String Pattern Discovery

Ayumi Shinohara

Department of Informatics, Kyushu University 33, Fukuoka 812-8581, JAPAN
PRESTO, Japan Science and Technology Agency
ayumi@i.kyushu-u.ac.jp

Abstract. Finding a good pattern which discriminates one set of strings from the other set is a critical task in knowledge discovery. In this paper, we review a series of our works concerning with the string pattern discovery. It includes theoretical analyses of learnabilities of some pattern classes, as well as development of practical data structures which support efficient string processing.

1 Introduction

A huge amount of text data or sequential data are accessible in these days. Especially, the growing popularity of Internet have caused an enormous increase of text data in the last decade. Moreover, a lot of biological sequences are also available due to various genome sequencing projects. Many of these data are stored as raw strings, or in semi-structured form such as HTML and XML, which are essentially strings. *String pattern discovery*, where one is interested in extracting patterns which characterizes a set of strings or sequential data, has attracted widespread attentions [1,36,13,24,12,3,4,30]. Discovering a *good rule* to separate two given sets, often referred as *positive examples* and *negative examples*, is a critical task in Machine Learning and Knowledge Discovery. In this paper, we review a series of our works for finding best string patterns efficiently, together with their theoretical background.

Our motivations originated in the development of a machine discovery system BONSAI [31], 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 initial implementation, a pattern associated with each node is restricted to a *substring pattern*, due to the limit of computation time. In order to allow more expressive patterns while keeping the computation in reasonable time, we have introduced various techniques gradually [15,17,16,7,19,8,21,32,6,18]. Essentially, they are combinations of pruning heuristics in the huge search space without sacrificing the optimality of the solution, and efficient data structures which support various string processing.

S. Ben-David, J. Case, A. Maruoka (Eds.): ALT 2004, LNAI 3244, pp. 1–13, 2004.
© Springer-Verlag Berlin Heidelberg 2004

In this paper, we describe the most fundamental ideas of these works, by focusing only on substring patterns, subsequence patterns, and episode patterns, as the target patterns to be discovered. We also show their learnabilities in the probably approximately correct (PAC) learning model, which are the background theory of our approach.

2 Preliminaries

For a finite *alphabet* Σ, 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. Let $\mathcal{N}$ be the set of natural numbers.

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.

A *pattern class* is a pair $\mathcal{C} = (\Pi, m)$, where Π is a set called the *pattern set* and $m : \Sigma^* \times \Pi \to \{0, 1\}$ is the *pattern matching function*. An element $p \in \Pi$ is called a *pattern*. For a pattern $p \in \Pi$ and string $w \in \Sigma^*$, we say p *of class* $\mathcal{C}$ *matches* w iff $m(w, p) = 1$. For a pattern p, we denote by $L_{\mathcal{C}}(p) = \{w \in \Sigma^* \mid m(w, p) = 1\}$ the set of strings in Σ^* which p of class $\mathcal{C}$ matches.

In this paper, we focus on the following pattern classes, and their languages.

Definition 1. *The* substring pattern class *is defined to be a pair* $(\Sigma^*, substr)$ *where* $substr(w, p) = 1$ *iff* p *is a substring of* w. *The* subsequence pattern class *is a pair* $(\Sigma^*, subseq)$ *where* $subseq(w, p) = 1$ *iff* p *is a subsequence of* w. *The* episode pattern class *is a pair* $(\Sigma^* \times \mathcal{N}, epis)$ *where* $epis(w, \langle p, k\rangle) = 1$ *iff* p *is a subsequence of some substring* v *of* w *such that* $|v| \leq k$.

Finally, the substring (subsequence, episode, reps.) pattern language, denoted by $L^{str}(p)$ ($L^{seq}(p)$, $L^{eps}(\langle p, k\rangle)$, *resp.), is a language defined by substring (subsequence, episode, resp.) pattern class.*

Remark that the substring pattern languages is a subclass of the episode pattern class, since $L^{\mathrm{str}}(p) = L^{\mathrm{eps}}(\langle p, |p|\rangle)$ for any $p \in \Sigma^*$. Subsequence pattern languages is also an subclass of the episode pattern class, since $L^{\mathrm{seq}}(p) = L^{\mathrm{eps}}(\langle p, \infty\rangle)$ for any $p \in \Sigma^*$.

3 PAC-Learnability

Valiant [35] introduced the *PAC-leaning model* as a formal model of concept learning from examples. An excellent textbook on this topic is written by Kearns and Vazirani [23]. This section briefly summarizes the PAC-learnability of the languages we defined in the last section.

For a pattern class $\mathcal{C} = (\Pi, m)$, a pair $\langle w, m(w, p)\rangle$ is called an *example* of a pattern $p \in \Pi$ for $w \in \Sigma^*$. It is called a *positive example* if $m(w, p) = 1$) and is

called a *negative example* otherwise. For an alphabet Σ and an integer $n \geq 0$, we denote by $\Sigma^{[n]}$ the set $\{w \in \Sigma^* : |w| \leq n\}$.

Definition 2. *A pattern class $\mathcal{C} = (\Pi, m)$ is* polynomial-time learnable *if there exist an algorithm $\mathcal{A}$ and a polynomial poly$(\cdot,\cdot,\cdot)$ which satisfy the following conditions for any pattern $p \in \Pi$, any real numbers ε, δ $(0 < \varepsilon, \delta < 1)$, any integer $n \geq 0$, and any probability distribution P on $\Sigma^{[n]}$:*

(a) $\mathcal{A}$ takes ε, δ, and n, as inputs.
(b) $\mathcal{A}$ may call EXAMPLE*, which generates examples of the pattern $p \in \Pi$, randomly according to the probability distribution P on $\Sigma^{[n]}$.*
(c) $\mathcal{A}$ outputs a pattern $q \in \Pi$ satisfying $P(L_{\mathcal{C}}(p) \cup L_{\mathcal{C}}(q) - L_{\mathcal{C}}(p) \cap L_{\mathcal{C}}(q)) < \varepsilon$ with probability at least $1 - \delta$.
(d) The running time of $\mathcal{A}$ is bounded by poly$(1/\varepsilon, 1/\delta, n)$.

The PAC-learnability is well-characterized in terms of *Vapnik-Chervonenkis dimension* [10] and the existence of *Occam algorithm* [9,11].

Theorem 1 ([10,29]). *A pattern class $\mathcal{C} = (\Pi, m)$ is polynomial-time learnable if the following conditions hold.*

(a) $\mathcal{C}$ is polynomial dimension*, i.e., there exists a polynomial $d(n)$ such that $|\{L_{\mathcal{C}}(p) \cap \Sigma^{[n]} : p \in \Pi\}| \leq 2^{d(n)}$.*
(b) There exists a polynomial-time algorithm called polynomial-time hypothesis finder *for $\mathcal{C}$ which produces a hypothesis from a sequence of examples such that it is consistent with the given examples.*

On the other hand, the pattern class $\mathcal{C}$ is polynomial-time learnable only if there exists a randomized polynomial-time hypothesis finder *for $\mathcal{C}$.*

It is not hard to verify that all of substring languages, subsequence languages, and episode pattern languages are of polynomial dimension. Therefore the problem is whether there exists a (randomized) polynomial-time hypothesis finder for these classes. For substring languages, we can easily construct a polynomial-time hypothesis finder for it, since the candidate patterns must be a substring of any positive examples, so that we have only to consider quadratic numbers of candidates. Indeed, we can develop a linear-time hypothesis finder for it [15], by utilizing the data structure called *Generalized Suffix Tree*. See also our recent generalization of it for finding *pairs* of substring patterns [6]. On the other hand, the consistency problem for subsequence languages language is **NP**-hard [26,22, 27]. Thus we have the following theorem.

Theorem 2. *The substring languages is polynomial-time PAC-learnable. On the other hand, subsequence languages nor episode languages is not polynomial-time PAC-learnable under the assumption* **RP**$\neq$**NP**.

Query learning model due to Angluin [2], and identification in the limit due to Gold [14] are also important learning models. We discussed in [25], the learnabilities of finite unions of subsequence languages in these models.

4 Finding Best Patterns Efficiently

From a practical viewpoint, we have to find a good pattern which discriminate positive examples from negative examples. We formulate the problem by following our paper [15]. Let *good* be a function from $\Pi^* \times 2^{\Sigma^*} \times 2^{\Sigma^*}$ to the set of real numbers. We formulate the problem of finding the best pattern according to the function *good* as follows.

Definition 3 (Finding the best pattern in $\mathcal{C} = (\Pi, m)$ according to *good*).

Input: *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output: *A pattern $p \in \Pi$ that maximizes the value $good(p, S, T)$.*

Intuitively, the value $good(p, S, T)$ expresses the goodness to distinguish S from T using the rule specified by the pattern p. We may choose an appropriate function *good* according to each applications. For example, the χ^2 values, entropy information gain, and gini index are frequently used. Essentially these statistical measures are defined by the numbers of strings that satisfy the rule specified by p. Thus we can describe the measure in the following form:

$$good(p, S, T) = f(x_p, y_p, |S|, |T|),$$

where $x_p = |S \cap L_{\mathcal{C}}(p)|$ and $y_p = |T \cap L_{\mathcal{C}}(p)|$. When the sets S and T are fixed, the values $x_{\max} = |S|$ and $y_{\max} = |T|$ are unchanged. Thus we abbreviate the function $f(x, y, x_{\max}, y_{\max})$ to $f(x, y)$ in the sequel.

Since the function $good(p, S, T)$ expresses the goodness of a pattern $p \in \Pi$ to distinguish two sets, it is natural to assume that the function f satisfies the *conicality*, defined as follows.

Definition 4. *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 $y_1 \leq y < y' \leq y_{\max}$.*

Actually, all of the above statistical measures are conic. We remark that any convex function is conic. We assume that f is conic and can be evaluated in constant time in the sequel.

We now describe the basic idea of our algorithms. Fig. 1 shows a naive algorithm which exhaustively searches all possible patterns one by one, and returns the best pattern that gives the maximum score. Since most time consuming part is obviously the lines 5 and 6, in order to reduce the search time, we should (1) reduce the possible patterns in line 3 *dynamically* by using some appropriate pruning heuristics, and (2) speed up to computing $|S \cap L_{\mathcal{C}}(p)|$ and $|T \cap L_{\mathcal{C}}(p)|$ for each pattern p. We deal with (1) in the next section, and we treat (2) in Section 7.

```
1  pattern FindBestPattern(StringSet S, T)
2      double maxVal = −∞;
3      pattern maxPat = null;
4      for all possible pattern p ∈ Π do
5          x = |S ∩ L_C(p)|;
6          y = |T ∩ L_C(p)|;
7          val = f(x, y);
8          if val > maxVal then
9              maxVal = val;
10             maxPat = p;
11     return maxPat;
```

Fig. 1. Exhaustive search algorithm for finding the best pattern in $\mathcal{C} = (\Pi, m)$

5 Pruning Heuristics

In this section, we introduce some pruning heuristics, inspired by Morishita and Sese [28], to construct a practical algorithm to find the best subsequence pattern and the best episode pattern, without sacrificing the optimality of the solution.

Lemma 1 ([15]). *For any patterns $p, q \in \Pi$ with $L_{\mathcal{C}}(p) \supseteq L_{\mathcal{C}}(q)$, we have*

$$f(x_q, y_q) \leq \max\{f(x_p, y_p), f(x_p, 0), f(0, y_p), f(0, 0)\}.$$

Lemma 2 ([15,16]). *For any subsequence patterns $p, q \in \Sigma^*$ such that p is a subsequence of q, we have $L^{seq}(p) \supseteq L^{seq}(q)$. Moreover, for $l \geq k$, we have $L^{eps}(\langle p, l\rangle) \supseteq L^{eps}(\langle q, k\rangle)$.*

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.

StringSet Maintain a set S of strings.
- **int** *numOfSubseq*(**string** p) : return the cardinality of the set $\{w \in S \mid p$ is a subsequence of $w\}$.

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 3 ([15]). *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 good(w, S, T) among the strings of length at most ℓ.*

```
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 = max{f(x, y), f(x, 0), f(0, y), f(0, 0)};
19              if |seq| < maxLength then
20                  queue.push(seq, upperBound);
21      return maxSeq;
```

Fig. 2. Algorithm *FindMaxSubsequence*.

6 Finding Best Threshold Values

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. We consider the following problem, that is a subproblem of *finding the best episode pattern*.

Definition 5 (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* $y_{\langle v,k\rangle} = |T \cap L^{eps}(\langle v, k\rangle)|$.

For strings $v, s \in \Sigma^*$, we define the *threshold value* θ of v for s by $\theta = min\{k \in \mathcal{N} \mid s \in L^{\text{eps}}(\langle v, k\rangle)\}$. If no such value, let $\theta = \infty$. Note that $s \notin L^{\text{eps}}(\langle v, k\rangle)$ for any $k < \theta$, and $s \in L^{\text{eps}}(\langle v, k\rangle)$ for any $\theta \leq k$. 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 follows from Lemma 2 that the sequence is non-decreasing. Moreover, 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 of v for some $s \in S$. By computing a sorted list of threshold values of v for all $s \in S$, 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 $v \in \Sigma^*$ 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 give an efficient solution to the finding the best threshold value problem.

Lemma 3. *Given $S, T \subseteq \Sigma^*$ and $v \in \Sigma^*$, we can finding the best threshold value in $O((||S|| + ||T||)|v|)$ time, where $||S||$ and $||T||$ represent the total length of the strings in S and T, respectively.*

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 $\{x_{\langle v,k\rangle}\}_{k=0}^{\infty}$, mentioned above.

By Lemma 1 and 2, we can use the value *upperBound* at $(x_{\langle v,\infty\rangle}, y_{\langle v,\infty\rangle})$ to prune branches in the search tree computed at line 20 marked by (*). We emphasize that the values at $(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})$ 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 4 ([16]). *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.*

```
1  string FindBestEpisode(StringSet S, T, int ℓ)
2      string prefix, v;
3      episodePattern maxSeq; /* pair of string and int */
4      double upperBound = ∞, maxVal = −∞, val;
5      int k′;
6      CompactRepr x̄, ȳ; /* CRS */
7      PriorityQueue queue;    /* Best First Search*/
8      queue.push("", ∞);
9      while not queue.empty() do
10         (prefix, upperBound) = queue.pop();
11         if upperBound < maxVal then break;
12         foreach c ∈ Σ do
13             v = prefix+ c;    /* string concatenation */
14             x̄ = S.crs(v);
15             ȳ = T.crs(v);
16             k′ = argmax_k{f(x_⟨v,k⟩, y_⟨v,k⟩)} and val = f(x_⟨v,k′⟩, y_⟨v,k′⟩);
17             if val > maxVal then
18                 maxVal = val;
19                 maxEpisode = ⟨v, k′⟩;
20(*)          upperBound = max{f(x_⟨v,∞⟩, y_⟨v,∞⟩), f(x_⟨v,∞⟩, 0),
                                f(0, y_⟨v,∞⟩), f(0, 0)};
21             if upperBound > maxVal and |v| < ℓ then
22                 queue.push(v, upperBound);
23     return maxEpisode;
```

Fig. 3. Algorithm *FindBestEpisode*.

7 Efficient Data Structures to Count Matched Strings

In this section, we introduces some efficient data structures to speed up answering the queries.

First we pay our attention to the following problem.

Definition 6 (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 [17], we considered a subsequence automaton as a deterministic complete finite automaton that recognizes all possible subsequences of a set of strings,

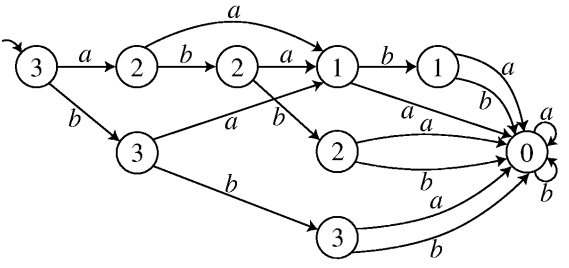

Fig. 4. Subsequence automaton for $S = \{abab, abb, bb\}$, where $\Sigma = \{a, b\}$. Each number on a state denotes the number of matched strings. For example, by traverse the states according to a string ab, we reach the state whose number is 2. It corresponds to the cardinality $|L^{\text{seq}}(ab)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

that is essentially the same as the directed acyclic subsequence graph (DASG) introduced by Baeza-Yates [5]. We showed an online construction of subsequence 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. In fact, we proved the lower bound $m = \Omega(n^k)$ for any $k > 0$ [34]. 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 similutaneously, 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 [15], we experimentally evaluated the optimal value of the parameter $mode$.

We now analyze the complexity of *episode pattern matching*. Given an episode pattern $\langle v, k\rangle$ and a string t, determine whether $t \in L^{\text{eps}}(\langle v, k\rangle)$ 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 introduced by Troíček in [33]. 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

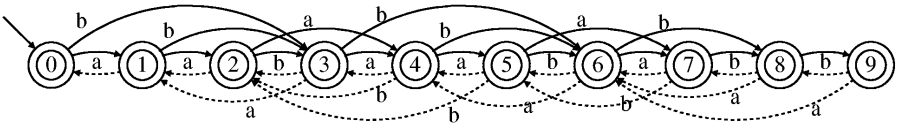

Fig. 5. *EDASG*(t) for $t = aabaababb$. Solid arrows denote the forward edges, and broken arrows denote the backward edges.

compute the threshold value θ of given v for t quickly in practice. As an example, *EDASG*($aabbab$) 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 $t_{i+1} \dots t_j$ 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.

8 Concluding Remarks

In this paper, we focused on the pattern discovery problem for substring, subsequence, and episode patterns to illustrate the basic ideas. We have already generalized it for various ways: considering variable-length-don't care patterns [19] and their variation [32], finding correlated patterns from given a set of strings with numeric attribute values [8], and finding a boolean combination of patterns instead of single pattern [6,18]. We also show another data structure to support fast counting of matched strings [21,20], and application to extend BONSAI system [7].

Acknowledgments. The work reported in this paper is an outcome of the joint research efforts with my colleagues: Setsuo Arikawa, Satoru Miyano, Takeshi

Shinohara, Satoru Kuhara, Masayuki Takeda, Hiroki Arimura, Shinichi Shimozono, Satoshi Matsumoto, Hiromasa Hoshino, Masahiro Hirao, Hideo Bannai, and Shunsuke Inenaga.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the 11th International Conference on Data Engineering*, Mar. 1995.

[2] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[3] H. Arimura, S. Arikawa, and S. Shimozono. Efficient discovery of optimal word-association patterns in large text databases. *New Generation Computing*, 18:49–60, 2000.

[4] H. Arimura, H. Asaka, H. Sakamoto, and S. Arikawa. Efficient discovery of proximity patterns with suffix arrays (extended abstract). In *Proc. the 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 152–156. Springer-Verlag, 2001.

[5] R. A. Baeza-Yates. Searching subsequences. *Theoretical Computer Science*, 78(2):363–376, Jan. 1991.

[6] H. Bannai, H. Hyyrö, A. Shinohara, M. Takeda, K. Nakai, and S. Miyano. Finding optimal pairs of patterns. In *Proc. 4th Workshop on Algorithms in Bioinformatics (WABI2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2004. (to appear).

[7] H. Bannai, S. Inenaga, A. Shinohara, M. Takeda, and S. Miyano. More speed and more pattern variations for knowledge discovery system BONSAI. *Genome Informatics*, 12:454–455, 2001.

[8] H. Bannai, S. Inenaga, A. Shinohara, M. Takeda, and S. Miyano. A string pattern regression algorithm and its application to pattern discovery in long introns. *Genome Informatics*, 13:3–11, 2002.

[9] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. Warmuth. Occam's razor. *Inf. Process. Lett.*, 24:377–380, 1987.

[10] A. Blumer, A. Ehrenheucht, D. Haussler, and M. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of ACM*, 36:929–965, 1989.

[11] R. Board and L. Pitt. On the necessity of Occam algorithms. *Theoretical Computer Science*, 100:157–184, 1992.

[12] A. Califano. SPLASH: Structural pattern localization analysis by sequential histograms. *Bioinformatics*, Feb. 1999.

[13] 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.

[14] E. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1967.

[15] M. Hirao, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best subsequence patterns. In *Proc. 3rd International Conference on Discovery Science (DS2000)*, volume 1967 of *Lecture Notes in Artificial Intelligence*, pages 141–154. Springer-Verlag, Dec. 2000. (Journal version is published in Theoretical Computer Science, Vol. 292, pp. 465–479, 2003).

[16] 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 (DS2001)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, Nov. 2001.

[17] 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 (SPIRE2000)*, pages 146–152. IEEE Computer Society, Sept. 2000.

[18] S. Inenaga, H. Bannai, H. Hyyrö, A. Shinohara, M. Takeda, K. Nakai, and S. Miyano. Finding optimal pairs of cooperative and competing patterns with bounded distance. In *Proc. 7th International Conference on Discovery Science (DS2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2004. (to appear).

[19] S. Inenaga, H. Bannai, A. Shinohara, M. Takeda, and S. Arikawa. Discovering best variable-length-don't-care patterns. In *Proc. 5th International Conference on Discovery Science (DS2002)*, volume 2534 of *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag, 2002.

[20] S. Inenaga, M. Takeda, A. Shinohara, H. Bannai, and S. Arikawa. Space-economical construction of index structures for all suffixes of a string. In *Proc. 27th Inter. Symp. on Mathematical Foundation of Computer Science (MFCS2002)*, volume 2420 of *Lecture Notes in Computer Science*, pages 341–352. Springer-Verlag, Aug. 2002.

[21] S. Inenaga, M. Takeda, A. Shinohara, H. Hoshino, and S. Arikawa. The minimum dawg for all suffixes of a string and its applications. In *Proc. 13th Ann. Symp. on Combinatorial Pattern Matching (CPM2002)*, volume 2373 of *Lecture Notes in Computer Science*, pages 153–167. Springer-Verlag, July 2002.

[22] T. Jiang and M. Li. On the complexity of learning strings and sequences. In *Proc. of 4th ACM Conf. Computational Learning Theory*, pages 367–371. ACM Press, 1991.

[23] M. Kearns and U. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

[24] L. Marsan and M.-F. Sagot. Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification. *J. Comput. Biol.*, 7:345–360, 2000.

[25] S. Matsumoto and A. Shinohara. Learning subsequence languages. In H. Kangassalo et al., editor, *Information Modeling and Knowledge Bases, VIII*, pages 335–344. IOS Press, 1997.

[26] S. Miyano, A. Shinohara, and T. Shinohara. Which classes of elementary formal systems are polynomial-time learnable? In *Proc. 2nd Workshop on Algorithmic Learning Theory (ALT91)*, pages 139–150, 1991.

[27] S. Miyano, A. Shinohara, and T. Shinohara. Polynomial-time learning of elementary formal systems. *New Generation Computing*, 18:217–242, 2000.

[28] 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.

[29] B. Natarajan. On learning sets and functions. *Machine Learning*, 4(1):67–97, 1989.

[30] L. Palopoli and G. Terracina. Discovering frequent structured patterns from string databases: an application to biological sequences. In *Proc. 5th International Conference on Discovery Science (DS2002)*, volume 2534 of *Lecture Notes in Computer Science*, pages 34–46. Springer-Verlag, 2002.

[31] 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.

[32] M. Takeda, S. Inenaga, H. Bannai, A. Shinohara, and S. Arikawa. Discovering most classificatory patterns for very expressive pattern classes. In *Proc. 6th International Conference on Discovery Science (DS2003)*, volume 2843 of *Lecture Notes in Computer Science*, pages 486–493. Springer-Verlag, 2003.

[33] Z. Troníček. Episode matching. In *Proc. of 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 143–146. Springer-Verlag, July 2001.

[34] Z. Troníček and A. Shinohara. The size of subsequence automaton. In *Proc. of 10th International Symposium on String Processing and Information Retrieval (SPIRE2003)*, volume 2857, pages 304–310. Springer-Verlag, 2003.

[35] L. G. Valiant. A theory of the learnable. *Communications of ACM*, 27:1134–1142, 1984.

[36] 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.