem is known to be NP-complete. Episode pattern is a generalized concept of subsequence pattern where the length of substring containing the subsequence is bounded. We generalize these problems to optimization problems, and give practical algorithms to solve them exactly. Our algorithms utilize some pruning heuristics based on the combinatorial properties of strings, and efficient data structures which recognize subsequence and episode patterns.

1 Introduction

In these days, a lot of text data or sequential data are available, and it is quite important to discover useful rules from these data. Finding a *good rule* to separate two given sets, often referred as *positive examples* and *negative examples*, is a critical task in Discovery Science as well as Machine Learning. String is one of the most fundamental structure to express and reserve information. In this paper, we review our recent work [7,8] which find a best pattern practically.

First we remind our motivations. Shimozono et al. [13] developed a machine discovery system BONSAI that produces a decision tree over regular patterns with alphabet indexing, from given positive set and negative set of strings. The core part of the system is to generate a decision tree which classifies positive examples and negative examples as correctly as possible. For that purpose, we have to find a *pattern* that maximizes the goodness according to the entropy information gain measure, recursively at each node of trees. In the current implementation, a pattern associated with each node is restricted to a *substring pattern*, due to the limit of computation time. One of our motivations of this study was to extend the BONSAI system to allow *subsequence patterns* as well as substring patterns at nodes, and accelerate the computation time.

However, there is a large gap between the complexity of finding the best *substring pattern* and *subsequence pattern*. Theoretically, the former problem can be solved in linear time, while the latter is NP-hard. In [7], we introduced a practical algorithm to find a best subsequence pattern that separates positive examples from negative examples, and showed some experimental results.

A drawback of subsequence patterns is that they are not suitable for classifying *long* strings over *small* alphabet, since a short subsequence pattern matches

S. Arikawa and A. Shinohara (Eds.): Progress in Discovery Science 2001, LNAI 2281, pp. 307–317.

with almost all long strings. Based on this observation, in [8] we considered *episode patterns*, which were originally introduced by Mannila et al. [10]. An episode pattern $\langle v, k \rangle$, where v is a string and k is an integer, *matches* with a string t if v is a subsequence for some substring u of t with $|u| \leq k$. Episode pattern is a generalization of subsequence pattern since subsequence pattern v is equivalent to episode pattern $\langle v, \infty \rangle$. We gave a practical solution to find a best episode pattern which separates a given set of strings from the other set of strings.

In this paper, we summarize our practical implementations of exact search algorithms that practically avoids exhaustive search. Since these problems are NP-hard, essentially we are forced to examine exponentially many candidate patterns in the worst case. Basically, for each pattern w, we have to count the number of strings that contain w as a subsequence in each of two sets. We call the task of counting the numbers as *answering subsequence query*. The computational cost to find the best subsequence pattern mainly comes from the total amount of time to answer these subsequence queries, since it is relatively heavy task if the sets are large, and many queries will be needed. In order to reduce the time, we have to either (1) ask queries as few as possible, or (2) speed up to answer queries. We attack the problem from both these two directions.

At first, we reduce the search space by appropriately pruning redundant branches that are guaranteed not to contain the best pattern. We use a heuristics inspired by Morishita and Sese [12], combined with some properties on the subsequence languages, and episode pattern languages.

Next, we accelerate answering for subsequence queries and episode pattern queries. Since the sets of strings are fixed in finding the best pattern, it is reasonable to preprocess the sets so that answering query for any pattern will be fast. We take an approach based on a deterministic finite automaton that accepts all subsequences of a string. Actually, we use subsequence automata for sets of strings, developed in [9] for subsequence query, and episode pattern recognizer for episode pattern query. These automata can answer quickly for subsequence query, at the cost of preprocessing time and space requirement to construct them.

2 Preliminaries

Let $\mathcal{N}$ be the set of integers. Let Σ be a finite *alphabet*, and let Σ^* be the set of all *strings* over Σ. For a string w, we denote by $|w|$ the length of w. For a set $S \subseteq \Sigma^*$ of strings, we denote by $|S|$ the number of strings in S, and by $||S||$ the total length of strings in S.

We say that a string v is a *prefix* (*substring*, *suffix*, resp.) of w if $w = vy$ ($w = xvy$, $w = xv$, resp.) for some strings $x, y \in \Sigma^*$. We say that a string v is a *subsequence* of a string w if v can be obtained by removing zero or more characters from w, and say that w is a *supersequence* of v. We denote by $v \preceq_{\rm str} w$ that v is a substring of w, and by $v \preceq_{\rm seq} w$ that v is a subsequence of w. For a string v, we define the *substring language* $L^{\rm str}(v)$ and *subsequence language*

$L^{\text{seq}}(v)$ as follows:

$$L^{\text{str}}(v) = \{w \in \Sigma^* \mid v \preceq_{\text{str}} w\}, \text{ and}$$
$$L^{\text{seq}}(v) = \{w \in \Sigma^* \mid v \preceq_{\text{seq}} w\}, \text{ respectively.}$$

An *episode pattern* is a pair of a string v and an integer k, and we define the *episode language* $L^{\text{eps}}(\langle v, k\rangle)$ by

$$L^{\text{eps}}(\langle v, k\rangle) = \{w \in \Sigma^* \mid {}^{\exists}u \preceq_{\text{str}} w \text{ such that } v \preceq_{\text{seq}} u \text{ and } |u| \leq k\}.$$

The following lemma is obvious from the definitions.

Lemma 1 ([7]). *For any strings $v, w \in \Sigma^*$,*

(1) if v is a prefix of w, then $v \preceq_{str} w$,
(2) if v is a suffix of w, then $v \preceq_{str} w$,
(3) if $v \preceq_{str} w$ then $v \preceq_{seq} w$,
(4) $v \preceq_{str} w$ if and only if $L^{str}(v) \supseteq L^{str}(w)$,
(5) $v \preceq_{seq} w$ if and only if $L^{seq}(v) \supseteq L^{seq}(w)$.

We formulate the problem by following our previous paper [7]. Readers should refer to [7] for basic idea behind this formulation. We say that a function f from $[0, x_{\max}] \times [0, y_{\max}]$ to real numbers is *conic* if

- for any $0 \leq y \leq y_{\max}$, there exists an x_1 such that
  - $f(x, y) \geq f(x', y)$ for any $0 \leq x < x' \leq x_1$, and
  - $f(x, y) \leq f(x', y)$ for any $x_1 \leq x < x' \leq x_{\max}$.
- for any $0 \leq x \leq x_{\max}$, there exists a y_1 such that
  - $f(x, y) \geq f(x, y')$ for any $0 \leq y < y' \leq y_1$, and
  - $f(x, y) \leq f(x, y')$ for any $y_1 \leq y < y' \leq y_{\max}$.

We assume that f is conic and can be evaluated in constant time in the sequel.

The following are the optimization problems to be tackled.

Definition 1 (Finding the best substring pattern according to f).
Input *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output *A string v that maximizes the value $f(x_v, y_v)$, where $x_v = |S \cap L^{str}(s)|$ and $y_s = |T \cap L^{str}(s)|$.*

Definition 2 (Finding the best subsequence pattern according to f).
Input *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output *A string v that maximizes the value $f(x_v, y_v)$, where $x_v = |S \cap L^{seq}(v)|$ and $y_v = |T \cap L^{seq}(v)|$.*

Definition 3 (Finding the best episode pattern according to f). .
Input *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output *A episode pattern $\langle v, k\rangle$ that maximizes the value $f(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})$, where $x_{\langle v,k\rangle} = |S \cap L^{eps}(\langle v, k\rangle)|$ and $y_{\langle v,k\rangle} = |T \cap L^{eps}(\langle v, k\rangle)|$.*

```
1  pattern FindMaxPattern(StringSet S, T)
2      maxVal = −∞;
3      for all possible pattern π do
4          x = |S ∩ L(π)|;
5          y = |T ∩ L(π)|;
6          val = f(x, y);
7          if val > maxVal then
8              maxVal = val;
9              maxPat = π;
10     return maxPat;
```

Fig. 1. Exhaustive search algorithm.

We remind that the first problem can be solved in linear time [7], while the latter two problems are NP-hard.

We review the basic idea of our algorithms. Fig. 1 shows a naive algorithm which exhaustively examines and evaluate all possible patterns one by one, and returns the best pattern that gives the maximum value. The most time consuming part is obviously the lines 4 and 5, and in order to reduce the search time, we should (1) reduce the possible patterns in line 3 *dynamically* by using some appropriate pruning method, and (2) speed up to compute $|S \cap L(\pi)|$ and $|T \cap L(\pi)|$ for each π. In Section 3, we deal with (1), and in Section 4, we treat (2).

3 Pruning Heuristics

In this section, we introduce some pruning heuristics, inspired by Morishita and Sese [12].

For a function $f(x, y)$, we denote $F(x, y) = \max\{f(x,y), f(x,0), f(0,y), f(0,0)\}$. From the definition of conic function, we can prove the following lemma.

Lemma 2. *For any patterns v and w with $L(v) \supseteq L(w)$, we have*

$$f(x_w, y_w) \leq F(x_v, y_v).$$

3.1 Subsequence Patterns

We consider finding subsequence pattern in this subsection. By Lemma 1 (5) and Lemma 2, we have the following lemma.

Lemma 3 ([7]). *For any strings $v, w \in \Sigma^*$ with $v \preceq_{seq} w$, we have*

$$f(x_w, y_w) \leq F(x_v, y_v).$$

In Fig. 2, we show our algorithm to find the best subsequence pattern from given two sets of strings, according to the function f. Optionally, we can specify the maximum length of subsequences. We use the following data structures in the algorithm.

```
1   string FindMaxSubsequence(StringSet S, T, int maxLength = ∞)
2       string prefix, seq, maxSeq;
3       double upperBound = ∞, maxVal = −∞, val;
4       int x, y;
5       PriorityQueue queue;    /* Best First Search*/
6       queue.push("", ∞);
7       while not queue.empty() do
8           (prefix, upperBound) = queue.pop();
9           if upperBound < maxVal then break;
10          foreach c ∈ Σ do
11              seq= prefix+ c;    /* string concatenation */
12              x = S.numOfSubseq(seq);
13              y = T.numOfSubseq(seq);
14              val = f(x, y);
15              if val > maxVal then
16                  maxVal = val;
17                  maxSeq = seq;
18*             upperBound = F(x, y);
19              if |seq| < maxLength then
20                  queue.push(seq, upperBound);
21      return maxSeq;
```

Fig. 2. Algorithm *FindMaxSubsequence*.

StringSet Maintain a set S of strings.

- **int** *numOfSubseq*(**string** *seq*) : return the cardinality of the set $\{w \in S \mid seq \preceq_{\text{seq}} w\}$.

PriorityQueue Maintain strings with their priorities.

- **bool** *empty*() : return **true** if the queue is empty.
- **void** *push*(**string** w, **double** *priority*) : push a string w into the queue with priority *priority*.
- **(string, double)** *pop*() : pop and return a pair (*string*, *priority*), where *priority* is the highest in the queue.

The next theorem guarantees the completeness of the algorithm.

Theorem 1 ([7]). *Let S and T be sets of strings, and ℓ be a positive integer. The algorithm FindMaxSubsequence(S, T, ℓ) will return a string w that maximizes the value $f(x_v, y_v)$ among the strings of length at most ℓ, where $x_v = |S \cap L^{str}(s)|$ and $y_s = |T \cap L^{str}(s)|$.*

Proof. We first consider the case that the lines 18 is removed. Since the value of *upperBound* is unchanged, **PriorityQueue** is actually equivalent to a simple queue. Then, the algorithm performs the exhaustive search in a breadth first manner. Thus the algorithm will compute the value $f(x_v, y_v)$ for *all* strings of length at most *maxLength*, in increasing order of the length, and it can find the best pattern trivially.

We now focus on the line 9, by assuming the condition $upperBound < maxVal$ holds. Since the *queue* is a priority queue, we have $F(x_v, y_v) \leq upperBound$ for any string v in the queue. By Lemma 3, $f(x_v, y_v) \leq F(x_v, y_v)$, which implies $f(x_v, y_v) < maxVal$. Thus no string in the queue can be the best subsequence and we jump out of the loop immediately.

Next, we consider the lines 18. Let v be the string currently represented by the variable *seq*. At lines 12 and 13, x_v and y_v are computed. At line 18, $upperBound = F(x_v, y_v)$ is evaluated, and if *upperBound* is less than the current maximum value *maxVal*, v is not pushed into *queue*. It means that any string w of which v is a prefix will not be evaluated. We can show that such a string w can never be the best subsequence as follows. Since v is a prefix of w, we know v is a subsequence of w, by Lemma 1 (1) and (3). By Lemma 3, we know $f(x_w, y_w) \leq F(x_v, y_v)$, and since $F(x_v, y_v) < maxVal$, the string w can never be the maximum. □

3.2 Episode Pattern

We now show a practical algorithm to find the best episode patterns. We should remark that the search space of episode patterns is $\Sigma^* \times \mathcal{N}$, while the search space of subsequence patterns was Σ^*. A straight-forward approach based on the last subsection might be as follows. First we observe that the algorithm *FindMaxSubsequence* in Fig. 2 can be easily modified to find the best episode pattern $\langle v, k \rangle$ *for any fixed threshold* k: we have only to replace the lines 12 and 13 so that they compute the numbers of strings in S and T that match with the episode pattern $\langle seq, k \rangle$, respectively. Thus, for each possible threshold value k, repeat his algorithm, and get the maximum. A short consideration reveals that we have only to consider the threshold values up to l, that is the length of the longest string in given S and T.

However, here we give a more efficient solution. Let us consider the following problem, that is a subproblem of *finding the best episode pattern* in Definition 3.

Definition 4 (Finding the best threshold value).
Input *Two sets* $S, T \subseteq \Sigma^*$ *of strings, and a string* $v \in \Sigma^*$.
Output *Integer* k *that maximizes the value* $f(x_{\langle v,k \rangle}, y_{\langle v,k \rangle})$, *where* $x_{\langle v,k \rangle} = |S \cap L^{eps}(\langle v, k \rangle)|$ *and* $y_{\langle v,k \rangle} = |T \cap L^{eps}(\langle v, k \rangle)|$.

The next lemma give a basic containment of episode pattern languages.

Lemma 4 ([8]). *For any two episode patterns* $\langle v, l \rangle$ *and* $\langle w, k \rangle$, *if* $v \preceq_{seq} w$ *and* $l \geq k$ *then* $L^{eps}(\langle v, l \rangle) \supseteq L^{eps}(\langle w, k \rangle)$.

By Lemma 2 and 4, we have the next lemma.

Lemma 5 ([8]). *For any two episode patterns* $\langle v, l \rangle$ *and* $\langle w, k \rangle$, *if* $v \preceq_{seq} w$ *and* $l \geq k$ *then* $f(x_{\langle w,k \rangle}, y_{\langle w,k \rangle}) \leq F(x_{\langle v,l \rangle}, y_{\langle v,l \rangle})$.

For strings $v, s \in \Sigma^*$, we define the *threshold value* θ of v for s by $\theta = min\{k \in \mathcal{N} \mid s \in L^{\mathrm{eps}}(\langle v, k\rangle)\}$. If no such value, let $\theta = \infty$. Note that $s \notin L^{\mathrm{eps}}(\langle v, k\rangle)$ for any $k < \theta$, and $s \in L^{\mathrm{eps}}(\langle v, k\rangle)$ for any $k \geq \theta$. For a set S of strings and a string v, let us denote by $\Theta_{S,v}$ the set of threshold values of v for some $s \in S$.

A key observation is that a best threshold value for given $S, T \subseteq \Sigma^*$ and a string $v \in \Sigma^*$ can be found in $\Theta_{S,v} \cup \Theta_{T,v}$ without loss of generality. Thus we can restrict the search space of the best threshold values to $\Theta_{S,v} \cup \Theta_{T,v}$.

From now on, we consider the numerical sequence $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$. (We will treat $\{y_{\langle v,k\rangle}\}_{k=0}^{\infty}$ in the same way.) It clearly follows from Lemma 4 that the sequence is non-decreasing. Remark that $0 \leq x_{\langle v,k\rangle} \leq |S|$ for any k. Moreover, $x_{\langle v,l\rangle} = x_{\langle v,l+1\rangle} = x_{\langle v,l+2\rangle} = \cdots$, where l is the length of the longest string in S. Hence, we can represent $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$ with a list having at most $\min\{|S|, l\}$ elements. We call this list *a compact representation of the sequence* $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$ (*CRS*, for short).

We show how to compute CRS for each v and a fixed S. Observe that $x_{\langle v,k\rangle}$ increases only at the threshold values in $\Theta_{S,v}$. By computing a sorted list of all threshold values in $\Theta_{S,v}$, we can construct the CRS of $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$. If using the counting sort, we can compute the CRS for any $v \in \Sigma^*$ in $O(|S|ml + |S|) = O(||S||m)$ time, where $m = |v|$.

We emphasize that the time complexity of computing the CRS of $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$ is the same as that of computing $x_{\langle v,k\rangle}$ for a single k $(0 \leq k \leq \infty)$, by our method.

After constructing CRSs $\bar{x}$ of $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$ and $\bar{y}$ of $\{y_{\langle v,k\rangle}\}_{k=0}^{\infty}$, we can compute the best threshold value in $O(|\bar{x}| + |\bar{y}|)$ time. Thus we have the following, which gives an efficient solution to the finding the best threshold value problem.

Lemma 6. *Given $S, T \subseteq \Sigma^*$ and $v \in \Sigma^*$, we can find the best threshold value in $O(\,(||S|| + ||T||)\cdot|v|\,)$ time.*

By substituting this procedure into the algorithm *FindMaxSubsequence*, we get an algorithm to find a best episode pattern from given two sets of strings, according to the function f, shown in Fig. 3. We add a method *crs*(v) to the data structure **StringSet** that returns CRS of $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$, as mentioned above.

By Lemma 5, we can use the value *upperBound* $= F(x_{v,\infty}, y_{v,\infty})$ to prune branches in the search tree computed at line 20 marked by (*). We emphasize that the value $F(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})$ is insufficient as *upperBound*. Note also that $x_{\langle v,\infty\rangle}$ and $y_{\langle v,\infty\rangle}$ can be extracted from $\bar{x}$ and $\bar{y}$ in constant time, respectively. The next theorem guarantees the completeness of the algorithm.

Theorem 2 ([8]). *Let S and T be sets of strings, and ℓ be a positive integer. The algorithm FindBestEpisode(S, T, ℓ) will return an episode pattern that maximizes $f(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})$, with $x_{\langle v,k\rangle} = |S \cap L^{eps}(\langle v, k\rangle)|$ and $y_{\langle v,k\rangle} = |T \cap L^{eps}(\langle v, k\rangle)|$, where v varies any string of length at most ℓ and k varies any integer.*

```
string FindBestEpisode(StringSet S, T, int ℓ)
    string prefix, v;
    episodePattern maxSeq; /* pair of string and int */
    double upperBound = ∞, maxVal = −∞, val;
    int k′;
    CompactRepr x̄, ȳ; /* CRS */
    PriorityQueue queue;    /* Best First Search*/
    queue.push("", ∞);
    while not queue.empty() do
        (prefix, upperBound) = queue.pop();
        if upperBound < maxVal then break;
        foreach c ∈ Σ do
            v = prefix+ c;    /* string concatenation */
            x̄ = S.crs(v);
            ȳ = T.crs(v);
            k′ = argmax_k{f(x_⟨v,k⟩, y_⟨v,k⟩)} and val = f(x_⟨v,k′⟩, y_⟨v,k′⟩);
            if val > maxVal then
                maxVal = val;
                maxEpisode = ⟨v, k′⟩;
(*)         upperBound = F(x_⟨v,∞⟩, y_⟨v,∞⟩);
            if upperBound > maxVal and |v| < ℓ then
                queue.push(v, upperBound);
    return maxEpisode;
```

Fig. 3. Algorithm *FindBestEpisode*.

4 Using Efficient Data Structures

We introduces some efficient data structures to speed up answering the queries.

4.1 Subsequence Automata

First we pay our attention to the following problem.

Definition 5 (Counting the matched strings).
Input *A finite set* $S \subseteq \Sigma^*$ *of strings.*
Query *A string* $seq \in \Sigma^*$*.*
Answer *The cardinality of the set* $S \cap L^{seq}(seq)$*.*

Of course, the answer to the query should be very fast, since many queries will arise. Thus, we should preprocess the input in order to answer the query quickly. On the other hand, the preprocessing time is also a critical factor in some applications. In this paper, we utilize automata that accept subsequences of strings.

In [9], we considered a subsequence automaton as a deterministic complete finite automaton that recognizes all possible subsequences of a set of strings, that is essentially the same as the directed acyclic subsequence graph (DASG) introduced by Baeza-Yates [2]. We showed an online construction of subsequence

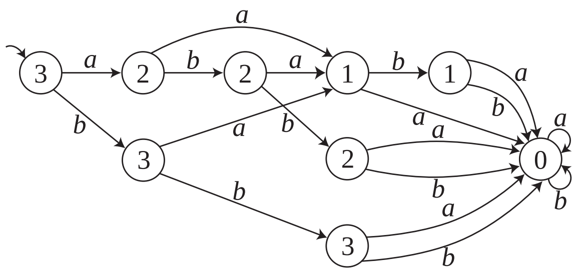

Fig. 4. Subsequence automaton for $S = \{abab, abb, bb\}$, where $\Sigma = \{a, b\}$. Each number on a state denotes the number of matched strings. For example, by traverse the states according to a string ab, we reach the state whose number is 2. It corresponds to the cardinality $|L^{\mathrm{seq}}(ab) \cap S| = 2$, since $ab \preceq_{\mathrm{seq}} abab$, $ab \preceq_{\mathrm{seq}} abb$ and $ab \not\preceq_{\mathrm{seq}} bb$.

automaton for a set of strings. Our algorithm runs in $O(|\Sigma|(m + k) + N)$ time using $O(|\Sigma|m)$ space, where $|\Sigma|$ is the size of alphabet, N is the total length of strings, and m is the number of states of the resulting subsequence automaton. We can extend the automaton so that it answers the above *Counting the matched strings* problem in a natural way (see Fig. 4).

Although the construction time is linear to the size m of automaton to be built, unfortunately $m = O(n^k)$ in general, where we assume that the set S consists of k strings of length n. (The lower bound of m is only known for the case $k = 2$, as $m = \Omega(n^2)$ [4].) Thus, when the construction time is also a critical factor, as in our application, it may not be a good idea to construct subsequence automaton for the set S itself. Here, for a specified parameter $mode > 0$, we partition the set S into $d = k/mode$ subsets $S_1, S_2, \ldots, S_d$ of at most $mode$ strings, and construct d subsequence automata for each S_i. When asking a query seq, we have only to traverse all automata simultaneously, and return the sum of the answers. In this way, we can balance the preprocessing time with the total time to answer (possibly many) queries. In [7], we experimentally evaluated the optimal value of the parameter $mode$.

4.2 Episode Directed Acyclic Subsequence Graphs

We now analyze the complexity of *episode pattern matching*. Given an episode pattern $\langle v, k\rangle$ and a string t, determine whether $t \in L^{\mathrm{eps}}(\langle v, k\rangle)$ or not. This problem can be answered by filling up the edit distance table between v and t, where only insertion operation with cost one is allowed. It takes $\Theta(mn)$ time and space using a standard dynamic programming method, where $m = |v|$ and $n = |t|$. For a fixed string, automata-based approach is useful. We use the Episode Directed Acyclic Subsequence Graph (EDASG) for string t, which was recently introduced by Troíček in [14]. Hereafter, let $EDASG(t)$ denote the EDASG for t. With the use of $EDASG(t)$, episode pattern matching can be answered quickly in practice, although the worst case behavior is still $O(mn)$. EDASG(t) is also useful to compute the threshold value θ of given v for t quickly in practice. As an

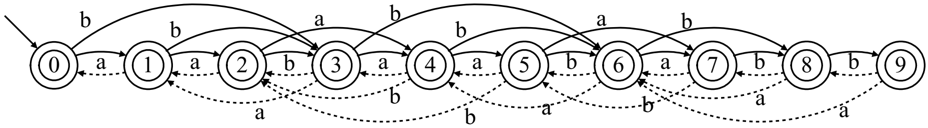

Fig. 5. *EDASG*(t), where $t = aabaababb$. Solid arrows denote the forward edges, and broken arrows denote the backward edges. The number in each circle denotes the state number.

example, *EDASG*$(aabaababb)$ is shown in Fig. 5. When examining if an episode pattern $\langle abb, 4\rangle$ matches with t or not, we start from the initial state 0 and arrive at state 6, by traversing the forward edges spelling abb. It means that the shortest prefix of t that contains abb as a subsequences is $t[0:6] = aabaab$, where $t[i:j]$ denotes the substring $t_{i+1} \ldots t_j$ of t. Moreover, the difference between the state numbers 6 and 0 corresponds to the length of matched substring $aabaab$ of t, that is, $6 - 0 = |aabaab|$. Since it exceeds the threshold 4, we move backwards spelling bba and reach state 1. It means that the shortest suffix of $t[0:6]$ that contains abb as a subsequence is $t[1:6] = abaab$. Since $6 - 1 > 4$, we have to examine other possibilities. It is not hard to see that we have only to consider the string $t[2:*]$. Thus we continue the same traversal started from state 2, that is the next state of state 1. By forward traversal spelling abb, we reach state 8, and then backward traversal spelling bba bring us to state 4. In this time, we found the matched substring $t[4:8] = abab$ which contains the subsequence abb, and the length $8 - 4 = 4$ satisfies the threshold. Therefore we report the occurrence and terminate the procedure.

It is not difficult to see that the EDASGs are useful to compute the threshold value of v for a fixed t. We have only to repeat the above forward and backward traversal up to the end, and return the minimum length of the matched substrings. Although the time complexity is still $\Theta(mn)$, practical behavior is usually better than using standard dynamic programming method.

5 Conclusion

In this paper, we focused on *finding the best* subsequence pattern and episode patterns. However, we can easily extend our algorithm to *enumerate all strings* whose values of the objective function exceed the given threshold, since essentially we examine all strings, with effective pruning heuristics. Enumeration may be more preferable in the context of *text data mining* [3,5,15].

It is challenging to apply our approach to find the best *pattern* in the sense of *pattern languages* introduced by Angluin [1], where the related consistency problems are shown to be very hard [11]. Hamuro *et al.* [6] implemented our algorithm for finding best subsequences, and reported a quite successful experiments on business data. We are now in the process of installing our algorithms into the core of the decision tree generator in the BONSAI system [13].

References

1. D. Angluin. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21(1):46–62, Aug. 1980.
2. R. A. Baeza-Yates. Searching subsequences. *Theoretical Computer Science*, 78(2):363–376, Jan. 1991.
3. A. Califano. SPLASH: Structural pattern localization analysis by sequential histograms. *Bioinformatics*, Feb. 1999.
4. M. Crochemore and Z. Troníček. Directed acyclic subsequence graph for multiple texts. Technical Report IGM-99-13, Institut Gaspard-Monge, June 1999.
5. R. Feldman, Y. Aumann, A. Amir, A. Zilberstein, and W. Klosgen. Maximal association rules: A new tool for mining for keyword co-occurrences in document collections. In *Proc. of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 167–170. AAAI Press, Aug. 1997.
6. Y. Hamuro, H. Kawata, N. Katoh, and K. Yada. A machine learning algorithm for analyzing string patterns helps to discover simple and interpretable business rules from purchase history. In *Progress in Discovery Science*, LNCS, 2002. (In this volume).
7. M. Hirao, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best subsequence patterns. In *Proc. of The Third International Conference on Discovery Science*, volume 1967 of *Lecture Notes in Artificial Intelligence*, pages 141–154. Springer-Verlag, Dec. 2000.
8. M. Hirao, S. Inenaga, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best episode patterns. In *Proc. of The Fourth International Conference on Discovery Science*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Nov. 2001.
9. H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. Online construction of subsequence automata for multiple texts. In *Proc. of 7th International Symposium on String Processing and Information Retrieval*. IEEE Computer Society, Sept. 2000. (to appear).
10. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episode in sequences. In U. M. Fayyad and R. Uthurusamy, editors, *Proc. of the 1st International Conference on Knowledge Discovery and Data Mining*, pages 210–215. AAAI Press, Aug. 1995.
11. S. Miyano, A. Shinohara, and T. Shinohara. Polynomial-time learning of elementary formal systems. *New Generation Computing*, 18:217–242, 2000.
12. S. Morishita and J. Sese. Traversing itemset lattices with statistical metric pruning. In *Proc. of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 226–236. ACM Press, May 2000.
13. S. Shimozono, A. Shinohara, T. Shinohara, S. Miyano, S. Kuhara, and S. Arikawa. Knowledge acquisition from amino acid sequences by machine learning system BONSAI. *Transactions of Information Processing Society of Japan*, 35(10):2009–2018, Oct. 1994.
14. Z. Troníček. Episode matching. In *Proc. of 12th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science. Springer-Verlag, July 2001. (to appear).
15. J. T. L. Wang, G.-W. Chirn, T. G. Marr, B. A. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 115–125. ACM Press, May 1994.