A Practical Algorithm to Find the Best Subsequence Patterns

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

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

Abstract. 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. We generalize the problem to an optimization problem, and give a practical algorithm to solve it exactly. Our algorithm uses pruning heuristic and subsequence automata, and can find the best subsequence. We show some experiments, that convinced us the approach is quite promising.

1 Introduction

String is one of the most fundamental structure to express and reserve information. In these days, a lot of string data are available. String processing has vast application area, such as Genome Informatics and Internet related works. It is quite important to discover useful rules from large text data or sequential data [1, 6, 9, 22]. 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.

Shimozono et al. [20] 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 is 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 this paper, we give a practical solution to find the best subsequence 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. Since the problem is 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) asking queries as few as possible, or (2) speeding 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 [18], combined with some properties on the subsequence languages.

Next, we accelerate answering for subsequence queries. Since the sets of strings are fixed in finding the best subsequence pattern, it is reasonable to preprocess the sets so that answering subsequence 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 [11]. Subsequence automaton can answer quickly for subsequence query, at the cost of preprocessing time and space requirement to construct it.

Since these two approaches are different in their aims, we expect that a balanced integration of these two would result in the most efficient way to find the best subsequence patterns. In order to verify the performance of our algorithm, we are performing some experiments on these two approaches. We report some results of the experiments, that convinced us it is quite promising.

2 Preliminaries

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, and for a set S, we denote by $|S|$ the cardinality of 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 $v \preceq_{\mathrm{str}} w$ that v is a substring of w, and by $v \preceq_{\mathrm{seq}} w$ that v is a subsequence of w. For a string v, we define the *substring language* $L^{\mathrm{str}}(v)$ and *subsequence language* $L^{\mathrm{seq}}(v)$ as follows:

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

The following lemma is obvious from the definitions.

Lemma 1. *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 $L^{seq}(v) \supseteq L^{seq}(w)$.*

3 Formulation of the Problem

Let *good* be a function from $\Sigma^* \times 2^{\Sigma^*} \times 2^{\Sigma^*}$ to the set of real numbers. We formulate the problem to be solved as follows.

Definition 1 (Finding the best pattern according to *good*).
Input *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output *A string $w \in \Sigma^*$ that maximizes the value $good(w, S, T)$.*

Intuitively, the value $good(w, S, T)$ expresses the goodness to distinguish S from T using the rule specified by a string w. The definition of *good* varies for each application. For examples, the χ^2 values, entropy information gain, and gini index are frequently used (See [18]). Essentially, these statistical measures are defined by the numbers of strings that satisfy the rule specified by w. In this paper, we only consider the rules defined as substring languages and subsequence languages. We call these problems as *finding the best substring pattern*, and *finding the best subsequence pattern*, respectively. Let L be either L^{str} or L^{seq}. Then any of the above examples of the measures can be described in the following form.

$$\begin{aligned} good(w, S, T) &= f(x_w, y_w, |S|, |T|), \text{ where} \\ x_w &= |S \cap L(w)|, \\ y_w &= |T \cap L(w)|. \end{aligned}$$

For example, the entropy information gain, which is introduced by Quinlan [19] and also used in BONSAI system [20], can be defined in terms of the function f as follows:

$$\begin{aligned} f(x, y, x_{\max}, y_{\max}) = &-\frac{x+y}{x_{\max}+y_{\max}} I(x, y) \\ &-\frac{x_{\max}-x+y_{\max}-y}{x_{\max}+y_{\max}} I(x_{\max}-x, y_{\max}-y), \end{aligned}$$

$$\text{where } I(s,t) = \begin{cases} 0 & (\text{if } s = 0 \text{ or } t = 0), \\ -\frac{s}{s+t}\log\frac{s}{s+t} - \frac{t}{s+t}\log\frac{t}{s+t} & (\text{otherwise}). \end{cases}$$

When the sets S and T are fixed, the values $x_{\max} = |S|$ and $y_{\max} = |T|$ become constants. Thus, we abbreviate the function $f(x, y, x_{\max}, y_{\max})$ to $f(x, y)$ in the sequel.

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

Definition 2. *We say that a function $f(x,y)$ 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.

Lemma 2. *Let $f(x,y)$ be a conic function defined over $[0,x_{\max}] \times [0,y_{\max}]$. For any $0 \leq x < x' \leq x_{\max}$ and $0 \leq y < y' \leq y_{\max}$, we have*

$$f(x,y) \leq \max\{f(x',y'), f(x',0), f(0,y'), f(0,0)\}, \text{ and}$$
$$f(x',y') \leq \max\{f(x,y), f(x,y_{\max}), f(x_{\max},y), f(x_{\max},y_{\max})\}.$$

Proof. We show the first inequality only. The second can be proved in the same way. Since f is conic, we have $f(x,y) \leq \max\{f(x,0), f(x,y')\}$. Moreover, we have $f(x,0) \leq \max\{f(0,0), f(x',0)\}$ and $f(x,y') \leq \max\{f(0,y'), f(x',y')\}$. Thus the inequality holds. □

In the rest of the paper, we assume that any function f associated with the objective function *good* is conic, and can be evaluated in constant time.

Now we consider the complexity of finding the best substring pattern and subsequence pattern, respectively. It is not hard to show that finding the best substring pattern can be solved in polynomial time, since there are only $O(N^2)$ substrings from given sets of strings, where N is the total length of the strings, so that we can check all candidates in a trivial way. Moreover, we can solve it in linear time, by using *generalized suffix trees* [12].

Theorem 1. *We can find the best substring pattern in linear time.*

On the other hand, it is not easy to find the best subsequence pattern. First we introduce a very closely related problem.

Definition 3 (Consistency problem for subsequence patterns).
Input: *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Question: *Is there a string w that is a subsequence for each string $s \in S$, but not a subsequence for any string $t \in T$?*

The problem can be interpreted as a special case of the finding the best subsequence pattern. The next theorem shows the problem is intractable.

Theorem 2 ([13, 16, 17]). *The consistency problem for subsequence patterns is NP-complete.*

Therefore, we are essentially forced to enumerate and evaluate exponentially many subsequence patterns in the worst case, in order to find the best subsequence pattern. In the next section, we show a practical solution based on pruning search trees. Our pruning strategy utilizes the property of subsequence languages and the conicality of the function.

4 Pruning Heuristics

In this section, we introduce two pruning heuristics, inspired by Morishita and Sese [18], to construct a practical algorithm to find the best subsequence pattern.

For a conic function $f(x, y)$, we define

$$F(x, y) = \max\{f(x, y), f(x, 0), f(0, y), f(0, 0)\}, \text{ and}$$
$$G(x, y) = \max\{f(x, y), f(x, y_{\max}), f(x_{\max}, y), f(x_{\max}, y_{\max})\}.$$

Theorem 3. *For any strings $v, w \in \Sigma^*$ with $v \preceq_{seq} w$,*

$$f(x_w, y_w) \le F(x_v, y_v), \tag{1}$$
$$f(x_v, y_v) \le G(x_w, y_w). \tag{2}$$

Proof. By Lemma 1 (5), $v \preceq_{\text{seq}} w$ implies that $L^{\text{seq}}(v) \supseteq L^{\text{seq}}(w)$. Thus $x_v = |S \cap L^{\text{seq}}(v)| \ge |S \cap L^{\text{seq}}(w)| = x_w$. In the same way, we can show $y_v \ge y_w$. By Lemma 2, we have $f(x_w, y_w) \le F(x_v, y_v)$. The second inequality can be verified similarly. □

In Fig. 1, 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.

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

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 4. *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.*

Proof. First of all, we consider the behavior of the algorithm whose lines marked by '$*$' are commented out. That is, we first assume that the lines 10, 13 and 20–23 are skipped. In this case, we show that the algorithm performs the exhaustive

```
1     string $FindMaxSubsequence$(StringSet $S$, $T$, int $maxLength = \infty$)
2         string $prefix$, $seq$, $maxSeq$;
3         double $upperBound = \infty$, $maxVal = -\infty$, $val$;
4         int $x$, $y$;
5         StringSet $Forbidden = \emptyset$;
6         PriorityQueue $queue$;    /* Best First Search*/
7         $queue.push("", \infty)$;
8         while not $queue.empty()$ do
9             $(prefix, upperBound) = queue.pop()$;
10 *          if $upperBound < maxVal$ then break;
11            foreach $c \in \Sigma$ do
12                $seq = prefix +$ c;    /* string concatenation */
13 *              if $Forbidden.numOfSuperseq(seq) == 0$ then
14                    $x = S.numOfSubseq(seq)$;
15                    $y = T.numOfSubseq(seq)$;
16                    $val = f(x, y)$;
17                    if $val > maxVal$ then
18                        $maxVal = val$;
19                        $maxSeq = seq$;
20 *                  $upperBound = \max\{f(x,y), f(x,0), f(0,y), f(0,0)\}$;
21 *                  if $upperBound < maxVal$ then
22 *                      $Forbidden.append(seq)$;
23 *                  else
24                        if $|seq| < maxLength$ then
25                            $queue.push(seq, upperBound)$;
26        return $maxSeq$;
```

Fig. 1. Algorithm *FindMaxSubsequence*. In our pseudocode, indentation indicates block structure, and the **break** statement is to jump out of the closest enclosing loop.

search in a breadth first manner. Since the value of *upperBound* is unchanged, **PriorityQueue** is actually equivalent to a simple queue. The lines 14–16 evaluate the value $good(seq, S, T)$ of a string *seq*, and if it exceeds the current maximum value *maxVal*, we update *maxVal* and *maxSeq* in lines 17–19. Thus the algorithm will examine *all* strings of length at most ℓ, in increasing order of the length, and it can find the maximum.

We now consider the lines 20, 21, and 23. Let v be the string currently represented by the variable *seq*. At lines 14 and 15, x_v and y_v are computed. At line 20, $upperBound = F(x_v, y_v)$ is estimated and if *upperBound* is less than the current maximum value *maxVal*, the algorithm skips pushing v into the queue. It means that any string w of which v is a prefix will not 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

Theorem 3 (1), the value $f(x_w, y_w) \leq F(x_v, y_v)$, and since $F(x_v, y_v) < \mathit{maxVal}$, the string w can never be the maximum.

Assume the condition $\mathit{upperBound} < \mathit{maxVal}$ holds at line 10. It implies that any string v in the *queue* can never be the best subsequence, since the *queue* is a priority queue so that $F(x_v, y_v) \leq \mathit{upperBound}$, which means $f(x_v, y_v) \leq F(x_v, y_v)$ by Theorem 3 (1). Therefore $f(x_v, y_v) < \mathit{maxVal}$ for any string v in the *queue*, and we can jump out of the loop immediately.

Finally, we take account of lines 13 and 22. Initially, the set *Forbidden* of strings is empty. At line 22, a string v is appended to *Forbidden* only if $\mathit{upperBound} = F(x_v, y_v) < \mathit{maxVal}$. At line 13, if the condition

$$\mathit{Forbidden.numOfSuperseq}(\mathit{seq}) == 0$$

does not hold, *seq* will not be evaluated. Moreover, any string of which *seq* is a prefix will not be evaluated either, since we does not push *seq* in the *queue* at line 25 in this case. Nevertheless, we can show that these cuts never affect the final output as follows. Assume that $\mathit{Forbidden.numOfSuperseq}(\mathit{seq}) \neq 0$ for a string *seq*. It implies that there exists a string $u \in \mathit{Forbidden}$ such that *seq* is a supersequence of u. In another word, u is a subsequence of *seq*. Since u is in *Forbidden*, we know that $F(x_u, y_u) < \mathit{maxVal}$ at some moment. By Theorem 3 (2), the value $f(x_{seq}, y_{seq})$ can never exceeds *maxVal*. Thus the output of the algorithm is not changed by these cuts. □

By the above theorem, we can safely prune the branches. We now consider the cost of performing these heuristics. The cost of the first heuristics at lines 20, 21, and 23 is negligible, since evaluating the *upperBound* at line 20 is negligible compared to evaluate x and y at lines 14 and 15. On the other hand, the second heuristics at lines 13 and 22 may be expensive, since the evaluation of $\mathit{Forbidden.numOfSuperseq}(\mathit{seq})$ may not be so easy when the set *Forbidden* becomes large.

Anyway, one of the most time-consuming part of the algorithm is the lines 14 and 15. Here, for a string *seq*, we have to count the number of strings in the sets S and T that are subsequences of *seq*. We remark that the set S and T are fixed within the algorithm *FindMaxSubsequence*. Thus we have a possibility to speed up counting, at the cost of some appropriate preprocessing. We will discuss it in the next section.

5 Using Subsequence Automata

In this section, we pay our attention to the following problem.

Definition 4 (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 our application. In this paper, we utilize automata that accept subsequences of strings. Baeza-Yates [5] introduced the directed acyclic subsequence graph (DASG) of a string 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 string. On the other hand, Troníček and Melichar [21] showed a left-to-right algorithm for building the DASG for a single string.

We now turn our attention to the case of a set S of strings. A straightforward approach is to build DASGs for each string in S. Given a query string *seq*, we traverse all DASGs simultaneously, and return the total number of DASGs that accept *seq*. It clearly runs in $O(k|seq|)$ time, where k is the number of strings in S. When the running time is more critical, we can build a product of k DASGs so that the running time becomes $O(|seq|)$ time, at the cost of preprocessing time and space requirement. This is the DASG for a set of strings.

Baeza-Yates also presented a right-to-left algorithm for building the DASG for a set of strings [5]. Moreover, Troníček and Melichar [21], and Crochemore and Troníček [7] showed left-to-right algorithms for building the DASG for a set of strings.

In [11], 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 DASG. 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. This is the fastest algorithm to construct a subsequence automaton for a set of strings, to the best of our knowledge. We can extend the automaton so that it answers the above *Counting the matched strings* problem in a natural way (See Fig. 2).

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 $m = \Omega(n^2)$ [7].) 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 the next section, we experimentally evaluate the optimal value of the parameter *mode* in some situation.

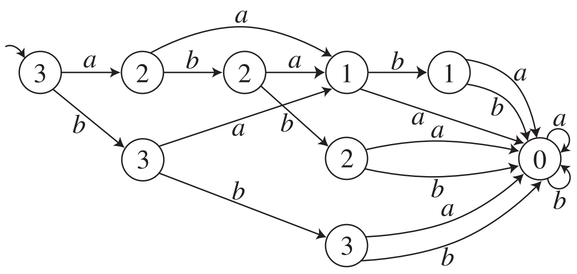

Fig. 2. Subsequence automaton for $S = \{abab, abb, bb\}$, where $\Sigma = \{a, b\}$. Each number on a state denotes the number of matched strings. For example, by traverse the states according to a string ab, we reach the state whose number is 2. It corresponds to the cardinality $|L^{\text{seq}}(ab)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

6 Implementation and Experiments

In this section, we report some results on our experiments. We are implementing our algorithm in Fig. 1 using C++ language with Standard Template Library (STL). For the **PriorityQueue**, we use the standard priority_queue in STL. Concerning with the **StringSet**, we have implemented the function *numOfSubseq* (*seq*) in the following two ways depending on the value of *mode*. In case of $mode = 0$, we do not use subsequence automata. For each string w in the set, we check whether *seq* is a subsequence of w or not in a trivial way, and return the number of matched strings. Thus we do not need to preprocess the set. For the cases $mode \geq 1$, we construct $k/mode$ subsequence automata in the preprocess, where k is the number of strings in the set. On the other hand, the function *numOfSuperseq*(*seq*) is implemented in a trivial way without using any special data structure.

We examined the following two data as input.

Transmembrane 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 [20]. The average length of the strings is about 30. S_1 consists of 70 transmembrane domains, and T_1 consists of 100 non-transmembrane domains.

DNA DNA sequences of yeast genome over $\Sigma = \{\mathtt{A}, \mathtt{T}, \mathtt{G}, \mathtt{C}\}$. The lengths of the strings are all 30. We selected two sets S_2 and T_2 based on the functional categories. $|S_2| = 31$ and $|T_2| = 35$.

We note that $\langle S_1, T_1 \rangle$ is an easy instance, while $\langle S_2, T_2 \rangle$ is a hard instance, in the sense that the best score for $\langle S_1, T_1 \rangle$ is high, while that for $\langle S_2, T_2 \rangle$ is low. As we will report, the facts affect the practical behaviors of our algorithm.

In order to verify the effect of the first heuristics and the second heuristics, we compared the searching time to find the best subsequence pattern of our algorithm.

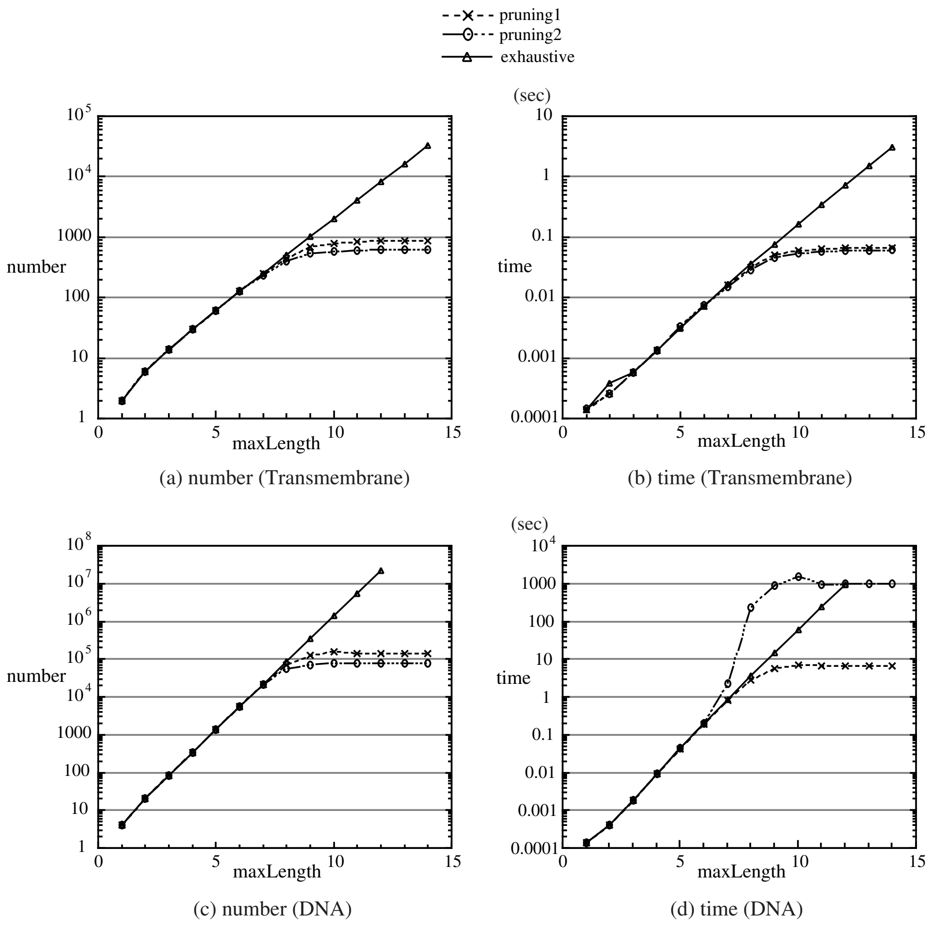

Fig. 3. Number of strings actually evaluated and running time, where maxLength varies.

pruning1 We use the first heuristics only, by commented out the lines 13 and 22.
pruning2 We use both the first and second heuristics.
exhaustive We do not use any heuristics, by commented out the lines 10, 13 and 20–23.

Our experiments were carried out both on a workstation AlphaServer DS20 with an Alpha 21264 processor at 500MHz running Tru64 UNIX operating system (WS), and on a personal computer with Pentium III processor at 733MHz running Linux (PC).

First we verified the effect of the first heuristics and the second heuristics. Fig. 3 shows the numbers of strings actually evaluated and the running time at PC, when maxLength varies and *mode* was fixed to 0. The both graphs (a)

Table 1. Preprocessing time and search time (seconds) at PC. The data is Transmembrane.

mode	0	1	2	3	4	5	6	7	8	9	10
preprocessing	-	0.023	0.054	0.120	0.273	0.470	0.796	1.378	2.108	3.083	4.543
exhaustive	1.502	1.560	0.906	0.710	0.599	0.535	0.494	0.460	0.425	0.414	0.379
pruning1	0.067	0.077	0.046	0.037	0.031	0.025	0.023	0.022	0.020	0.019	0.018
pruning2	0.060	0.069	0.047	0.040	0.035	0.033	0.031	0.030	0.029	0.029	0.028

and (c) show that the pruning2 gives the most effective pruning with respect to the number of evaluated strings, as we expected. For example, pruning2 reduces the search space approximately half compared to pruning1, when maxLength is 14 in (c). However, the running time behaves differently as we expected. The graph (b) shows that the running time reflects the number of evaluated strings, while the graph (c) shows that pruning2 was much slower than pruning1. This is because the overhead of maintaining the set *Forbidden* and the response time of the query to *Forbidden*, since we implemented it in a trivial way. By comparing (a) and (b) with (c) and (d) respectively, we see that the instance $\langle S_1, T_1 \rangle$ of Transmembrane is easy to solve compared to $\langle S_2, T_2 \rangle$ of DNA, because some short subsequences with high score were found in an early stage so that the search space is reduced drastically.

We now verify the effect of introducing subsequence automata. Table 1 shows the preprocess time, and search time for each search method, where *mode* is changed from 0 to 10. We can see that the preprocessing time increases with the *mode*, as we expected, since the total size of the automata increases. On the other hand, the search time decreases monotonically with the *mode* for any search method except the case $mode = 0$, since each subsequence query will be answered quickly by using subsequence automata. The search time in the case $mode = 1$ is slightly slower than that in the case $mode = 0$. It implies that traversing an automaton is not so faster than naive matching of subsequence when answering subsequence queries. We suppose that the phenomena arise mainly from the effect of CPU caches.

In order to see the most preferable value of *mode* at which the total running time is minimized, refer to Fig. 4 (a), (b), and (c) that illustrates Table 1. The total running time, that is the sum of preprocessing and search time, is minimized at $mode = 3$ for exhaustive search (a). On the other hand, unfortunately, for both pruning1 in (b) and pruning2 in (c), the total running time is minimized at $mode = 0$. It means that in this case, subsequence automata could not reduce the running time. Especially, at the workstation (d), search without using subsequence automata ($mode = 0$) is much faster than any other *mode*. We guess that it is also caused by the CPU caches.

By these results, we verified that the pruning heuristics and subsequence automata reduce the time to find the best subsequence pattern, independently.

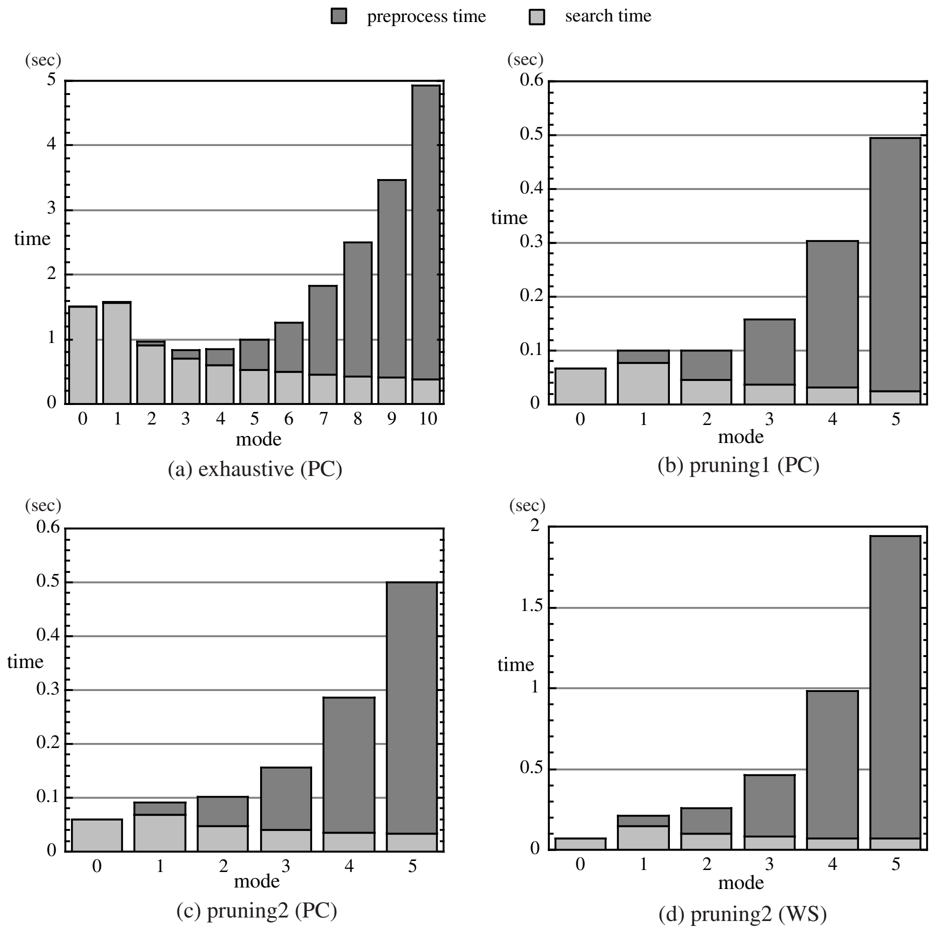

Fig. 4. Total running time of (a) exhaustive search and (b)(c)(d) pruning search. The experiments (a), (b) and (c) are performed at PC, while (d) at WS.

7 Concluding Remarks

We have discussed how to find a subsequence that maximally distinguishes given two sets of strings, according to a specified objective function. The only requirement to the objective function is the *conicality*, that is weaker than the *convexity*, and almost of all natural measures to distinguish two sets will satisfy the property.

In this paper, we focused on *finding the best* subsequence pattern. 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* [6, 9, 22].

In our current implementation, the function *numOfSuperseq* is realized in a trivial way, that slows down the pruning2 in some situation. If we can construct a *supersequence automata* efficiently, the second heuristic will be more effective.

We remark that the function G in Theorem 3 (2) is not actually used in our algorithm, since our algorithm starts from the empty string and tries to extend it. Another approach is also possible, that starts from a given string and tries to shrink it. In this case, the function G will be applicable.

In [8, 15] an *episode matching* is considered, where the total length of the matched strings is bounded by a given parameter. It will be very interesting to extend our approach to find the best *episode* to distinguish two sets of strings. Moreover, it is also challenging to apply our approach to find the best *pattern* in the sense of *pattern languages* introduced by Angulin [2], where the related consistency problems are shown to be very hard [13, 14, 17]. Arimura et al. showed an another approach to find the best *proximity pattern* [3, 4, 10]. It may be interesting to combine these approaches into one.

We plan to install our algorithm into the core of the decision tree generator in the BONSAI system [20].

Acknowledgements

The authors would like to thank Prof. Albert Apostolico and Prof. Hiroki Arimura for fruitful discussion.

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. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21(1):46–62, Aug. 1980.
3. H. Arimura and S. Shimozono. Maximizing agreement with a classification by bounded or unbounded number of associated words. In *Proc. of 9th Annual International Symposium on Algorithms and Computation*, volume 1533 of *Lecture Notes in Computer Science*. Springer-Verlag, Dec. 1998.
4. H. Arimura, A. Wataki, R. Fujino, and S. Arikawa. A fast algorithm for discovering optimal string patterns in large text databases. In *Proc. the 8th International Workshop on Algorithmic Learning Theory*, volume 1501 of *Lecture Notes in Artificial Intelligence*, pages 247–261. Springer-Verlag, Oct. 1998.
5. R. A. Baeza-Yates. Searching subsequences. *Theoretical Computer Science*, 78(2):363–376, Jan. 1991.
6. A. Califano. SPLASH: Structural pattern localization analysis by sequential histograms. *Bioinformatics*, Feb. 1999.
7. M. Crochemore and Z. Troníček. Directed acyclic subsequence graph for multiple texts. Technical Report IGM-99-13, Institut Gaspard-Monge, June 1999.
8. 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.

9. 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*, Lecture Notes in Computer Science, pages 167–174. AAAI Press, Aug. 1997.
10. 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.
11. 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).
12. L. C. K. Hui. Color set problem with applications to string matching. In *Proc. 3rd Annual Symposium on Combinatorial Pattern Matching*, volume 644 of *Lecture Notes in Computer Science*, pages 230–243. Springer-Verlag, 1992.
13. 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.
14. K.-I. Ko and W. Tzeng. Three Σ_2^p-complete problems in computational learning theory. *Computational Complexity*, 1(3):269–310, 1991.
15. 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.
16. 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.
17. S. Miyano, A. Shinohara, and T. Shinohara. Polynomial-time learning of elementary formal systems. *New Generation Computing*, 18:217–242, 2000.
18. 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, May 2000.
19. J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
20. 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.
21. Z. Troníček and B. Melichar. Directed acyclic subsequence graph. In *Proc. of the Prague Stringology Club Workshop '98*, pages 107–118, Sept. 1998.
22. 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.