A Practical Algorithm to Find the Best Episode Patterns

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

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

Abstract. Episode pattern is a generalized concept of subsequence pattern where the length of substring containing the subsequence is bounded. Given two sets of strings, consider an optimization problem to find a best episode pattern that is common to one set but not common in the other set. The problem is known to be NP-hard. We give a practical algorithm to solve it exactly.

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.

In [4], Hirao et al. considered *subsequence patterns* as rules. A subsequence pattern s *matches* with a string t if s can be obtained by deleting zero or more characters from t. They 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 with almost all long strings.

In this paper, we consider *episode patterns*, which were originally introduced by Mannila et al. [5]. 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 give a practical solution to find a best episode pattern which separates a given set of strings from the other set of strings. We propose a practical implementation of exact search algorithm that practically avoids exhaustive search. The key idea is to introduce some heuristics to reduce the search space based on the combinatorial properties of episode patterns, and to utilize an efficient data structure that helps to determine whether an episode pattern matches with a fixed string, at the cost of preprocessing time and space requirement to construct it.

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

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

We formulate the problem by following our previous paper [4]. Readers should refer to [4] 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 is the optimization problem to be tackled.

Definition 1 (Finding the best episode pattern according to f).
Input *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output *An 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 $y_{\langle v,k\rangle} = |T \cap L^{eps}(\langle v, k\rangle)|$.*

We remark that the problem is NP-hard, since it is a generalization of *finding the best subsequence pattern [4].*

From the conicality of function f and the property of episode patterns, we can prove the following lemmas.

Lemma 1 ([4]). *For any $0 \leq x < x' \leq x_{\max}$ and $0 \leq y < y' \leq y_{\max}$, we have*

$$f(x, y) \leq \max\{f(x', y'), f(x', 0), f(0, y'), f(0, 0)\}.$$

Lemma 2. *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 1 and 2, we have the next lemma, that plays a key role in our algorithm which will be described in Section 4.

Lemma 3. *For any two episode patterns $\langle v, l\rangle$ and $\langle w, k\rangle$, if $v \preceq_{seq} w$ and $l \geq k$ then $f(x_{\langle w,k\rangle}, y_{\langle w,k\rangle}) \leq \max\{f(x_{\langle v,l\rangle}, y_{\langle v,l\rangle}), f(x_{\langle v,l\rangle}, 0), f(0, y_{\langle v,l\rangle}), f(0, 0)\}$.*

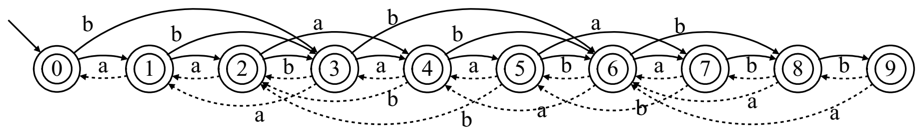

Fig. 1. *EDASG*(t), where $t = aabaababb$. Solid arrows denote the forward edges, and broken arrows denote the backward edges.

3 Episode Directed Acyclic Subsequence Graphs

We first 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 recently introduced by Troíček in [8]. A Directed Acyclic Subsequence Graph (DASG) [2] for a string t is a finite automaton that accepts all subsequences of t. An EDASG is a directed graph which combines two DASGs for t and the reversed string t^R. It contains two kinds of edges, *forward edges* corresponding to DASG(t), and *backward edges* corresponding to DASG(t^R). As an example, *EDASG*$(aabaababb)$ is shown in Fig. 1. 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.

With the use of *EDASG*(t), episode pattern matching can be answered quickly in practice, although the worst case behavior is still $O(mn)$. An on-line linear-time algorithm for constructing *EDASG*(t) for a string $t \in \Sigma^*$ was proposed in [8].

For strings $v, t \in \Sigma^*$, we define the *threshold value* θ of v for t by $\theta = min\{k \in \mathcal{N} \mid t \in L^{\text{eps}}(\langle v, k\rangle)\}$. If no such value, let $\theta = \infty$. Note that $t \notin L^{\text{eps}}(\langle v, k\rangle)$ for any $k < \theta$, and $t \in L^{\text{eps}}(\langle v, k\rangle)$ for any $\theta \leq k$. 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.

From now on, for a set S of strings and a string v, we consider the numerical sequence $\{x_k\}_{k=0}^{\infty}$, where $x_k = |S \cap L^{\text{eps}}(\langle v, k\rangle)|$. It clearly follows from Lemma 2 that the sequence is non-decreasing. Moreover, notice that $0 \leq x_k \leq |S|$ for any k, and $x_l = x_{l+1} = x_{l+2} = \cdots$, where l is the length of the longest string in S. It implies that $\{x_k\}_{k=0}^{\infty}$ consists of at most $\min\{|S|, l\}$ distinct values. Hence we can represent $\{x_k\}_{k=0}^{\infty}$ as a list of pairs (k, x_k) such that $x_{k-1} \neq x_k$. The length of the list is bounded by $\min\{|S|, l\}$. We call this list *a compact representation of the sequence* $\{x_k\}_{k=0}^{\infty}$ (*CRS*, for short).

We now show how to compute CRS for each v and a fixed S. Observe that x_k increases only at the threshold values of v for some $t \in S$. For each string $t_i \in S$, we compute the threshold value θ_i of v for t_i, and sort these threshold values in increasing order. From these sorted values, we can construct the CRS in linear time. To be summarized, if we use 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 $\{x_k\}_{k=0}^{\infty}$ is the same as that of computing x_k for a single k $(0 \leq k \leq \infty)$, by our method. In the next section, we use a data structure **StringSet** which supports the method to compute the CRS for any given string v.

4 Algorithm

The basic structure of the algorithm is similar to that in [4].

Fig. 2 shows our algorithm to find a best episode pattern from given two sets of strings, according to the function f. Optionally, we can specify the maximum length of episode patterns by the parameter ℓ. Here, we use a data structure **PriorityQueue** that supports the following methods.

– **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.

At line 16 marked by (*), we can simultaneously compute k' and *val* by using CRSs $\bar{x}$ and $\bar{y}$ in $O(|\bar{x}| + |\bar{y}|)$ time. By Lemma 3, we can use the value *upperBound* to prune branches in the search tree computed at line 20 marked by (**). Note 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 1. *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 maxi-*

```
1        string $FindBestEpisode$(StringSet $S$, $T$, int $\ell$)
2            string $prefix$, $v$;
3            episodePattern $maxSeq$; /* pair of string and int */
4            double $upperBound = \infty$, $maxVal = -\infty$, $val$;
5            int $k'$;
6            CompactRepr $\bar{x}$, $\bar{y}$; /* CRS */
7            PriorityQueue $queue$;    /* Best First Search*/
8            $queue.push$("", $\infty$);
9            while not $queue.empty()$ do
10               ($prefix$, $upperBound$) = $queue.pop()$;
11               if $upperBound < maxVal$ then break;
12               foreach $c \in \Sigma$ do
13                   $v = prefix +$ c;    /* string concatenation */
14                   $\bar{x} = S.crs(v)$;
15                   $\bar{y} = T.crs(v)$;
16 (*)               $k' = \mathrm{argmax}_k\{f(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})\}$ and $val = f(x_{\langle v,k'\rangle}, y_{\langle v,k'\rangle})$;
17                   if $val > maxVal$ then
18                       $maxVal = val$;
19                       $maxEpisode = \langle v, k'\rangle$;
20 (**)              $upperBound = \max\{f(x_{\langle v,\infty\rangle}, y_{\langle v,\infty\rangle}), f(x_{\langle v,\infty\rangle}, 0),$
                                     $f(0, y_{\langle v,\infty\rangle}), f(0,0)\}$;
21                   if $upperBound > maxVal$ and $|v| < \ell$ then
22                       $queue.push(v, upperBound)$;
23           return $maxEpisode$;
```

Fig. 2. Algorithm *FindBestEpisode*. In our pseudocode, the **break** statement is to jump out of the closest enclosing loop.

mizes $f(x_{\langle v,k\rangle}, y_{\langle v,k\rangle})$, 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.

5 Conclusion

We developed a practical algorithm to find the best episode pattern to separate given two sets of strings. Episode pattern is a generalization of subsequence pattern, and the search space of episode patterns is much larger than that of subsequence patterns. Nevertheless, our algorithm enabled to find the best episode pattern efficiently: the running time will *not* be much slower than that for finding subsequence patterns.

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 [6]. Fujino et al. showed an another approach to find the best *proximity pattern* [3]. It may be interesting to combine these approaches into one. We are now in the process of installing our algorithm into the core of the decision tree generator in the BONSAI system [7].

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. R. Fujino, H. Arimura, and S. Arikawa. Discovering unordered and ordered phrase association patterns for text mining. In *Proc. of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, volume 1805 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Apr. 2000.
4. 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.
5. 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.
6. S. Miyano, A. Shinohara, and T. Shinohara. Polynomial-time learning of elementary formal systems. *New Generation Computing*, 18:217–242, 2000.
7. 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.
8. Z. Troníček. Episode matching. In *Proc. of 12th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, pages 143–146. Springer-Verlag, July 2001.