Online Construction of Subsequence Automata for Multiple Texts

Hiromasa Hoshino Ayumi Shinohara Masayuki Takeda Setsuo Arikawa
Department of Informatics, Kyushu University 33,
Fukuoka 812-8581, Japan
{ hoshino, ayumi, takeda, arikawa } @i.kyushu-u.ac.jp

Abstract

We consider a deterministic finite automaton which accepts all subsequences of a set of texts, called subsequence automaton. *We show an online algorithm for constructing subsequence automaton for a set of texts. It runs in $O(|\Sigma|(m+k)+N)$ time using $O(|\Sigma|m)$ space, where $|\Sigma|$ is the size of alphabet, m is the size of the resulting subsequence automaton, k is the number of texts, and N is the total length of texts. It can be used to preprocess a given set $\mathcal{S}$ of texts in such a way that for any query $w \in \Sigma^*$, returns in $O(|w|)$ time the number of texts in $\mathcal{S}$ which contain w as a subsequence. We also show an upper bound of the size of automaton compared to the minimum automaton.*

1 Introduction

A string s is called a *subsequence* of a string t if s can be obtained from t by deleting zero or more symbols. The basic problem is to determine whether a string s is a subsequence of a string t or not. It is almost trivial to show that the problem can be solved in $O(|s|+|t|)$. When s is fixed, we can solve it in $O(|t|)$ time by constructing a finite automaton which accepts all strings of which s is a subsequence [14].

For the case that t is fixed, Baeza-Yates [1] introduced the directed acyclic subsequence graph (DASG) of a single text t as the smallest deterministic partial finite automaton that recognizes all possible subsequences of t. By using DASG of t, we can determine whether a string s is a subsequence of a string t in $O(|s|)$ time. He showed a right-to-left algorithm for building the DASG for a single text. On the other hand, Troníček and Melichar [15] showed a left-to-right algorithm building the DASG for a single text.

We now turn our attention to the case of multiple texts. A string s is called a *common subsequence* of a set $\mathcal{S}$ of texts if s is a subsequence of t for any $t \in \mathcal{S}$. Finding a common subsequence of a set of texts is one of the most basic task to characterize the set. The longest common subsequence problem is NP-complete if the number k of strings is not fixed in advance [7].

In machine learning, given positive and negative examples, a learning algorithm tries to find a general rule which explains the examples correctly. When we use a subsequence as a rule to distinguish positive strings from negative strings, the main task is to find a *consistent* subsequence with given examples. Here, we call a subsequence s is consistent with positive strings *Pos* and negative strings *Neg* if s is a subsequence for any $w \in$ *Pos* and s is not a subsequence for any $w \in$ *Neg*. The computational complexity related to this problem was studied in [6, 9, 10]. It is shown that finding a consistent subsequence with given positive and negative examples is NP-complete.

In some application area, finding a subsequence that is *maximally* consistent with given examples is more important, because there might be no consistent subsequence in real data due to noises or inconsistency of the data itself. The measure to be maximized depends on application domains, including the information gain due to Quinlan [11], χ^2-value, and so on [4, 13].

Among these various situations, a common basic operation is to count the number of texts which contain a given string as a subsequence. When a set $\mathcal{S}$ of texts is fixed, it is worth considering to construct a data structure that returns the number for a query string w in linear time with respect to $|w|$, not to the total length of the strings in $\mathcal{S}$. A straightforward approach is to build DASGs for each text in $\mathcal{S}$. Given a query string w, we have only to traverse all DASGs simultaneously, and return the total number of DASGs that accept w. It clearly runs in $O(k|w|)$ time, where k is the number of texts in $\mathcal{S}$. When the running time is more critical, we can build a product of k DASGs so that the running time becomes $O(|w|)$ time, at the cost of preprocessing time and space requirement. This is the DASG for multiple texts. We remark that the preprocessing time is also critical in such situations that the set $\mathcal{S}$ of texts are dynamically changed, typically in machine learning applications [4, 13]. Moreover, an online algorithm is preferred in some applications.

Baeza-Yates presented a right-to-left algorithm for building the DASG for multiple texts also [1]. Moreover, Troníček and Melichar [15], and Crochemore and Troníček [2] showed left-to-right algorithms for building the DASG for a set of texts. Unfortunately, none of these algorithms is online.

In this paper, we consider a subsequence automaton as a deterministic complete finite automaton that recognizes all possible subsequences of multiple texts, that is essentially the same as DASG. We note that our automaton is also acyclic except the single 'error' state. We show an online construction of subsequence automaton for multiple texts. Our algorithm runs in $O(|\Sigma|(m+k)+N)$ time using $O(|\Sigma|m)$ space, where $|\Sigma|$ is the size of alphabet, m is the size of the resulting subsequence automaton, and N is the total length of texts.

For k texts of length n, since the number m of states can be bounded by $O(n^k)$, our algorithm runs in $O(|\Sigma|(n^k+k)+kn)$ time using $O(|\Sigma|n^k)$ space. The running time is superior to the offline algorithm proposed by Crochemore and Troníček [2], that requires $O(k|\Sigma|n^k)$ time.

Moreover, we prove that the size of the subsequence automaton M constructed by our algorithm is at most $k! \cdot m_{\min}$, where $m_{\min}$ is the size of minimum automaton equivalent to M. We also show our experimental results on the expected size of a subsequence automaton M over randomly generated texts, and the expected ratio of the size of M to that of minimum automaton equivalent to M. Finally, we consider the trade off between construction time and the response time.

2 Preliminaries

Let Σ be a finite alphabet, and let ε be the empty string. For a string $w \in \Sigma^*$, we denote by $|w|$ the length of w, We denote by $w[i]$ the i-th character of w, and by $w[i:j]$ the substring of w starting at i and ending at j, that is, $w[i:j] = w[i]\cdots w[j]$. If $i > j$, we define $w[i:j] = \varepsilon$. We abbreviate $w[i:|w|]$ to $w[i:]$. For a set $\mathcal{S}$, we denote by $|\mathcal{S}|$ the cardinality of $\mathcal{S}$.

A *subsequence* of a string w is any string obtained by deleting zero or more symbols from w. For a tuple $\boldsymbol{t} = [p_1, p_2, \ldots, p_k]$, we denote the i-th element by $\boldsymbol{t}[i] = p_i$. Let $\mathcal{N}$ be the set of all natural numbers, and let $\mathcal{N}(m) = \{i \in \mathcal{N} \mid 0 \le i \le m\} \cup \{\infty\}$ for $m \in \mathcal{N}$.

3 Subsequence Automata

In this section, we define a subsequence automaton (SA) for a single text, and for multiple texts. We follow the standard notation of finite automata [5]. A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is an input alphabet, $\delta : Q \times \Sigma \to Q$ is a transition

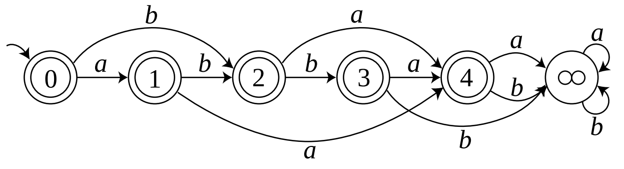

Figure 1. Subsequence automaton for string $abba$, where $\Sigma = \{a, b\}$.

function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. We denote by q_∞ the 'error' state such that $q_\infty \notin F$ and $\delta(q_\infty, c) = q_\infty$ for all $c \in \Sigma$. The size of a finite automaton refers to the number of states.

Definition 1 (Subsequence automaton for a single text) *Let T be a string of length n in Σ^*. A* subsequence automaton *for T is a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where*

- $Q = \mathcal{N}(n)$,
- $q_0 = 0$,
- $\delta(q, c) = \min(\{j \mid q < j \le n$ *and* $T[j] = c\} \cup \{\infty\})$,
- $F = Q - \{q_\infty\}$, *where* $q_\infty = \infty$.

Example 1 *The subsequence automaton for string $abba$ is shown in Fig. 1, where $\Sigma = \{a, b\}$.*

We can easily verify that the following theorem holds.

Theorem 1 *Let T be a string and M be the subsequence automaton for T. Then M accepts all subsequences of T.*

Definition 2 (Subsequence automaton for multiple texts) *Let $\mathcal{S} = \{T_1, T_2, \ldots, T_k\} \subseteq \Sigma^*$ be a finite set of strings, and let $n_i = |T_i|$ for each $i = 1, 2, \ldots, k$. A* subsequence automaton *for $\mathcal{S}$ is a tuple $M = (Q, \Sigma, \delta, q_0, F)$, where*

- $Q = \mathcal{N}(n_1) \times \mathcal{N}(n_2) \times \cdots \times \mathcal{N}(n_k)$,
- $q_0 = [0, 0, \ldots, 0]$,
- $\delta([q_1, q_2, \ldots, q_k], c) = [\delta_{T_1}(q_1, c), \delta_{T_2}(q_2, c), \ldots, \delta_{T_k}(q_k, c)]$, *where each δ_{T_i} is the transition function of the subsequence automaton for T_i, that is, $\delta_{T_i}(q_i, c) = \min(\{j \mid q_i < j \le n_i$ and $T_i[j] = c\} \cup \{\infty\})$,*
- $F = Q - \{q_\infty\}$, *where* $q_\infty = [\infty, \infty, \ldots, \infty]$.

Example 2 *The subsequence automaton for strings aa and abb is shown in Fig. 2, where $\Sigma = \{a, b\}$. For clarity, we draw reachable states only.*

The next theorem is immediate from the definition.

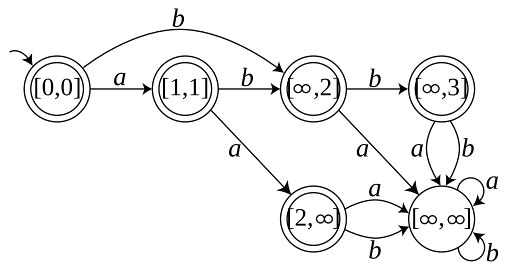

Figure 2. Subsequence automaton for strings aa and abb, where $\Sigma = \{a, b\}$.

```
SA $sa$
while ($c$ = getchar()) != EOF
    if $c$ == LF
        $sa$.extendDimension()
    else
        $sa$.appendChar($c$)
```

Figure 3. Online construction of subsequence automaton.

Theorem 2 *Let M be the subsequence automaton for $\mathcal{S} \subseteq \Sigma^*$. Then M accepts all subsequences of any text $T \in \mathcal{S}$.*

Remark that for each state $q = [q_1, q_2, \ldots, q_k]$, the cardinality of the set $\{q_i \mid q_i \neq \infty\}$ corresponds to the number of texts in $\mathcal{S}$ that contains a string leading to this state as a subsequence. Therefore we associate the number with each state, and we refer to it as a function *Match* : $Q \to \mathcal{N}$.

In the sequel, we extend the domain of the transition function δ to $Q \times \Sigma^*$ in the standard way; $\delta(q, \varepsilon) = q$, and $\delta(q, cw) = \delta(\delta(q, c), w)$ for $q \in Q$, $c \in \Sigma$, and $w \in \Sigma^*$.

4 Construction of Subsequence Automata

This section shows an online algorithm which constructs a subsequence automaton for a set of texts. We introduce two procedures *extendDimension* and *appendChar* as member functions of the class *SA*. By these two procedures, the whole construction algorithm will be described in Fig. 3. Here, we assume that each text is separated by the character "LF" , and the input stream ends by the character "EOF".

We first explain the basic idea of the algorithm. Let us remind the subsequence automaton for strings aa and abb in Fig. 2. Suppose that we append a character a to the second text abb. The resulting subsequence automaton should be changed as in Fig. 5. The difference between these

```
class SA

SA::SA(void)    /* constructor of SA */
    create the initial state $q_0$ and the error state $q_\infty$
    for each $a \in \Sigma$
        D[$q_0$][$a$] = D[$q_\infty$][$a$] = $q_\infty$
        NonSolidArcList[$q_\infty$][$a$] = $\{q_0\}$
        targetStates[$a$] = $\{q_\infty\}$
    Match[$q_0$] = 1 ; Match[$q_\infty$] = 0

void SA::extendDimension(void)
    for each $a \in \Sigma$
        /* delete NonSolidArcList */
        for each $i \in$ targetStates[$a$]
            NonSolidArcList[$i$][$a$] = $\phi$
        $r$ = D[$q_0$][$a$]
        NonSolidArcList[$r$][$a$] = $\{q_0\}$
        targetStates[$a$] = $\{r\}$
    Match[$q_0$] ++

void SA::appendChar(char $c$)
    lastTargetStates = targetStates[$c$]
    targetStates[$c$] = $\phi$
    for each $i \in$ lastTargetStates    /* copy NonSolidArcList */
        lastNonSolidArcs[$i$] = NonSolidArcList[$i$][$c$]
        NonSolidArcList[$i$][$c$] = $\phi$
    for each $i \in$ lastTargetStates
        create a new state $p$    /* copy the state $i$ */
        for each $a \in \Sigma$
            $r$ = D[$i$][$a$] ; D[$p$][$a$] = $r$
            NonSolidArcList[$r$][$a$] = NonSolidArcList[$r$][$a$]$\cup\{p\}$
            targetStates[$a$] = targetStates[$a$]$\cup\{r\}$
        Match[$p$] = Match[$i$] + 1
        for each $j \in$ lastNonSolidArcs[$i$]
            D[$j$][$c$] = $p$
        lastNonSolidArcs[$i$] = $\phi$

int SA::query(string $s$)
    $q$ = $q_0$
    for each character $c$ in $s$
        $q$ = D[$q$][$c$]
    return Match[$q$]
```

Figure 4. Member functions in the class SA to construct and reply.

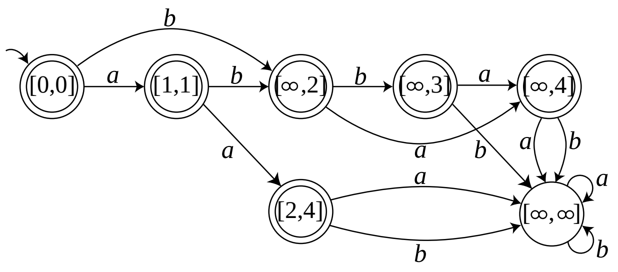

Figure 5. Subsequence automaton for strings aa and $abba$, where $\Sigma = \{a, b\}$.

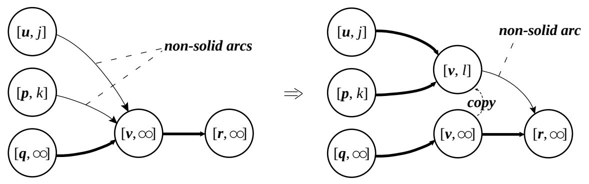

Figure 6. Main operation. A character is appended to T_i at position ℓ.

two automata is caused by the following three transitions in Fig. 2: $\delta([1,1], a) = [2, \infty]$, $\delta([\infty, 2], a) = [\infty, \infty]$, and $\delta([\infty, 3], a) = [\infty, \infty]$. Remark that the second component of the state will be changed from a natural number into ∞ by any of these transitions. We formalize this concept as follows.

Definition 3 *For a transition $\delta(q, c) = r$, we call a triple $\langle q, c, r\rangle$ an* arc*, and r is the* target state *of the arc. We define that an arc $\langle \boldsymbol{u}, c, \boldsymbol{v}\rangle$ is* non-solid with respect to c *iff $\boldsymbol{u}[i] \neq \infty$ and $\boldsymbol{v}[i] = \infty$, where i is the index of the text to which a new character is to be appended. We say that an arc is* solid *iff the arc is not* non-solid*.*

Let $\langle \boldsymbol{u}, c, \boldsymbol{v}\rangle$ be a non-solid arc with respect to c, and let $\boldsymbol{u}[i] = j$. The fact $\boldsymbol{v}[i] = \infty$ implies that the character c does not appear after the j-th position of the text T_i at this moment. Therefore, when appending the character c to the text T_i, we have to copy the state $\boldsymbol{v}$ and change that $\boldsymbol{v}[i] = \ell$, where ℓ is the position of the character appended in T_i (see Fig. 6). Here, for a tuple $\boldsymbol{t} = [p_1, p_2, \ldots, p_k]$, we define $[\boldsymbol{t}, p_{k+1}] = [p_1, p_2, \ldots, p_k, p_{k+1}]$.

In the algorithm shown in Fig. 4, the array "Match" corresponds to the function *Match* and the array "D" does to the transition function δ.

Extend dimension We show the procedure *extendDimension* in Fig. 4, which is called when the input character is "LF". By the definition of a non-solid arc, all arcs from the initial state are non-solid in this case. Thus we add them to the array "NonSolidArcList" of non-solid arcs, and do target states of them to the array "targetStates" of target states. Since the empty string ε is also a subsequence of any string, we add one to Match[q_0].

Append a character Fig. 4 shows the procedure *appendChar* that appends a character. The most important point is that all the non-solid arcs which share the same target state should keep sharing the copy of it, after modifying the arcs (see Fig. 6 again). The array "targetStates" does the job. Since all arcs from a new state are non-solid, we add them to "NonSolidArcList", and do target states to "targetStates".

Reply to the query In Fig. 4, we show the procedure *query*, which returns the number of strings that contain the given string s as a subsequence. It is obvious since the number equals to *Match*$(\delta(q_0, s))$.

We remark that our algorithm often creates unreachable states. A state will be unreachable if all arcs which point them are non-solid with respect to a single character c, and c is appended. By introducing the *reference number* of a state, that is the number of arcs which point the state, we can maintain that all states are always reachable: when appending a character, if the reference number of a state is equal to the number of non-solid arcs which point the state, we treat the state as a new one instead of creating another one. We verified that this modification is very effective to reduce the size of subsequence automaton in practice.

4.1 Complexity

We show that the algorithm runs in $O(|\Sigma|(m + k) + N)$ time using $O(|\Sigma|m)$ space, where k is the number of texts, N is the total length of texts, and m is the size of the subsequence automaton constructed by it.

First, we analyze the total amount of time consuming by the main routine. We estimate the time complexity of *extendDimension* and *appendChar* separately. It takes $O(N)$ time to read texts.

Next, we consider the time consuming by *extendDimension*. The cost of deleting the NonSolidArcList is the same that of creating it. So, we charge the cost of deleting to appending. The procedure requires $O(|\Sigma|)$ time. Since the procedure operates k times, the total amount of time is $O(|\Sigma|k)$ time.

Finally, we estimate the time of *appendChar* by charging each cost of operations either to a state or to an arc. We charge the costs of copying to states. When a state is copied, it takes $O(|\Sigma|)$ time to set transition functions, to register a copied state in the NonSolidArcList, and to register a new target state in the targetStates for each character in Σ. Since there are m states, the total amount of time to copy states is $O(|\Sigma|m)$.

On the other hand, we charge the cost of modifying non-solid arcs to themselves. It is noticed that the total number of arcs is $|\Sigma|m$, and each arc which becomes solid once never changes to be non-solid, except for arcs from the initial state. Since the modification can be done in $O(1)$ time for each non-solid arc, the total amount of time to modify is $O(|\Sigma|m)$.

Thus, the total amount of the time of the whole construction is $O(|\Sigma|(m+k)+N)$. The space complexity is $O(|\Sigma|m)$, since the total number of arcs is $O(|\Sigma|m)$.

For the sake of comparison with the algorithm proposed by Crochemore and Troníček [2], assume that we have k texts of length n. Since the number m of states can be bounded by $O(n^k)$, our algorithm runs in $O(|\Sigma|(n^k+k)+kn)$ time using $O(|\Sigma|n^k)$ space. The running time is superior to the offline algorithm proposed by Crochemore and Troníček, that requires $O(k|\Sigma|n^k)$ time.

4.2 Upper bound of the size of SA

The subsequence automaton built by our algorithm is not always minimum since it may contain equivalent states. We can minimize it by standard method. In fact, we can do it in linear time with respect to the size of automaton by applying the algorithm in [12], since our automaton is acyclic except the single error state.

In this section, we show that the size of automaton M constructed by our algorithm is at most $k! \cdot m_{\min}$, where $m_{\min}$ is the size of minimum automaton equivalent to M.

Definition 4 *We say that a state $q \in Q$ is* equivalent *to $q' \in Q$ (with respect to the number of matched texts) iff Match$(\delta(q,w))$ = Match$(\delta(q',w))$ for all $w \in \Sigma^*$. We denote by $q \equiv q'$ that q is equivalent to q'.*

In the sequel, we often denote a state $[q_1, q_2, \ldots, q_k] \in Q$ by $[p_1, p_2, \ldots, p_k]^r$ such that $p_i = n_i - q_i$, where n_i is the length of T_i. Note that $p_i = -\infty$ if $q_i = \infty$. Let δ_{T_i} be the transition function of the subsequence automaton for each text T_i. The next lemma can be verified easily.

Lemma 1 *For any two components p_i and p_j $(i \neq j)$ of a state $[p_1, p_2, \ldots, p_k]^r \in Q$, the following statement holds.*

If either $p_i > p_j$, or $p_i = p_j$ and $T_i[n_i - p_i + 1 :] \neq T_j[n_j - p_j + 1 :]$ then

$$\begin{aligned} \delta_{T_i}(n_i - p_i, T_i[n_i - p_i + 1 :]) &= n_i, \quad \textit{and} \\ \delta_{T_j}(n_j - p_j, T_i[n_i - p_i + 1 :]) &= \infty. \end{aligned}$$

Lemma 2 *If $[p_1, p_2, \ldots, p_k]^r \equiv [p'_1, p'_2, \ldots, p'_k]^r$, then a sequence $(p'_1, p'_2, \ldots, p'_k)$ is the permutation of a sequence $(p_1, p_2, \ldots, p_k)$.*

Proof. We show that if $(p'_1, p'_2, \ldots, p'_k)$ is not a permutation of $(p_1, p_2, \ldots, p_k)$, then $[p_1, p_2, \ldots, p_k]^r \not\equiv [p'_1, p'_2, \ldots, p'_k]^r$. We consider an one to one function f over $\{1, 2, \ldots. k\}$ such that f maximizes the cardinality of the set $\{i \mid p_i = p'_{f(i)}$ and $T_i[n_i - p_i + 1 :] = T_{f(i)}[n_{f(i)} - p'_{f(i)} + 1 :]\}$. Let $D_f = \{i \mid T_i[n_i - p_i + 1 :] \neq T_{f(i)}[n_{f(i)} - p'_{f(i)} + 1 :]\}$, $D'_f = \{f(i) \mid i \in D_f\}$, $E_f = \{i \mid T_i[n_i - p_i + 1 :] = T_{f(i)}[n_{f(i)} - p'_{f(i)} + 1 :]\}$, and $E'_f = \{f(i) \mid i \in E_f\}$. Since $(p'_1, p'_2, \ldots, p'_k)$ is not a permutation of $(p_1, p_2, \ldots, p_k)$, there is an index i such that $p_i \neq p'_{f(i)}$, $D_f \neq \phi$, and $D'_f \neq \phi$. For a string $w \in \Sigma^*$, let $d = |\{i \in D_f \mid \delta_{T_i}(n_i - p_i, w) = \infty\}|$, $d' = |\{i \in D'_f \mid \delta_{T_i}(n_i - p'_i, w) = \infty\}|$, $e = |\{i \in E_f \mid \delta_{T_i}(n_i - p_i, w) = \infty\}|$, and $e' = |\{i \in E'_f \mid \delta_{T_i}(n_i - p'_i, w) = \infty\}|$. Clearly, $e = e'$ for any string w. *Match*$(\delta([p_1, p_2, \ldots, p_k]^r, w)) = k - (d + e)$, and *Match*$(\delta([p'_1, p'_2, \ldots, p'_k]^r, w)) = k - (d' + e')$. Let $\hat{p}$ be the largest element in $\{p_i, p'_{f(i)} \mid i \in D_f\}$ and let i_{max} be its index. Note that i_{max} is not necessarily unique. Let $w = T_{i_{max}}[n_{i_{max}} - \hat{p} + 1 :]$.

Case 1: $\hat{p} = p_{i_{max}}$. Clearly, $\delta_{T_{i_{max}}}(n_{i_{max}} - \hat{p}, w) = n_{i_{max}} \neq \infty$. There is no $i \in D'_f$ such that $w = T_i[n_i - p'_i + 1 :]$ by the definition of f. Since $\hat{p}$ is the largest element in $\{p_i, p'_{f(i)} \mid i \in D_f\}$, we have $d > d'$ by Lemma 1. Thus, *Match*$(\delta([p_1, p_2, \ldots, p_k]^r, w)) \neq$ *Match*$(\delta([p'_1, p'_2, \ldots, p'_k]^r, w))$.

Case 2: $\hat{p} = p'_{i_{max}}$. In the same way as Case 1, we have $d < d'$, that implies *Match*$(\delta([p_1, p_2, \ldots, p_k]^r, w)) \neq$ *Match*$(\delta([p'_1, p'_2, \ldots, p'_k]^r, w))$.

In both cases, we have $[p_1, p_2, \ldots, p_k]^r \not\equiv [p'_1, p'_2, \ldots, p'_k]^r$. ∎

The following theorem is immediate from Lemma 2.

Theorem 3 *Let M be a subsequence automaton constructed by our algorithm for a set of k texts. The number of equivalent states in M with respect to the number of matched text is at most $k!$.*

Therefore, the size of the automaton M is at most $k! \cdot m_{\min}$.

4.3 Experimental results on the size of SA

We now consider the size of subsequence automata for k texts of length n. The trivial upper bound is $O(n^k)$. The lower bound for $k > 2$ texts is not known, while Crochemore and Troníček [2] showed that $\Omega(n^2)$ states are required for $k = 2$ at the worst case.

In this section, we show our experimental results on the size of subsequence automata when the texts are randomly generated for $|\Sigma| = 2$. Fig. 7 shows the minimum, maximum and average size during ten trials. Remark that the

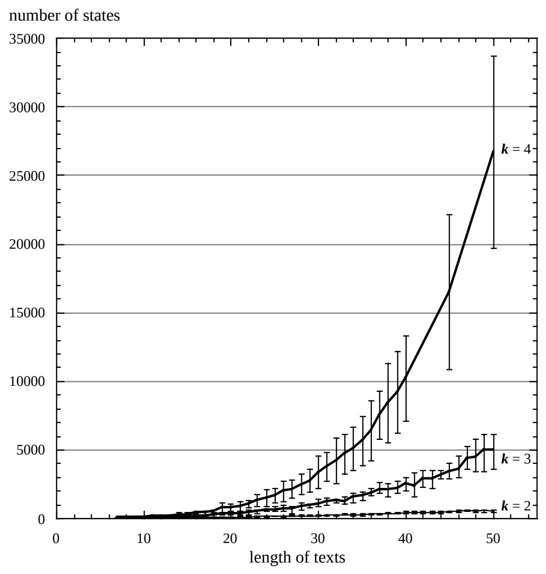

Figure 7. The size of subsequence automata.

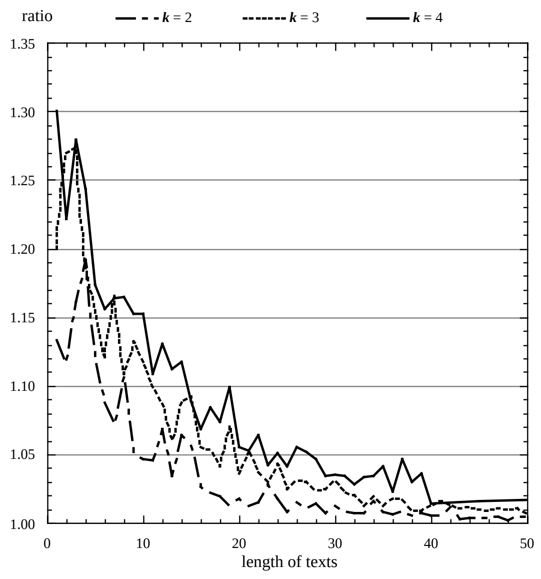

Figure 8. The ratios of the size of subsequence automata compared to that of minimum automata.

size is exactly equal to the length of the text plus two when $k = 1$. Fig. 8 shows the ratios of the size of subsequence automata to that of minimized them. We can see that the ratios are much less than the theoretical upper bound $k!$ in practice, and they become close to one as the length of texts increases. The result shows that there is not large difference in the size between our automaton and minimum automaton equivalent to it.

5 Trade off between constructing time and response time

For a set $\mathcal{S}$ of strings, once the subsequence automaton for $\mathcal{S}$ is constructed, the response for queries is very fast. However, in some applications, the construction time is also critical. For example, in our recent paper [4], we are developing a system to find a subsequence that maximally separates given positive set $\mathcal{S}_{pos}$ of strings from negative set $\mathcal{S}_{neg}$ of strings. For each $\mathcal{S}_{pos}$ and $\mathcal{S}_{neg}$, we use subsequence automata to count the number of matched strings for each candidate subsequence. It would be useless if the construction of subsequence automata M requires more time than the sum of reduced time to answer queries by using M. In this sense, there is a trade off between the constructing time and the sum of response time. Naturally, it depends on the number of queries. In order to strike a balance, we take the following approach. For a specified parameter $\ell > 0$, we partition the set $\mathcal{S}$ into $d = k/\ell$ subsets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_d$ of at most ℓ strings, and construct d subsequence automata for each $\mathcal{S}_i$. When asking a query, we have only to traverse all automata similutaneously, and return the sum of the answers. If $\ell = 0$, we do not use subsequence automata.

We empirically estimated the performance as follows. The inputs are two sets $\mathcal{S}_{pos}$ and $\mathcal{S}_{neg}$ of amino acid sequences taken from the PIR database, that are converted into strings over binary alphabet $\Sigma = \{0, 1\}$, according to the alphabet indexing discovered by BONSAI [13]. The average length of the strings is approximately 30, and $|\mathcal{S}_{pos}| = 70$ and $|\mathcal{S}_{neg}| = 100$. The query set consists of all strings of length at most 13 over Σ. That is, we asked 16383 queries. The experiment was carried out on an AlphaServer DS20 with an Alpha 21264 processor at 500MHz.

Fig. 9 shows the results. We can see that the construction time increases with ℓ, as we expected, since the total size of the automata increases. On the other hand, the sum of response time decreases with ℓ. In this case, $\ell = 3$ yields the minimum running time.

6 Concluding Remarks

We gave an online algorithm to construct a deterministic finite automaton, called *subsequence automaton*, which accepts all subsequence of a set of texts. It can be used to preprocess a given set $\mathcal{S}$ of texts in such a way that for any query w, returns the number of texts in $\mathcal{S}$ which contains w as a subsequence in $O(|w|)$ time. We also show an upper bound of the size of automaton compared to the minimum automaton.

It will be a challenging problem to extend our algorithm

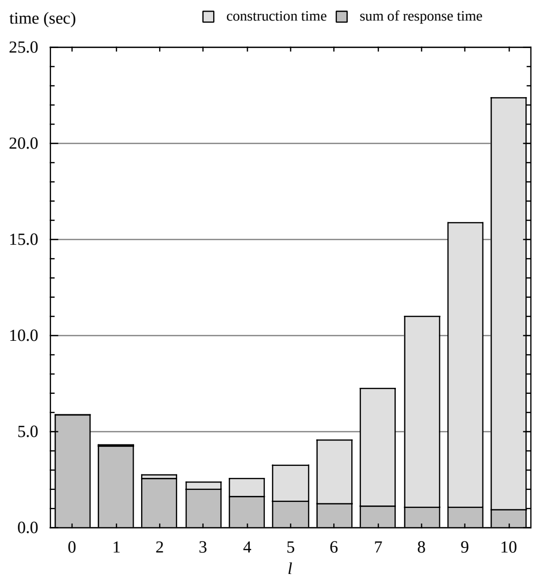

Figure 9. Trade off between construction time and response time.

so that we can treat an *episode matching* [3, 8], where the total length of the matched subsequence is bounded.

Acknowledgements

We gratefully acknowledge Prof. Alberto Apostolico for his helpful suggestions.

References

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

[2] M. Crochemore and Z. Troníček. Directed acyclic subsequence graph for multiple texts. Technical Report IGM-99-13, Institut Gaspard-Monge, June 1999.

[3] G. Das, R. Fleischer, L. Gasieniek, D. Gunopulos, and J. Kärkkäinen. Episode matching. In A. Apostolico and J. Hein, editors, *Proc. of the 8th Annual Symposium on Combinatorial Pattern Matching*, volume 1264 of *Lecture Notes in Computer Science*, pages 12–27. Springer-Verlag, 1997.

[4] M. Hirao, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find best subsequence patterns. Technical Report DOI-TR-CS-175, Department of Informatics, Kyushu University, June 2000.

[5] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[6] 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, 1991.

[7] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25:322–336, Apr. 1978.

[8] H. Mannila, H. Toivonen, and A. I. Vercamo. Discovering frequent episode in sequences. In *Proc. of the 1st International Conference on Knowledge Discovery and Data Mining*, pages 210–215. AAAI Press, Aug. 1995.

[9] 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.

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

[11] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[12] D. Revuz. Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, Jan. 1992.

[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. Problems related to subsequences and supersequences. In *Proc. of 6th International Symposium on String Processing and Information Retrieval*, pages 199–205. IEEE Computer Society, 1999.

[15] Z. Troníček and B. Melichar. Directed acyclic subsequence graph. In *Proc. of the Prague Stringology Club Workshop '98*, pages 107–118, Sept. 1998.