Discovering Best Variable-Length-Don't-Care Patterns

Shunsuke Inenaga[1], Hideo Bannai[3], Ayumi Shinohara[1,2], Masayuki Takeda[1,2], and Setsuo Arikawa[1]

[1] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
[2] PRESTO, Japan Science and Technology Corporation (JST)
{s-ine, ayumi, takeda, arikawa}@i.kyushu-u.ac.jp
[3] Human Genome Center, University of Tokyo, Tokyo 108-8639, Japan
bannai@ims.u-tokyo.ac.jp

Abstract. A *variable-length-don't-care pattern* (*VLDC pattern*) is an element of set $\Pi = (\Sigma \cup \{\star\})^*$, where Σ is an alphabet and $\star$ is a wildcard matching any string in Σ^*. Given two sets of strings, we consider the problem of finding the VLDC pattern that is the most common to one, and the least common to the other. We present a practical algorithm to find such best VLDC patterns exactly, powerfully sped up by pruning heuristics. We introduce two versions of our algorithm: one employs a *pattern matching machine* (*PMM*) whereas the other does an index structure called the *Wildcard Directed Acyclic Word Graph* (*WDAWG*). In addition, we consider a more generalized problem of finding the best pair $\langle q, k\rangle$, where k is the *window size* that specifies the length of an occurrence of the VLDC pattern q matching a string w. We present three algorithms solving this problem with pruning heuristics, using the *dynamic programming* (*DP*), PMMs and WDAWGs, respectively. Although the two problems are NP-hard, we experimentally show that our algorithms run remarkably fast.

1 Introduction

A vast amount of data is available today, and discovering useful *rules* from those data is quite important. Very commonly, information is stored and manipulated as *strings*. In the context of strings, rules are *patterns*. Given two sets of strings, often referred to as *positive examples* and *negative examples*, it is desired to find the pattern that is the most common to the former and the least common to the latter. This is a critical task in Discovery Science as well as in Machine Learning.

A string y is said to be a *substring* of a string w if there exist strings $x, z \in \Sigma^*$ such that $w = xyz$. Substring patterns are possibly the most basic patterns to be used for the separation of two sets S, T of strings. Hirao et al. [8] stated that such best substrings can be found in linear time by constructing the *suffix tree* for $S \cup T$ [12,21,7]. They also considered *subsequence patterns* as rules for separation. A subsequence pattern p is said to *match* a string w if p can be obtained by removing zero or more characters from w [2]. Against the fact that finding

S. Lange, K. Satoh, and C.H. Smith (Eds.): DS 2002, LNCS 2534, pp. 86–97, 2002.

the best subsequence patterns to separate given two sets of strings is NP-hard, they proposed an algorithm to solve the problem with practically reasonable performance. More recently, an efficient algorithm to discover the best *episode patterns* was proposed in [9]. An episode pattern $\langle p, k \rangle$, where p is a string and k is an integer, is said to *match* a string w if p is a subsequence of a substring u of w with $|u| \leq k$ [14,6,20]. The problem to find the best episode patterns is also known to be NP-hard.

In this paper, we focus on a pattern containing a *wildcard* that matches any string. The wildcard is called a *variable length don't care* and is denoted by $\star$. A *variable-length-don't-care pattern* (*VLDC pattern*) is an element of $\Pi = (\Sigma \cup \{\star\})^*$, and is also sometimes called a *regular pattern* as in [19]. When $a, b \in \Sigma$, $ab\star bb\star ba$ is an example of a VLDC pattern and, for instance, matches string $abbbbaaaba$ with the first and second $\star$'s replaced by b and aaa, respectively. The *language* $L(q)$ of a pattern $q \in \Pi$ is the set of strings obtained by replacing $\star$'s in q with arbitrary strings. Namely, $L(ab\star bb\star ba) = \{abubbvba \mid u, v \in \Sigma^*\}$. The class of this language corresponds to a class of the *pattern languages* proposed by Angluin [1]. VLDC patterns are generalization of substring patterns and subsequence patterns. For instance, consider a pattern string $abc \in \Sigma^*$. The substring matching problem corresponding to the pattern is given by the VLDC pattern $\star abc\star$. Also, the VLDC pattern $\star a\star b\star c\star$ leads to the subsequence pattern matching problem.

This paper is devoted to introducing a practical algorithm to discover the best VLDC pattern to distinguish two given sets S, T of strings. To speed up the algorithm, firstly we restrict the search space by means of pruning heuristics inspired by Morishita and Sese [16]. Secondly, we accelerate the matching phase of the algorithm in two ways, as follows: In [11], we introduced an *index structure* called the *Wildcard Directed Acyclic Word Graph* (*WDAWG*). The WDAWG for a text string w recognizes all possible VLDC patterns matching w, and thus enables us to examine whether a given VLDC pattern q matches w in $O(|q|)$ time. More recently, a space-economical version of its construction algorithm was presented in [10]. We use WDAWGs for quick matching of VLDC patterns. Another approach is to preprocess a given VLDC pattern q, building a DFA accepting $L(q)$. We use it as a *pattern matching machine* (*PMM*) which runs over a text string w and determines whether or not q matches w in $O(|w|)$ time.

We furthermore propose a generalization of the VLDC pattern matching problem. That is, we introduce an integer k called the *window size* which specifies the length of an occurrence of a VLDC pattern that matches $w \in \Sigma^*$. The introduction of k leads to the generalization of the episode patterns as well. Specifying the length of an occurrence of a VLDC pattern is of great significance especially when classifying *long* strings over a *small* alphabet, since a short VLDC pattern surely matches most long strings. Therefore, for example, when two sets of biological sequences are given to be separated, this approach is adequate and promising. Pruning heuristic to speed up our algorithm finding the best pair $\langle q, k \rangle$ is also presented. We propose three approaches effective in computing the best pair, using the *dynamic programming*, PMMs, and WDAWGs, respectively.

We declare that this work generalizes and outperforms the ones accomplished in [8,9], since it is capable of discovering more advanced and useful patterns. In fact, we show some experimental results that convince us of the accuracy of our algorithms as well as their fast performances. Moreover, we are now installing our algorithms into the core of the decision tree generator in BONSAI [17], a powerful machine discovery system.

We here only give basic ideas for our pruning heuristics, that are rather straightforward extensions of those developed in our previous work [8,9]. Interested readers are invited to refer to our survey report [18].

2 Finding the Best Patterns to Separate Sets of Strings

2.1 Notation

Let $\mathcal{N}$ be the set of integers. Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. The substring of a string w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i : j] = \varepsilon$ for $j < i$. The *reversal* of a string w is denoted by w^R, that is, $w^R = w[n]w[n-1] \ldots w[1]$ where $n = |w|$.

For a set $S \subseteq \Sigma^*$ of strings, the number of strings in S is denoted by $|S|$ and the total length of strings in S is denoted by $\|S\|$.

Let $\Pi = (\Sigma \cup \{\star\})^*$, where $\star$ is a *variable length don't care* matching any string in Σ^*. An element $q \in \Pi$ is a *variable-length-don't-care pattern* (*VLDC pattern*). For example, $\star a \star ab \star ba \star$ is a VLDC pattern with $a, b \in \Sigma$. We say a VLDC pattern q *matches* a string w if w can be obtained by replacing $\star$'s in q with some strings. In the running example, the VLDC-pattern $\star a \star ab \star ba \star$ matches string $abababbbaa$ with the $\star$'s replaced by ab, b, b and a, respectively. For any $q \in \Pi$, $|q|$ denotes the sum of numbers of characters and $\star$'s in q.

2.2 Finding the Best VLDC Patterns

We write as $q \preceq u$ if u can be obtained by replacing $\star$'s in q with arbitrary elements in Π.

Definition 1. *For a VLDC pattern $q \in \Pi$, we define $L(q)$ by*

$$L(q) = \{w \in \Sigma^* \mid q \preceq w\}.$$

According to the above definition, we have the following lemma.

Lemma 1. *For any $q, u \in \Pi$, if $q \preceq u$, then $L(q) \supseteq L(u)$.*

Let *good* be a function from $\Sigma^* \times 2^{\Sigma^*} \times 2^{\Sigma^*}$ to the set of real numbers. In what follows, we formulate the problem to solve.

Definition 2 (Finding the best VLDC pattern according to *good*).
Input: *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output: *A VLDC pattern $q \in \Pi$ that maximizes the score of $good(q, S, T)$.*

Intuitively, the score of $good(q, S, T)$ expresses the "goodness" of q in the sense of distinguishing S from T. The definition of *good* varies with applications. For examples, the χ^2 values, entropy information gain, and gini index can be used. Essentially, these statistical measures are defined by the numbers of strings that satisfy the rule specified by q. Any of the above-mentioned measures can be expressed by the following form:

$$\begin{aligned} good(q, S, T) &= f(x_q, y_q, |S|, |T|), \text{ where} \\ x_q &= |S \cap L(q)|, \\ y_q &= |T \cap L(q)|. \end{aligned}$$

When S and T are fixed, $|S|$ and $|T|$ are regarded as constants. On this assumption, we abbreviate the notation of the function to $f(x, y)$ in the sequel.

We say that a function f from $[0, x_{\max}] \times [0, y_{\max}]$ to real numbers is *conic* if

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

In the sequel, we assume that f is conic and can be evaluated in constant time. The optimization problem to be tackled follows.

Definition 3 (Finding the best VLDC pattern according to f).
Input: *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output: *A VLDC pattern $q \in \Pi$ that maximizes the score of $f(x_q, y_q)$, where $x_q = |S \cap L(q)|$ and $y_q = |T \cap L(q)|$.*

The problem is known to be NP-hard [15], and thus we essentially have exponentially many candidates. Therefor, we reduce the number of candidates by using the pruning heuristic inspired by Morishita and Sese [16].

The following lemma derives from the conicality of function f.

Lemma 2 ([8]). *For any $0 \le x < x' \le x_{\max}$ and $0 \le y < y' \le y_{\max}$, we have $f(x, y) \le \max\{f(x', y'), f(x', 0), f(0, y'), f(0, 0)\}$.*

By Lemma 1 and Lemma 2, we have the next lemma, basing on which we can perform the pruning heuristic to speed up our algorithm.

Lemma 3. *For any two VLDC patterns $q, u \in \Pi$, if $q \preceq u$, then $f(x_u, y_u) \le \max\{f(x_q, y_q), f(x_q, 0), f(0, y_q), f(0, 0)\}$.*

2.3 Finding the Best VLDC Patterns within a Window

We here consider a natural extension of the problem mentioned previously. We introduce an integer k called the *window size*. Let $q \in \Pi$ and $q[i], q[j]$ be the first and last characters in q that are not $\star$, respectively, where $1 \le i \le j \le |q|$. If q matches $w \in \Sigma^*$, let $w[i'], w[j']$ be characters to which $q[i]$ and $q[j]$ can correspond, respectively, where $1 \le i' \le j' \le |w|$. (Note that we might have more than one combination of i' and j'.) If there exists a pair i', j' satisfying $j' - i' < k$, we say that q *occurs* w within a window of size k. Then the pair $\langle q, k \rangle$ is said to *match* the string w.

Definition 4. *For a pair $\langle q, k \rangle$ with $q \in \Pi$ and $k \in \mathcal{N}$, we define $L'(\langle q, k \rangle)$ by*

$$L'(\langle q, k \rangle) = \{w \in \Sigma^* \mid \ \langle q, k \rangle \textit{ matches } w\}.$$

According to the above definition, we have the following lemma.

Lemma 4. *For any $\langle q, k \rangle$ and $\langle p, j \rangle$ with $q, p \in \Pi$ and $k, j \in \mathcal{N}$, if $q \preceq p$ and $j \ge k$, then $L'(\langle q, k \rangle) \supseteq L'(\langle p, j \rangle)$.*

The problem to be tackled is formalized as follows.

Definition 5 (Finding the best VLDC pattern and window size according to f).
Input: *Two sets $S, T \subseteq \Sigma^*$ of strings.*
Output: *A pair $\langle q, k \rangle$ with $q \in \Pi$ and $k \in \mathcal{N}$ that maximizes the score of $f(x_{\langle q,k \rangle}, y_{\langle q,k \rangle})$, where $x_{\langle q,k \rangle} = |S \cap L'(\langle q, k \rangle)|$ and $y_{\langle q,k \rangle} = |T \cap L'(\langle q, k \rangle)|$.*

We stress that the value of k is *not* given beforehand, i.e., we compute not only q but also k with which the score of function f is maximum. Therefore, the search space of this problem is $\Pi \times \mathcal{N}$, while that of the problem in Definition 3 is Π. We remark that this problem is also NP-hard.

By Lemma 4 and Lemma 2, we achieve the following lemma that plays a key role for the heuristic to prune the search tree.

Lemma 5. *For any $\langle q, k \rangle$ and $\langle p, j \rangle$ with $q, u \in \Pi$ and $k, j \in \mathcal{N}$, if $q \preceq u$ and $j \ge k$, $f(x_{\langle p,j \rangle}, y_{\langle p,j \rangle}) \le \max\{f(x_{\langle q,k \rangle}, y_{\langle q,k \rangle}), f(x_{\langle q,k \rangle}, 0), f(0, y_{\langle q,k \rangle}), f(0, 0)\}$.*

3 Efficient Match of VLDC Patterns

Definition 6 (Counting the matched VLDC patterns).
Input: *A set $S \subseteq \Sigma^*$ of strings.*
Query: *A VLDC pattern $q \in \Pi$.*
Answer: *The cardinality of set $S \cap L(q)$.*

This is a sub-problem of the one given in Definition 3. It must be answered as fast as possible, since we are given quite many VLDC patterns as queries. Here, we utilize two practical methods which allows us to answer the problem quickly.

3.1 Using a DFA for a VLDC Pattern

Our first idea is to use a deterministic finite-state automaton (DFA) for a pattern. Given a VLDC pattern $q \in \Pi$, we construct a DFA that accepts $L(q)$ and use it as a pattern matching machine (PMM) which runs over text strings in S. For any $q \in \Pi$, a DFA can be constructed in $O(|q|)$ time.

Lemma 6. *Let $S \subseteq \Sigma^*$ and $q \in \Pi$. Then $|S \cap L(q)|$ can be computed in $O(|q|)$ preprocessing time and in $O(\|S\|)$ running time.*

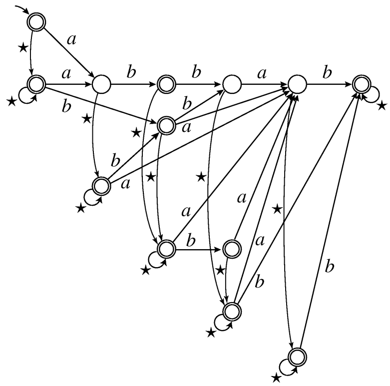

Fig. 1. *WDAWG*(w) where $w = abbab$.

3.2 Using Wildcard Directed Acyclic Word Graphs

The second approach is to use an index structure for a text string $w \in S$ that recognizes all VLDC patterns matching w.

The *Directed Acyclic Word Graph* (*DAWG*) is a classical, textbook index structure [5], invented by Blumer et al. in [3]. The DAWG of a string $w \in \Sigma^*$ is denoted by $DAWG(w)$, and is known to be the smallest deterministic automaton that recognizes all suffixes of w [4]. By means of $DAWG(w)$, we can examine whether or not a given pattern $p \in \Sigma^*$ is a substring of w in $O(|p|)$ time.

Recently, we introduced *Minimum All-Suffixes Directed Acyclic Word Graphs* (*MASDAWGs*) [11]. The MASDAWG of a string $w \in \Sigma^*$, which is denoted by $MASDAWG(w)$, is the minimization of the collection of the DAWGs for all suffixes of w. More precisely, $MASDAWG(w)$ is the smallest automaton with $|w|+1$ initial nodes, in which the directed acyclic graph induced by all reachable nodes from the k-th initial node conforms with the DAWG of the k-th suffix of w.

Several important applications of MASDAWGs were given in [11], one of which corresponds to a significantly time-efficient solution to the VLDC pattern matching problem. Namely, a variant of $MASDAWG(w)$, called *Wildcard DAWG* (*WDAWG*) of w and denoted by $WDAWG(w)$, was introduced in [11]. $WDAWG(w)$ is the smallest automaton that accepts all VLDC patterns matching w. $WDAWG(w)$ with $w = abbab$ is displayed in Fig. 1.

Theorem 1. *When* $|\Sigma| \geq 2$, *the number of nodes of* $WDAWG(w)$ *for a string* w *is* $\Theta(|w|^2)$. *It is* $\Theta(|w|)$ *for a unary alphabet.*

Theorem 2. *For any string* $w \in \Sigma^*$, $WDAWG(w)$ *can be constructed in time linear in its size.*

For all strings in $S \subseteq \Sigma^*$, we construct WDAWGs. Then we obtain the following lemma that is a counterpart of Lemma 6.

Lemma 7. *Let* $S \subseteq \Sigma^*$ *and* $q \in \Pi$. *Let* $N = \sum_{w \in S} |w|^2$. *Then* $|S \cap L(q)|$ *can be computed in* $O(N)$ *preprocessing time and in* $O(|q| \cdot |S|)$ *running time.*

In spite of the quadratic space requirement of WDAWGs, it is meaningful to construct them because of the following reasons. Assume that, for a string w in S, a VLDC pattern q has been recognized by $\mathit{WDAWG}(w)$. We then memorize the node at which q was accepted. It allows us a rapid search of any VLDC pattern qr with $r \in \Pi$, since we only need to move $|r|$ transitions from the memorized node. Therefore, WDAWGs are significantly useful especially in our situation. Moreover, WDAWGs are also helpful for pruning the search tree. Once knowing that a VLDC pattern q does not match any string in S by using the WDAWGs, we need not consider any $u \in \Pi$ such that $q \preceq u$.

4 How to Compute the Best Window Size

Definition 7 (Computing the best window size according to f).
Input: *Two sets $S, T \subseteq \Sigma^*$ of strings and a VLDC pattern $q \in \Pi$.*
Output: *An integer $k \in \mathcal{N}$ that maximizes the score of $f(x_{\langle q,k \rangle}, y_{\langle q,k \rangle})$, where $x_{\langle q,k \rangle} = |S \cap L'(\langle q, k \rangle)|$ and $y_{\langle q,k \rangle} = |T \cap L'(\langle q, k \rangle)|$.*

This is a sub-problem of the one in Definition 5, where a VLDC pattern is given beforehand.

Let ℓ be the length of the longest string in $S \cup T$. A short consideration reveals that, as candidates for k, we only have to consider the values from $|q|$ up to ℓ, which results in a rather straightforward solution. In addition to that, we give a more efficient computation method, whose basic principle originates in [9].

For a string $u \in \Sigma^*$ and a VLDC pattern $q \in \Pi$, we define the *threshold value* θ of q for u by

$$\theta_{u,q} = \min\{k \in \mathcal{N} \mid u \in L'(\langle q, k \rangle)\}.$$

If there is no such value, let $\theta_{u,q} = \infty$. Note that $u \notin L'(\langle q, k \rangle)$ for any $k < \theta$ and $u \in L'(\langle q, k \rangle)$ for any $k \geq \theta$. The set of threshold values for $q \in \Pi$ with respect to $S \subseteq \Sigma^*$ is defined as $\Theta_{S,q} = \{\theta_{u,q} \mid u \in S\}$.

A key observation is that the best window size for given $S, T \subseteq \Sigma^*$ and a VLDC pattern $q \in \Pi$ can be found in set $\Theta_{S,q} \cup \Theta_{T,q}$ without loss of generality. Thus we can restrict the search space for the best window size to $\Theta_{S,q} \cup \Theta_{T,q}$. It is therefore important to quickly solve the following sub-problem.

Definition 8 (Computing the minimum window size).
Input: *A string $w \in \Sigma^*$ and a VLDC pattern $q \in \Pi$.*
Output: *The threshold value $\theta_{w,q}$.*

We here show our three approaches to efficiently solve the above sub-problem. The first is to adopt the standard dynamic programming method. For a string $w \in \Sigma^*$ with $|w| = n$ and a pattern $q \in \Pi$ with $|q| = m$, let d_{ij} be the length of the shortest suffix of $w[1:j]$ that $q[1:i]$ matches, where $0 \leq i \leq m$ and $0 \leq j \leq n$. We can compute all d_{ij}'s in $O(mn)$ time, basing on the following recurrences: $d_{00} = 0$,

$$d_{0j} = \begin{cases} 0 & \text{if } q[1] = \star \\ \infty & \text{otherwise} \end{cases} \quad \text{for } j \geq 1,$$

$$d_{i0} = \begin{cases} d_{i-1,0} & \text{if } q[1] = \star \\ \infty & \text{otherwise} \end{cases} \quad \text{for } i \geq 1, \text{ and}$$

$$d_{ij} = \begin{cases} \min\{d_{i-1,j-1}+1, d_{i,j-1}+1, d_{i-1,j}\} & \text{if } q[i] = \star \\ d_{i-1,j-1}+1 & \text{if } q[i] = w[j] \\ \infty & \text{otherwise} \end{cases} \quad \text{for } i \geq 1 \text{ and } j \geq 1.$$

Then $\theta_{w,q} = \begin{cases} \min_{1 \leq j \leq n}\{d_{mj}\} & \text{if } q[m] = \star \\ d_{mn} & \text{otherwise.} \end{cases}$

Remark that if the row d_{mj} $(1 \leq j \leq n)$ is memorized, it will save the computation time for any pattern qr with $r \in \Pi$.

The second approach is to preprocess a given VLDC pattern $q \in \Pi$. We construct a DFA accepting $L(q)$ and another DFA for $L(q^R)$, and utilize them as PMMs running over a given string $w \in \Sigma^*$. If $q[1] \neq \star$ ($q[m] \neq \star$, respectively), we have only to compute the shortest prefix (suffix, respectively) of w that q matches and return its length. We now consider the case $q[1] = q[m] = \star$. Firstly, we run the DFA for $L(q)$ over w. Suppose that q is recognized between positions i and j in w, where $1 \leq i < j \leq |w|$ and $j - i > |q|$. A delicate point is that it is unsure whether $w[i : j]$ corresponds to the shortest occurrence of q ending at position j. How can we find the shortest one? It can be found by running the DFA for $L(q^R)$ *backward*, over w from position j. Assume that q^R is recognized at position k, where $i \leq k < j - |q|$. Then $w[k : j]$ corresponds to the shortest occurrence of q ending at position j. After that, we resume the running of the DFA of $L(q)$ from position $k + 1$, and continue the above procedure until encountering position $|w|$. The pair of positions of the shortest distance gives the threshold value $\theta_{w,q}$. This method is feasible in $O(m)$ preprocessing time and in $O(mn)$ running time, where $m = |q|$ and $n = |w|$.

The third approach is to preprocess a text string $w \in \Sigma^*$, i.e., we construct $WDAWG(w)$ and $WDAWG(w^R)$. For any $w \in \Sigma^*$, each and every node of $WDAWG(w)$ can be associated with a position in w [11]. Thus we can perform a procedure similar to the second approach above, which enables us to find the threshold value $\theta_{w,q}$. This approach takes us $O(n)$ preprocessing time and $O(mn)$ running time, where $m = |q|$ and $n = |w|$.

As a result, we obtain the following:

Lemma 8. *Let $w \in \Sigma^*$ and $q \in \Pi$ with $|w| = n$ and $|q| = m$. The threshold value $\theta_{w,q}$ can be computed in $O(mn)$ running time.*

5 Computational Experiments

The algorithms were implemented in the Objective Caml Language. All calculations were performed on a Desktop PC with dual Xeon 2.2GHz CPU (though our algorithms only utilize single CPU) with 1GB of main memory running Debian Linux. In all the experiments, the entropy information gain is used as the score for which the search is conducted.

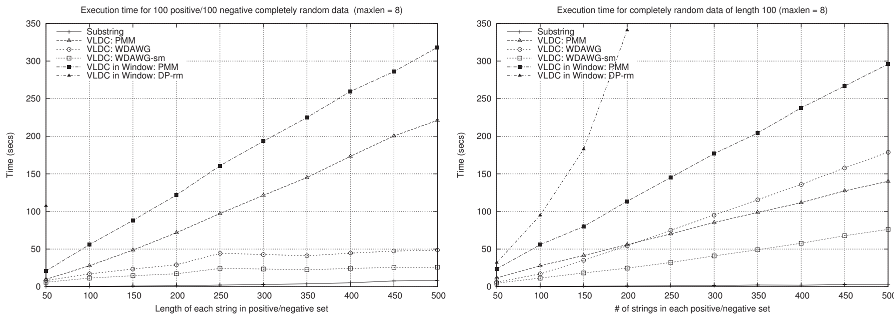

Fig. 2. Execution time (in seconds) for artificial data for: different lengths of the examples (left) different number of examples in each positive/negative set (right). The maximum length of patterns to be searched for is set to 8. WDAWG-sm is matching using the WDAWG with state memoization. DP-rm is matching using the dynamic programming table with row memoization. Only one point is shown for DP-rm in the left graph, since a greater size caused memory swapping, and the computation was not likely to end in a reasonable amount of time.

5.1 Artificial Data

We first tested our algorithms on an artificial dataset. The datasets were created as follows: The alphabet was set to $\Sigma = \{a, b, c, d\}$. We then randomly generate strings over Σ of length l. We created 3 types of datasets: 1) a completely random set, 2) a set where a randomly chosen VLDC pattern $\star ccd \star a \star ddad \star$ is embedded in the positive examples, and 3) a set where a pair of a VLDC pattern and a window size $\langle \star ccd \star a \star ddad \star, 19 \rangle$ is embedded in the positive examples. In 2) and 3), a randomly generated string is used as a positive example if the pattern matches it, and used as a negative example otherwise, until both positive and negative set sizes are n. Examples for which the set size exceeds n are discarded.

Fig. 2 shows the execution times for different l and n, for the completely random dataset. We can see that the execution time grows linearly in n and l as expected, although the effect of pruning seems to take over for VLDC patterns in the left graph, when the length of each sequence is long. Searching for VLDC patterns and window sizes using dynamic programming with row memoization, does not perform very well.

Fig. 3 shows the execution times for different maximum lengths of VLDC patterns to look for, for the 3 datasets (The length of a VLDC pattern is defined as the length of the pattern representation, excluding any $\star$'s on the ends). We can see that the execution time grows exponentially as we increase the maximum pattern length searched for, until the pruning takes effect. The lower left graph in Fig. 3 shows the effect of performance of an exhaustive search, run on the completely random dataset, compared to searches with the branch and bound pruning for the different datasets. The pruning is more effective when it is more likely to have a good solution.

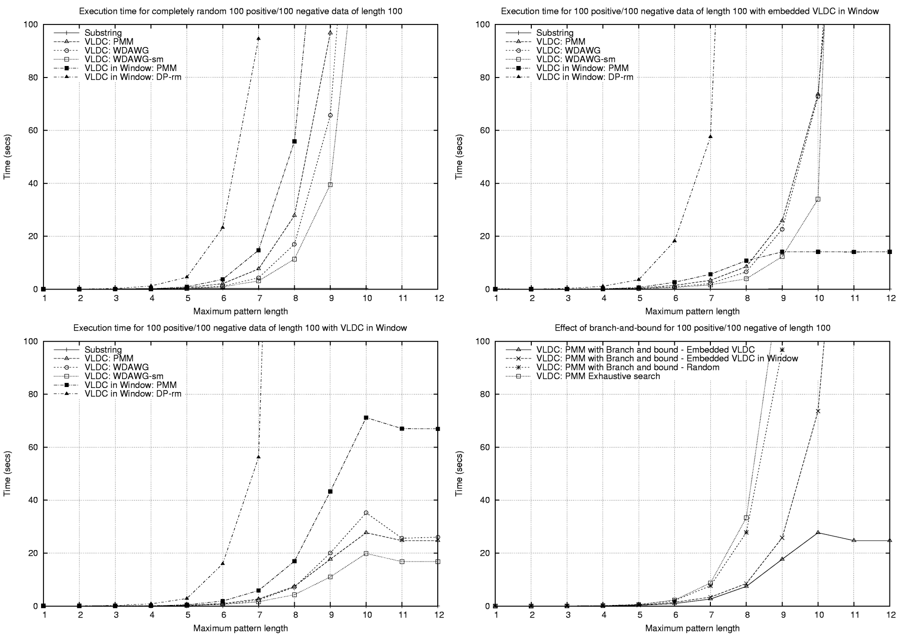

Fig. 3. Execution time (in seconds) for artificial data for different maximum lengths of patterns to be searched for with: completely random data (upper left), VLDC and window size embedded data (upper right), VLDC embedded data (lower left). The lower right graph shows the effect of pruning of the search space for the different data sets, compared to exhaustive search on the completely random dataset.

5.2 Real Data

To show the usefulness of VLDC patterns and windows, we also tested our algorithms on actual protein sequences. We use the data available at `http://www.cbs.dtu.dk/services/TargetP/`, which consists of protein sequences which are known to contain *protein sorting signals*, that is, (in many cases) a short amino acid sequence segment which holds the information which enables the protein to be carried to specified compartments inside the cell. The dataset for plant proteins consisted of: 269 sequences with signal peptide (SP), 368 sequences with mitocondrial targeting peptide (mTP), 141 sequences with chloroplast transit peptide (cTP), and 162 "Other" sequences. The average length of the sequences was around 419, and the alphabet is the set of 20 amino acids.

Using the signal peptides as positive examples, and all others as negative examples, we searched for the best pair $\langle p, k\rangle$ with maximum length of 10 using PMMs. To limit the alphabet size, we classify the amino acids into 3 classes $\{0, 1, 2\}$, according to the hydropathy index [13]. The most hydrophobic amino acids {A, M, C, F, L, V, I} (hydropathy ≥ 0.0) are converted to 0, {P,Y,W,S,T,G} ($-3.0 \leq$ hydropathy < 0.0) to 1, and {R, K, D, E, N, Q, H} (hydropathy < -3.0) to 2. We obtained the pair $\langle 0{\star}00{\star}00000{\star}, 26\rangle$, which occurs in $213/269 = 79.2\%$ of the sequences with SP, and $26/671 = 3.9\%$ of the other sequences. The calculation took exactly 50 minutes. This pattern can be interpreted as capturing the well known hydrophobic h-region of SP [22]. Also, the VLDC pattern suggests that the match occurs in the first 26 amino acid residues of the protein, which is natural since SP, mTP, cTP are known to be *N-terminal sorting signals*, that is, they are known to appear near the head of the protein sequence. A best substring search quickly finds the pattern ${\star}00000001{\star}$ in 36 seconds, but only gives us a classifier that matches $152/269 = 56.51\%$ of the SP sequences, and $41/671 = 6.11\%$ of the others.

For another example, we use the mTP set as positive examples, and the SP and Other sets as negative examples. This time, we convert the alphabet according to the net charge of the amino acid. Amino acids {D, E} (negative charge) are converted to 0, {K, R} (positive charge) to 1, and the rest {A, L, N, M, F, C, P, Q, S, T, G, W, H, Y, I, V} to 2. The calculation took about 21 minutes and we obtain the pair $\langle 2{\star}1{\star}1{\star}2221{\star}, 28\rangle$ which occurs in $334/368 = 90.76\%$ of the mTP sequences and $(73/431 = 16.94\%)$ of the SP and Other sequences. This pattern can also be regarded as capturing existing knowledge about mTPs [23]: They are fairly abundant in K or R, but do not contain much D or E. The pattern also suggests a periodic appearance of K or R, which is a characteristic of an amphiphilic α-helix that mTPs are reported to have. A best substring search finds pattern ${\star}212221{\star}$ in 20 seconds, which gives us a classifier that matches $318/368 = 86.41\%$ of sequences with mTP and $255/431 = 59.16\%$ of the other sequences.

References

1. D. Angluin. Finding patterns common to a set of strings. *J. Comput. Sys. Sci.*, 21:46–62, 1980.
2. R. A. Baeza-Yates. Searching subsequences (note). *Theoretical Computer Science*, 78(2):363–376, Jan. 1991.
3. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.
4. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.
5. M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, New York, 1994.
6. G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Kärkkäinen. Episode matching. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM'97)*, volume 1264 of *Lecture Notes in Computer Science*, pages 12–27. Springer-Verlag, 1997.
7. D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.
8. M. Hirao, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best subsequence patterns. In *Proc. The Third International Conference on Discovery Science*, volume 1967 of *Lecture Notes in Artificial Intelligence*, pages 141–154. Springer-Verlag, 2000.

9. M. Hirao, S. Inenaga, A. Shinohara, M. Takeda, and S. Arikawa. A practical algorithm to find the best episode patterns. In *Proc. The Fourth International Conference on Discovery Science*, volume 2226 of *Lecture Notes in Artificial Intelligence*, pages 435–440. Springer-Verlag, 2001.
10. S. Inenaga, A. Shinohara, M. Takeda, H. Bannai, and S. Arikawa. Space-economical construction of index structures for all suffixes of a string. In *Proc. 27th International Symposium on Mathematical Foundations of Computer Science (MFCS'02)*, Lecture Notes in Computer Science. Springer-Verlag, 2002. To appear.
11. 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 Annual Symposium on Combinatorial Pattern Matching (CPM'02)*, volume 2373 of *Lecture Notes in Computer Science*, pages 153–167. Springer-Verlag, 2002.
12. S. R. Kosaraju. Fast pattern matching in trees. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 178–183, 1989.
13. J. Kyte and R. Doolittle. A simple method for displaying the hydropathic character of a protein. *J. Mol. Biol.*, 157:105–132, 1982.
14. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episode in sequences. In *Proc. 1st International Conference on Knowledge Discovery and Data Mining*, pages 210–215. AAAI Press, 1995.
15. S. Miyano, A. Shinohara, and T. Shinohara. Polynomial-time learning of elementary formal systems. *New Generation Computing*, 18:217–242, 2000.
16. 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, 2000.
17. 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, 1994.
18. A. Shinohara, M. Takeda, S. Arikawa, M. Hirao, H. Hoshino, and S. Inenaga. Finding best patterns practically. In *Progress in Discovery Science*, volume 2281 of *Lecture Notes in Artificial Intelligence*, pages 307–317. Springer-Verlag, 2002.
19. T. Shinohara. Polynomial-time inference of pattern languages and its applications. In *Proc. 7th IBM Symp. Math. Found. Comp. Sci.*, pages 191–209, 1982.
20. Z. Troníček. Episode matching. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01)*, volume 2089 of *Lecture Notes in Computer Science*, pages 143–146. Springer-Verlag, 2001.
21. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
22. G. von Heijne. The signal peptide. *J. Membr. Biol.*, 115:195–201, 1990.
23. G. von Heijne, J. Steppuhn, and R. G. Herrmann. Domain structure of mitochondrial and chloroplast targeting peptides. *Eur. J. Biochem.*, 180:535–545, 1989.