Finding Optimal Pairs of Cooperative and Competing Patterns with Bounded Distance

Shunsuke Inenaga[1], Hideo Bannai[2], Heikki Hyyrö[3], Ayumi Shinohara[3,5],
Masayuki Takeda[4,5], Kenta Nakai[2], and Satoru Miyano[2]

[1] Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23),
FIN-00014 University of Helsinki, Finland
inenaga@cs.helsinki.fi

[2] Human Genome Center, Institute of Medical Science,
The University of Tokyo, Tokyo 108-8639, Japan
{bannai,knakai,miyano}@ims.u-tokyo.ac.jp

[3] PRESTO, Japan Science and Technology Agency (JST)
helmu@cs.uta.fi

[4] SORST, Japan Science and Technology Agency (JST)

[5] Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
{ayumi,takeda}@i.kyushu-u.ac.jp

Abstract. We consider the problem of discovering the optimal pair of substring patterns with bounded distance α, from a given set S of strings. We study two kinds of pattern classes, one is in form $p \wedge_\alpha q$ that are interpreted as *cooperative* patterns within α distance, and the other is in form $p \wedge_\alpha \neg q$ representing *competing* patterns, with respect to S. We show an efficient algorithm to find the optimal pair of patterns in $O(N^2)$ time using $O(N)$ space. We also present an $O(m^2N^2)$ time and $O(m^2N)$ space solution to a more difficult version of the optimal pattern pair discovery problem, where m denotes the number of strings in S.

1 Introduction

Pattern discovery is an intensively studied sub-area of Discovery Science. A large amount of effort was paid to devising efficient algorithms to extract interesting, useful, and even surprising substring patterns from massive string datasets such as biological sequences [1, 2]. Then this research has been extended to more complicated but very expressive pattern classes such as subsequence patterns [3, 4], episode patterns [5, 6], VLDC patterns [7] and their variations [8].

Another interesting and challenging direction of this research is discovery of *optimal pattern pairs*, whereas the above-mentioned algorithms are only for finding optimal *single* patterns. Very recently, in [9] we developed an efficient $O(N^2)$ time algorithm for finding optimal pairs of substring patterns combined with any Boolean functions such as $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NOT), where N denotes the total string length in the input dataset. For instance, the algorithm allows to find pattern pairs in form $p \wedge q$, which can be interpreted as two sequences with *cooperative* functions. Some developments have been made for finding cooperative pattern pairs with a certain distance α, denoted, for now, by $p \wedge_\alpha q$, in terms of *structured motifs* [10, 11] and *proximity patterns* [12, 13].

This paper extends special cases in the work of [9] by producing a generic algorithm to find the optimal pairs of *cooperative* patterns in form $p \wedge_\alpha q$, and *competing* patterns in form $p \wedge_\alpha \neg q$. We develop a very efficient algorithm that runs in $O(N^2)$ time using only $O(N)$ space for *any* given threshold parameter $\alpha \geq 0$, hence not increasing the asymptotic complexity of [9] which does not deal with any bounded distance between two patterns. The pattern pairs discovered by our algorithm are optimal in that they are guaranteed to be the highest scoring pair of patterns with respect to a given scoring function. Our algorithm can be adjusted to handle several common problem formulations of pattern discovery, for example, pattern discovery from positive and negative sequence sets [1, 14, 8, 15], as well as the discovery of patterns that *correlate* with a given numeric attribute (e.g. gene expression level) assigned to the sequences [16–18, 2, 19].

The efficiency of our algorithm comes from the uses of helpful data structures, *suffix trees* [20] and *sparse suffix trees* [21, 22]. We also present an efficient implementation that uses *suffix arrays* and *lcp arrays* [23] to simulate bottom-up traversals on suffix trees and sparse suffix trees.

Comparison to related works. Marsan and Sagot [10] gave an algorithm which is limited to finding pattern pairs in form $p \wedge_\alpha q$. Although the time complexity is claimed to be $O(Nm)$, where m denotes the number of strings in the dataset, a significant difference is that the length of each pattern is fixed and is regarded as a constant, whereas our algorithm does not impose such a restriction. Arimura et al. [12, 13] presented an algorithm to find the optimal (α, d)-proximity pattern that can be interpreted into a sequence $p_1 \wedge_\alpha \cdots \wedge_\alpha p_d$ of d patterns with bounded distance α. Although the expected running time of their algorithm is $O(\alpha^{d-1}N(\log N)^d)$, the worst case running time is still $O(\alpha^d N^{d+1} \log N)$. Since in our case $d = 2$, it turns out to be $O(\alpha^2 N^3 \log N)$ which is rather bad compared to our $O(N^2)$ time complexity. In addition, it is quite easy to extend our algorithm to finding (α, d)-proximity pattern in $O(N^d)$ time using only $O(N)$ space. Since the algorithm of [13] consumes $O(dN)$ space, our algorithm improves both time and space for extracting proximity patterns. On the other hand, an algorithm to discover pattern pairs in form $p \wedge_\alpha \neg q$ was given recently in [24], in terms of *missing patterns.* As the algorithm performs in $O(n^2)$ time using $O(n)$ space for a single input string of length n, our algorithm generalizes it for a set of strings with the same asymptotic complexity.

None of the above previous algorithms considers the fact that if one pattern of pair $p \wedge_\alpha q$ is very long, its ending position may be *outside* of the region specified by α, or for pattern pair in form $p \wedge_\alpha \neg q$, it may *invade* the α-region with respect to some other occurrence of p. Since such pattern pairs seem to be of less interest for some applications, we design a version of our algorithm which permits us to find pattern pair $p \wedge_\alpha q$ such that both p and q occur *strictly within* the α-region of each occurrence of p, and pattern pair $p \wedge_\alpha \neg q$ such that

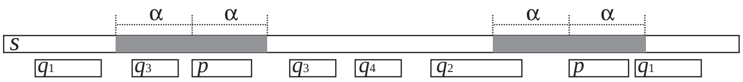


Fig. 1. From pattern p in string s, q_1 begins within α-distance, q_2 begins outside α-distance, q_3 occurs within α-distance, and q_4 occurs outside α-distance.

q is *strictly outside* the α-region of any occurrences of p. This version of our algorithm runs in $O(m^2N^2)$ time using $O(m^2N)$ space.

2 Preliminaries

2.1 Notation

Let Σ be a finite alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq n$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i:j]$ for $1 \leq i \leq j \leq n$. For any set S, let $|S|$ denote the cardinality of S. Let $[i, j]$ denote the set of consecutive integers $i, i+1, \ldots, j$.

For strings p and s, let $Beg(p, s)$ denote the set of all beginning positions of p in s, e.g.,

$$Beg(p, s) = \{i \mid s[i:i+|p|-1] = p\}.$$

Also, let $Cov(p, s)$ denote the set of all positions in s covered by p, e.g.,

$$Cov(p, s) = \cup_{j \in Beg(p,s)}[j, j+|p|-1].$$

For strings p, s and threshold value $\alpha \geq 0$, let us define set $D(p, s, \alpha)$ and of integers by

$$D(p, s, \alpha) = (\cup_{j \in Beg(p,s)}[j-\alpha, j+\alpha]) \cap [1, |s|].$$

That is, $D(p, s, \alpha)$ consists of the regions inside s that are closer than or equal to α positions from the beginning position of some occurrence of p. The shaded regions illustrated in Fig. 1 are the regions in $D(p, s, \alpha)$.

If $|Beg(q, s) \cap D(p, s, \alpha)| \geq 1$, that is, if one or more occurrences of q in s are within α positions from at least one occurrence of p in s, then q is said to *begin within α-distance* from p, and otherwise *begin outside α-distance*. Similarly, if $|Cov(q, s) \cap D(p, s, \alpha)| \geq 1$, that is, if there is at least one overlap between an occurrence of p and a region in $D(p, s, \alpha)$, then q is said to *occur within α-distance* from p, and otherwise *occur outside α-distance*. See also Fig. 1.

Let $\psi(p, s)$ be a Boolean matching function that has the value `true` if p is a substring of s, and `false` otherwise. We define $\langle p, F, q\rangle$ as a *Boolean pattern* *pair* (or simply *pattern pair*) which consists of two patterns p, q and a Boolean function $F : \{\texttt{true}, \texttt{false}\} \times \{\texttt{true}, \texttt{false}\} \to \{\texttt{true}, \texttt{false}\}$. We say that a single pattern or pattern pair π *matches* string s if and only if $\psi(\pi, s) = \texttt{true}$.

For α-distance, we define four Boolean matching functions F_1, F_2, F_3, and F_4 as follows: F_1 is such that $\psi(\langle p, F_1, q\rangle, s) = \texttt{true}$ if $\psi(p, s) = \texttt{true}$ and q begins within α-distance from p, and $\psi(\langle p, F_1, q\rangle, s) = \texttt{false}$ otherwise. F_2 is such that $\psi(\langle p, F_2, q\rangle, s) = \texttt{true}$ if $\psi(p, s) = \texttt{true}$ and q begins outside α-distance from p, and $\psi(\langle p, F_2, q\rangle, s) = \texttt{false}$ otherwise. F_3 is such that $\psi(\langle p, F_3, q\rangle, s) = \texttt{true}$ if $\psi(p, s) = \texttt{true}$ and q occurs within α-distance from p, and $\psi(\langle p, F_3, q\rangle, s) = \texttt{false}$ otherwise. F_4 is such that $\psi(\langle p, F_4, q\rangle, s) = \texttt{true}$ if $\psi(p, s) = \texttt{true}$ and q occurs outside α-distance from p, and $\psi(\langle p, F_4, q\rangle, s) = \texttt{false}$ otherwise. In the sequel, let us write as:

$$\langle p, F_1, q\rangle = p \wedge_{b(\alpha)} q, \quad \langle p, F_2, q\rangle = p \wedge_{b(\alpha)} \neg q,$$
$$\langle p, F_3, q\rangle = p \wedge_{c(\alpha)} q, \quad \langle p, F_4, q\rangle = p \wedge_{c(\alpha)} \neg q.$$

Note that, by setting $\alpha \geq |s|$, we always have $D(p, s, \alpha) = [1, |s|]$. Therefore, the above pattern pairs are generalizations of pattern pairs such as $(p \wedge q)$ and $(p \wedge \neg q)$ that are considered in our previous work [9].

Given a set $S = \{s_1, \ldots, s_m\}$ of strings, let $M(\pi, S)$ denote the subset of strings in S that π matches, that is,

$$M(\pi, S) = \{s_i \in S \mid \psi(\pi, s_i) = \texttt{true}\}.$$

We suppose that each $s_i \in S$ is associated with a numeric attribute value r_i. For a single pattern or pattern pair π, let $\sum_{M(\pi,S)} r_i$ denote the sum of r_i over all s_i in S such that $\psi(\pi, s_i) = \texttt{true}$, that is,

$$\sum_{M(\pi,S)} r_i = \sum_{s_i \in S} (r_i \mid \psi(\pi, s_i) = \texttt{true}).$$

As S is usually fixed in our case, we shall omit S where possible and let $M(\pi)$ and $\sum_{M(\pi)} r_i$ be a shorthand for $M(\pi, S)$ and $\sum_{M(\pi,S)} r_i$, respectively.

2.2 Problem Definition

In general, the problem of finding a good pattern from a given set S of strings refers to finding a pattern π that maximizes some suitable scoring function *score* with respect to the strings in S. We concentrate on *score* that takes parameters of type $|M(\pi)|$ and $\sum_{M(\pi)} r_i$, and assume that the score value computation itself can be done in constant time if the required parameter values are known. We formulate the pattern pair discovery problem as follows:

Problem 1. Given a set $S = \{s_1, \ldots, s_m\}$ of m strings, where each string s_i is assigned a numeric attribute value r_i, a threshold parameter $\alpha \geq 0$, a scoring function $score : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$, and a Boolean Function $F \in \{F_1, \ldots, F_4\}$, find the Boolean pattern pair $\pi \in \{\langle p, F, q\rangle \mid p, q \in \Sigma^*\}$ that maximizes $score(|M(\pi)|, \sum_{M(\pi)} r_i)$.

The specific choice of the scoring function depends highly on the particular application, for example, as follows: The positive/negative sequence set discrimination problem [1, 14, 8, 15] is such that, given two disjoint sets of sequences S_1 and S_2, where sequences in S_1 (the positive set) are known to have some biological function, while the sequences in S_2 (the negative set) are known not to, find pattern pairs which match more sequences in S_1, and less in S_2. Common scoring functions that are used in this situation include the entropy information gain, the Gini index, and the chi-square statistic, which all are essentially functions of $|M(\pi, S_1)|$, $|M(\pi, S_2)|$, $|S_1|$ and $|S_2|$. The correlated patterns problem [16–18, 2, 19] is such that we are given a set S of sequences, with a numeric attribute value r_i associated with each sequence $s_i \in S$, and the task is to find pattern pairs whose occurrences in the sequences correlate with their numeric attributes. In this framework, scoring functions such as mean squared error, which is a function of $|M(\pi)|$ and $\sum_{M(\pi)} r_i$, can be used.

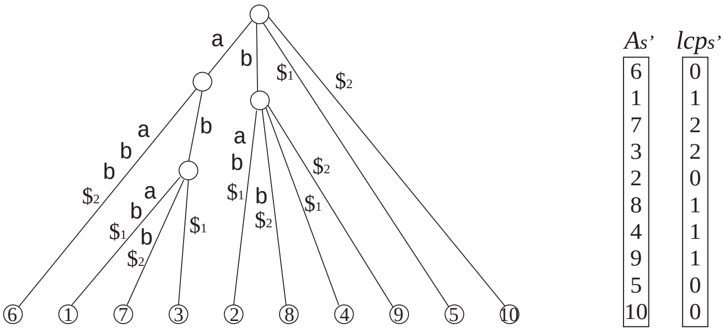
2.3 Data Structures

We intensively use the following data structures in our algorithms. The efficiency of our algorithms, in both time and space consumptions, comes from the use of these data structures.

The *suffix tree* [20] for a string s is a rooted tree whose edges are labeled with substrings of s, satisfying the following characteristics. For any node v in the suffix tree, let $l(v)$ denote the string spelled out by concatenating the edge labels on the path from the root to v. For each leaf v, $l(v)$ is a distinct suffix of s, and for each suffix in s, there exists such a leaf v. Each leaf v is labeled with $pos(v) = i$ such that $l(v) = s[i : |s|]$. Also, each internal node has at least two children, and the first character of the labels on the edges to its children are distinct. It is well known that suffix trees can be represented in linear space and constructed in linear time with respect to the length of the string [20].

The *generalized suffix tree* (*GST*) for a set $S = \{s_1, \ldots, s_m\}$ of strings is basically the suffix tree for the string $s_1\$_1 \cdots s_m\$_m$, where each $\$_i$ $(1 \leq i \leq m)$ is a distinct character which does not appear in any of the strings in the set. However, all paths are ended at the first appearance of any $\$_i$, and each leaf is labeled with id_i that specifies the string in S the leaf is associated with. For set $\{\mathtt{abab}, \mathtt{aabb}\}$, the corresponding GST is shown in Fig. 2. GSTs are also linear time constructible and can be stored in linear space with respect to the total length of the strings in the input set [20].

The *sparse suffix tree* (*SST*) [21, 22] for a string s is a rooted tree which represents a subset of the suffixes of s, i.e., a suffix tree of s from which the *dead leaves* (and corresponding internal nodes) are removed, where the dead leaves stand for the leaves that correspond to the suffixes *not* in the subset. On the other hand, the leaves existing in the SST are called the *living leaves*. An SST can be constructed in linear time by once building the full suffix tree and pruning dead leaves from it. (Direct construction of SSTs was also considered in [21, 22] but the pruning approach is sufficient for our purposes.)



As'	*lcps'*
6	0
1	1
7	2
3	2
2	0
8	1
4	1
9	1
5	0
10	0

Fig. 2. The tree is the GST for set $\{\texttt{abab}, \texttt{aabb}\}$. Each leaf y is labeled by $pos(y)$ with respect to the concatenated string $s' = \texttt{abab}\$_1\texttt{aabb}\$_2$. The left and right arrays are the suffix and lcp arrays of string s', respectively.

The *suffix array* [23] A_s for a string s of length n, is a permutation of the integers $1, \ldots, n$ representing the lexicographic ordering of the suffixes of s. The value $A_s[i] = j$ in the array indicates that $s[j : n]$ is the ith suffix in the lexicographic ordering. The *lcp array* for a string s, denoted lcp_s, is an array of integers representing the longest common prefix lengths of adjacent suffixes in the suffix array, that is,

$$lcp_s[i] = \max\{k \mid s[A_s[i-1] : A_s[i-1] + k - 1] = s[A_s[i] : A_s[i] + k - 1]\}.$$

Note that each position i in A_s corresponds to a leaf in the suffix tree of s, and $lcp_s[i]$ denotes the length of the path from the root to leaves of positions $i - 1$ and i for which the labels are equivalent. Fig. 2 shows the suffix array $A_{s'}$ and the lcp array $lcp_{s'}$ of string $s' = \texttt{abab}\$_1\texttt{aabb}\$_2$.

Recently, three methods for constructing the suffix array directly from a string in linear time have been developed [25–27]. The *lcp* array can be constructed from the suffix array also in linear time [28]. It has been shown that several (and potentially many more) algorithms which utilize the suffix tree can be implemented very efficiently using the suffix array together with its *lcp* array [28, 29]. This paper presents yet another good example for efficient implementation of an algorithm based conceptually on full and sparse suffix trees, but uses the suffix and *lcp* arrays.

The *lowest common ancestor* $lca(x, y)$ of any two nodes x and y in a tree is the deepest node which is common to the paths from the root to both of the nodes. The tree can be pre-processed in linear time to answer the lowest common ancestor (*lca-query*) for any given pair of nodes in constant time [30]. In terms of the suffix array, the *lca*-query is almost equivalent to a *range minimum query* (*rm-query*) on the *lcp* array, which, given a pair of positions i and j, $rmq(i, j)$ returns the position of the minimum element in the sub-array $lcp[i : j]$. The *lcp* array can also be pre-processed in linear time to answer the *rm*-query in constant time [30, 31].

In the sequel, we will show a clever implementation of our algorithms which simulates bottom-up traversals on SGSTs using suffix and *lcp* arrays. Our algorithms need no explicit prunings of leaves from the SGSTs which means that the suffix and *lcp* arrays do not have to be explicitly modified or reconstructed. Therefore our algorithms are both time and space efficient in practice.

3 Algorithms

In this section we present our efficient algorithms to solve Problem 1 for Boolean functions $F_1, \ldots, F_4$. Let $N = |s_1\$_1 \cdots s_m\$_m|$, where $S = \{s_1, \ldots, s_m\}$. The algorithms calculate scores for all possible pattern pairs, and output a pattern pair of the highest score.

3.1 Algorithm for Single Patterns

For a single pattern, we need to consider only patterns of form $l(v)$, where v is a node in the GST for S: If a pattern has a corresponding path that ends in the middle of an edge of the GST, it will match the same set of strings as the pattern corresponding to the end node of that same edge, and hence the score would be the same. Below, we recall the $O(N)$ time algorithm of [9] that computes $\sum_{M(l(v))} r_i$ for all single pattern candidates of form $l(v)$. The algorithm is derived from the technique for solving the *color set size problem* [32]. Note that we do not need to consider separately how to compute $|M(l(v))|$: If we give each attribute r_i the value 1, then $\sum_{M(l(v))} r_i = |M(l(v))|$.

For any node v in the GST of S, let $LF(v)$ be the set of all leaves in the subtree rooted by v, $c_i(v)$ be the number of leaves in $LF(v)$ with the label id_i, and

$$\sum_{LF(v)} r_i = \sum (c_i(v) r_i \mid \psi(l(v), s_i) = \texttt{true}).$$

For any such v, $\psi(l(v), s_i) = \texttt{true}$ for at least one string s_i. Thus, we have

$$\begin{aligned}\sum_{M(l(v))} r_i &= \sum (r_i \mid \psi(l(v), s_i) = \texttt{true}) \\ &= \sum_{LF(v)} r_i - \sum ((c_i(v) - 1) r_i \mid \psi(l(v), s_i) = \texttt{true}).\end{aligned}$$

We define the above subtracted sum to be a *correction factor*, denoted by $corr(l(v), S) = \sum((c_i(v) - 1) r_i \mid \psi(l(v), s_i) = \texttt{true})$. The following recurrence

$$\sum_{LF(v)} r_i = \sum (\sum_{LF(v')} r_i \mid v' \text{is a child of } v)$$

allows us to compute the values $\sum_{LF(v)} r_i$ for all v during a linear time bottom-up traversal of the GST. Now we need to remove the redundancies, represented by $corr(l(v), S)$, from $\sum_{LF(v)} r_i$.

Let $I(id_i)$ be the list of all leaves with the label id_i in the order they appear in a depth-first traversal of the tree. Clearly the lists $I(id_i)$ can be constructed in linear time for all labels id_i. We note four properties in the following proposition:

Proposition 1. *For the GST of S, the following properties hold:*

(1) The leaves in $LF(v)$ with the label id_i form a continuous interval of length $c_i(v)$ in the list $I(id_i)$.
(2) If $c_i(v) > 0$, a length-$c_i(v)$ interval in $I(id_i)$ contains $c_i(v) - 1$ adjacent (overlapping) leaf pairs.
(3) If $x, y \in LF(v)$, the node $lca(x, y)$ belongs to the subtree rooted by v.
(4) For any $s_i \in S$, $\psi(l(v), s_i) = \mathtt{true}$ if and only if there is a leaf $x \in LF(v)$ with the label id_i.

We initialize each node v to have a correction value 0. Then, for each adjacent leaf pair x, y in the list $I(id_i)$, we add the value r_i into the correction value of the node $lca(x, y)$. It follows from properties (1) - (3) of Proposition 1, that now the sum of the correction values in the nodes of the subtree rooted by v equals $(c_i(v) - 1)r_i$. After repeating the process for each of the lists $I(id_i)$, property (4) tells that the preceding total sum of the correction values in the subtree rooted by v becomes

$$\sum((c_i(v) - 1)r_i \mid \psi(l(v), s_i) = \mathtt{true}) = corr(l(v), S).$$

Now a single linear time bottom-up traversal of the tree enables us to cumulate the correction values $corr(l(v), S)$ from the subtrees into each node v, and at the same time we may record the final values $\sum_{M(l(v))} r_i$.

The preceding process involves a constant number of linear time traversals of the tree, as well as a linear number of constant time lca-queries (after a linear time preprocessing). Hence the values $\sum_{M(l(v))} r_i$ are computed in linear time.

3.2 Algorithms for Pattern Pairs with α-Distance

The algorithm described in Section 3.1 permits us to compute $\sum_{M(l(v))} r_i$ in $O(N)$ time. In this section, we concentrate on how to compute the values $\sum_{M(\pi)} r_i$ for various types of pattern pair π with bounded distance α. In so doing, we go over the $O(N)$ choices for the first pattern, and for each fixed pattern $l(v_1)$, we compute $\sum_{M(\pi)} r_i$ where π consists of $l(v_1)$ and second pattern candidate $l(v)$. The result of this section is summarized in the following theorem:

Theorem 1. *Problem 1 is solvable in $O(N^2)$ time using $O(N)$ space for $F \in \{F_1, F_2\}$, and $O(m^2N^2)$ time and $O(m^2N)$ space for $F \in \{F_3, F_4\}$.*

Below, we will give the details of our algorithm for each type of pattern pair.

For Pattern Pairs in form $p \wedge_{b(\alpha)} q$. Let $D(l(v_1), S, \alpha) = \cup_{s_i \in S} D(l(v_1), s_i, \alpha)$. It is not difficult to see that $D(l(v_1), S, \alpha)$ is $O(N)$ time constructible (see Theorem 9 of [24]). We then mark as 'living' every leaf of the GST whose position belongs to $D(l(v_1), S, \alpha)$. This aims at conceptually constructing the *sparse generalized suffix tree* (*SGST*) for the subset of suffixes corresponding to $D(l(v_1), S, \alpha)$, and this can be done in $O(N)$ time. Then, we additionally label each string $s_i \in S$, and the corresponding leaves in the SGST, with the Boolean value $\psi(l(v_1), s_i)$. This can be done in $O(N)$ time. Now the trick is to cumulate the sums and correction factors *only* for the nodes existing in the SGST, and *separately* for different values of the additional label. The end result is that we will obtain the values $\sum_{M(\pi)} r_i = \sum(r_i \mid \psi(\pi, s_i) = \texttt{true})$ in linear time, where π denotes the pattern pair $l(v_1) \wedge_{b(\alpha)} l(v)$ for all nodes v in the conceptual SGST. Since there are only $O(N)$ candidates for $l(v_1)$, the total time complexity is $O(N^2)$. The space requirement is $O(N)$, since we repeatedly use the same GST for S and the additional information storage is also linear in N at each phase of the algorithm.

For Pattern Pairs in form $p \wedge_{b(\alpha)} \neg q$. Let us denote any pattern pair in form $p \wedge_{b(\alpha)} \neg q$ by $\overline{\pi}$. For the pattern pairs of this form, it stands that

$$\begin{aligned}\sum_{M(\overline{\pi})} r_i &= \sum(r_i \mid \psi(\overline{\pi}, s_i) = \texttt{true}) \\ &= \sum(r_i \mid \psi(p, s) = \texttt{true}) - \sum_{M(\pi)} r_i,\end{aligned}$$

where π denotes $p \wedge_{b(\alpha)} q$. For each fixed first pattern $l(v_1)$, $\sum_{M(\pi)} r_i$ and $\sum(r_i \mid \psi(l(v_1), s) = \texttt{true})$ can be computed in $O(N)$ time, and hence $\sum_{M(\overline{\pi})} r_i$ can be computed in $O(N)$ time as well. Since we have $O(N)$ choices for the first pattern, the total time complexity is $O(N^2)$, and the space complexity is $O(N)$.

For Pattern Pairs in form $p \wedge_{c(\alpha)} q$. There are two following types of patterns that have to be considered as candidates for the second pattern q:

(1) pattern q which has an occurrence beginning at some position in $D(l(v_1), S, \alpha)$.
(2) pattern q which has an occurrence beginning at some position in $[1, N] - D(l(v_1), S, \alpha)$, but overlapping with some positions in $D(l(v_1), S, \alpha)$.

Now, we separately treat these two kinds of patterns. In so doing, we consider two SGSTs, one consists only of the leaves x such that $pos(x) \in D(l(v_1), S, \alpha)$, and the other consists only of the leaves y such that $pos(y) \in [1, N] - D(l(v_1), S, \alpha)$. Denote these by $SGST_1$ and $SGST_2$, respectively.

For integer i with $1 \leq i \leq m$, let $N_i = |s_1\$_1 \cdots s_i\$_i|$. For any leaf y of $SGST_2$ with id_i, we define $len(y)$ as follows:

$$len(y) = \begin{cases} N_{i-1} + h - \alpha - pos(y) & \text{if } pos(y) < N_{i-1} + k - \alpha, \\ N_i - pos(y) & \text{otherwise,} \end{cases}$$

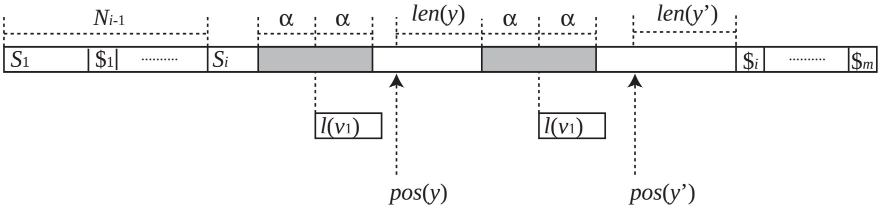


Fig. 3. For two leaves y, y' of $SGST_2$, $len(y)$ and $len(y')$ with respect to string $s_i \in S$ and the first pattern $l(v_1)$.

where h denotes the minimum element of $Beg(l(v_1), s_i)$ satisfying $N_{i-1} + h > pos(y)$, and $k = \max(Beg(l(v_1), s_i))$. Fig. 3 illustrates $len(y)$ with respect to s_i and $l(v_1)$. Computing $len(y)$ can be done in constant time for each leaf y, after a linear time preprocessing of scanning $D(l(v_1), S, \alpha)$ from right to left. For any node v of $SGST_2$, let $SLF_i(v)$ denote the list of all leaves that are in the subtree of $SGST_2$ rooted at v and are labeled with id_i. Then, we compute $len_i(v) = \min_{y \in SLF_i(v)}(len(y))$ for every internal node v in $SGST_2$. For all nodes v and $i = 1, \ldots, m$, $len_i(v)$ can be computed in $O(mN)$ time using $O(mN)$ space. Notice that only nodes v such that for some i, $len_i(v) < |l(v)|$ in $SGST_2$, can give candidates for the second pattern, since the occurrences of the substrings represented by the other nodes are completely outside $D(l(v_1), S, \alpha)$.

Consider an internal node v' and its parent node v in $SGST_2$. Assume the length of the edge between v and v' is more than one. Suppose the subtree of $SGST_2$ rooted at v' contains a leaf with id_i. Now consider an 'implicit' node t that is on the edge from v to v'. Then, even if $len_i(v') < |l(v')|$, it might stand that $len_i(t) \geq |l(t)|$ since $len_i(t) = len_i(v')$ and $|l(t)| < |l(v')|$ always hold. Note that in this case we have to take into account such implicit node t for the second pattern candidate, independently from v'. On the other hand, if we locate t so that $len_i(t) = |l(t)|$, we will have $len_i(t') > |l(t')|$ for any implicit node t' upper than t in the edge, and will have $len_i(t'') < |l(t'')|$ for any implicit node t'' lower than t in the edge. Thus this single choice of t suffices. Since the subtree can contain at most m distinct id's, each edge can have at most m such implicit nodes on it, and because $SGST_2$ has $O(N)$ edges, the total number of the implicit nodes we have to treat becomes $O(mN)$. What still remains is how to locate these implicit nodes in $O(mN)$ time. This is actually feasible by using the so-called *suffix links* and the *canonization* technique introduced in [33]. For each of these implicit nodes we have to maintain m number of information as mentioned in the above paragraph, and hence in total we are here required $O(m^2N)$ time and space.

Cumulating the sums and correction factors is done in two rounds: For simplicity, let us denote pattern pair $l(v_1) \wedge_{c(\alpha)} l(v)$ by π. First we traverse $SGST_1$ and for each encountered node v, calculate the value $\sum_{M(\pi),1} r_i$ and store it in v. Then, we traverse $SGST_2$ and for each encountered node w calculate the value $\sum_{M(\pi),2} r_i$ if necessary, while checking whether $len_i(w) < |l(w)|$ or not, and store it in w but *separately* from $\sum_{M(\pi),1} r_i$. Then, the final value $\sum_{M(\pi)} r_i = \sum_{M(\pi),1} r_i + \sum_{M(\pi),2} r_i$. Since we go over $O(N)$ choices for the first pattern, the resulting algorithm requires $O(m^2N^2)$ time and $O(m^2N)$ space.

For Pattern Pairs in form $p \wedge_{c(\alpha)} \neg q$. As similar arguments to pattern pairs in form $p \wedge_{b(\alpha)} \neg q$ hold, it can also be done $O(m^2N^2)$ time and $O(m^2N)$ space.

3.3 Implementation

We show an efficient implementation of the algorithms for pattern pairs in form $p \wedge_{b(\alpha)} q$ and $p \wedge_{b(\alpha)} \neg q$, both run in $O(N^2)$ time using $O(N)$ space as previously explained. We further modify the implementation of the algorithm in [9], which is an extension of the Substring Statistics algorithm in [28], that uses the suffix and lcp arrays to simulate the bottom up traversals of the generalized suffix tree. We note that the simulation does not increase the asymptotic complexity of the algorithm, but rather it is expected to increase the efficiency of the traversal, as confirmed in previous works [34, 28, 29].

Fig. 4 shows a pseudo-code for solving the Color Set Size problem with pruned edges using a suffix array, which is used in the algorithm of the previous subsection. There are two differences from [9]: (1) the correction factors are set to $lca(i, j)$ of consecutive 'living' leaves i, j $(i < j)$ corresponding to the same string. Since the simulation via suffix arrays does not explicitly construct all internal nodes of the suffix tree, we use another array CF to hold the correction factors. The correction factor for these leaves i, j is summed into $CF[rmq(i + 1, j)]$. (2) 'Dead' leaves are ignored in the bottom-up traversal. This can be done in several ways, the simplest which assigns 0 to the weights of each 'dead' leaf. However, although the worst-case time complexity remains the same, such an algorithm would report *redundant* internal nodes that only have at most a single 'living' leaf in its subtree. A more complete pruning can be achieved, as shown in the pseudo-code, in the following way: for any consecutive 'living' leaves i, j $(i < j)$ in the suffix array, we add up all $CF[k]$ for $i < k \leq j$ which is then added to the node corresponding to $lca(i, j)$. The correctness of this algorithm can be confirmed by the following argument: For a given correction factor $CF[k]$ $(i < k \leq j)$, since the query result came from consecutive 'living' leaves of the same string, they must have been from leaves i', j', such that $i' \leq i$ and $j \leq j'$. This means that $lcp[k] = lcp[rmq(i' + 1, j')] \leq lcp[rmq(i + 1, j)] \leq lcp[k]$. Therefore $lcp[rmq(i' + 1, j')] = lcp[rmq(i + 1, j)] = lcp[k]$, and $lca(i', j') = lca(i, j)$, which shows that $CF[k]$ should be summed into the node with depth $lcp[rmq(i+1, j)]$.

4 Computational Experiments

We tested the effectiveness of our algorithm using the two sets of predicted 3'UTR processing site sequences provided in [35], which are constructed based on the microarray experiments in [36] that measure the degradation rate of yeast

```
Let $Stack = \{(0,-1,0)\}$ be the stack;
$cc := 0$; $iplus1 = 0$;
foreach $j = 1,\ldots,n+1$ do:
    $cc := cc + CF[j]$;
    $(L,H,C) = top(Stack)$;
    if $j$ is a living leaf
        $(L_j,H_j,C_j) := (j, lcp[rmq(iplus1,j)], 0)$;
        while $(H > H_j)$ do:
            $(L,H,C) := pop(Stack)$;
            report $(L,H,C)$; /* $s[A[L]:A[L]+H-1]$ has count $C$ */
            $C_j := C + C_j$;
            $(L,H,C) := top(Stack)$;
        if $(H < H_j)$ then
            $push((L_j,H_j,C_j+cc),Stack)$;
        else /* $H = H_j$ */
            $(L,H) := pop(Stack)$;
            $push((L,H,C+C_j+cc),Stack)$;
        $push((j,N-A[j]+1,1),Stack)$;
        $cc := 0$; $iplus1 = j+1$;
```

Fig. 4. Core of the algorithm for solving the Color Set Size problem with pruned edges using a suffix array. We assume each leaf (position in array) has been labeled 'dead' or 'living', and the correction factors for 'living' leaves are stored in the array CF. A node in the suffix tree together with $|M(l(v))|$ is represented by a three-tuple (L,H,C).

mRNA. One set S_f consists of 393 sequences which have a fast degradation rate ($t_{1/2} < 10$ minutes), while the other set S_s consists of 379 predicted 3'UTR processing site sequences which have a slow degradation rate ($t_{1/2} > 50$ minutes). Each sequence is 100 nt long, and the total length of the sequences is $77,200$ nt. For the scoring function, we used the chi-squared statistic, calculated by $(|S_f|+|S_s|)(\mathtt{tp}*\mathtt{tn}-\mathtt{fp}*\mathtt{fn})^2/(\mathtt{tp}+\mathtt{fn})(\mathtt{tp}+\mathtt{fp})(\mathtt{tn}+\mathtt{fp})(\mathtt{tn}+\mathtt{fn})$ where $\mathtt{tp} = |M(\pi, S_f)|$, $\mathtt{fp} = |S_f| - \mathtt{tp}$, $\mathtt{tn} = |S_s| - \mathtt{fn}$, and $\mathtt{fn} = |M(\pi, S_s)|$.

For $b(\alpha)$ and $1 \leq \alpha \leq 30$, the best score was 48.3 ($p < 10^{-11}$) for the pair $\mathtt{AUA} \wedge_{b(10)} \neg\mathtt{UGUA}$, which matched 159/393 and 248/379 fast and slowly degrading sequences respectively, meaning that this pattern pair is more frequent in the slowly degrading sequences. This result is similar to the best pattern of the form $p \wedge \neg q$ obtained in [9], which was $\mathtt{UGUA} \wedge \neg\mathtt{AUCC}$ with score 33.9 (240/393 and 152/379 ($p < 10^{-8}$)), appearing more frequently in the fast degrading sequences. In fact, UGUA is known as a binding site of the PUF protein family which plays important roles in mRNA regulation [37]. Of the 268/393 and 190/379 sequences that contain both AUA and UGUA, their distances were farther than $\alpha = 10$ in only 37/268 of fast degrading sequences, while the number was 67/190 for slowly degrading sequences. This could mean that AUA is another sequence element whose distance from UGUA influences how efficiently UGUA functions.

We did not observe any notable difference in computational time compared to the algorithm in [9], that the marking of each leaf as 'dead' or 'living' could

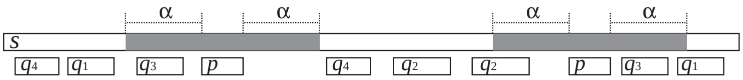


Fig. 5. From pattern p in string s, q_1 begins within α-gap, q_2 begins outside α-gap, q_3 occurs within α-gap, and q_4 occurs outside α-gap.

cause. For a given value of α, computation took around 2140 seconds for the above dataset on a PC with Xeon 3 GHz Processor running Linux.

5 Concluding Remarks

In this paper we studied the problem of finding the optimal pair of substring patterns p, q with bounded distance α, from a given set S of strings, which is an extension of the problem studied in [9]. We developed an efficient algorithm that finds the best pair of *cooperative* patterns in form $p \wedge_{b(\alpha)} q$, and *competing* patterns in form $p \wedge_{b(\alpha)} \neg q$, which performs in $O(N^2)$ time using $O(N)$ space, where N is the total length of the strings in S. For the more difficult versions of the problem, referred to finding $p \wedge_{c(\alpha)} q$ and $p \wedge_{c(\alpha)} \neg q$, we gave an algorithm running in $O(m^2N^2)$ time and $O(m^2N)$ space, where $m = |S|$. An interesting open problem is if the m-factors can be removed from the complexities.

Any pairs output from our algorithms are guaranteed to be optimal in the pattern classes, in the sense that they give the highest score due to the scoring function. Our algorithms are adapted to various applications such as the *positive/negative sequence set discrimination problem* for which the entropy information gain, the Gini index, and the chi-square statistic are commonly used as the scoring function, and the *correlated patterns problem* for which the mean squared error can be used.

It is possible to use our algorithms for bounded *gaps* α (see Fig. 5). Since the ending positions of the first patterns have to be considered in this case, we will naturally have $O(N^2)$ choices for the first pattern, and this fact suggests the use of suffix tries rather than suffix trees for the first patterns. Still, there are only $O(N)$ choices for the second pattern, and thus for the pattern pairs with respect to $b(\alpha)$, we can modify our algorithm to solve the problems in $O(N^3)$ time using $O(N^2)$ space. We remark that for $c(\alpha)$, a modified algorithm finds the optimal pattern pair in $O(m^2N^3)$ time using $O(N^2 + m^2N)$ space.

References

1. Shimozono, S., Shinohara, A., Shinohara, T., Miyano, S., Kuhara, S., Arikawa, S.: Knowledge acquisition from amino acid sequences by machine learning system BONSAI. Transactions of Information Processing Society of Japan **35** (1994) 2009–2018
2. Bannai, H., Inenaga, S., Shinohara, A., Takeda, M., Miyano, S.: Efficiently finding regulatory elements using correlation with gene expression. Journal of Bioinformatics and Computational Biology (2004) in press.

3. Baeza-Yates, R.A.: Searching subsequences (note). Theoretical Computer Science **78** (1991) 363–376
4. Hirao, M., Hoshino, H., Shinohara, A., Takeda, M., Arikawa, S.: A practical algorithm to find the best subsequence patterns. In: Proc. 3rd International Conference on Discovery Science (DS'00). Volume 1967 of LNAI., Springer-Verlag (2000) 141–154
5. Mannila, H., Toivonen, H., Verkamo, A.I.: Discovering frequent episode in sequences. In: Proc. 1st International Conference on Knowledge Discovery and Data Mining, AAAI Press (1995) 210–215
6. Hirao, M., Inenaga, S., Shinohara, A., Takeda, M., Arikawa, S.: A practical algorithm to find the best episode patterns. In: Proc. 4th International Conference on Discovery Science (DS'01). Volume 2226 of LNAI., Springer-Verlag (2001) 435–440
7. Inenaga, S., Bannai, H., Shinohara, A., Takeda, M., Arikawa, S.: Discovering best variable-length-don't-care patterns. In: Proc. 5th International Conference on Discovery Science (DS'02). Volume 2534 of LNCS., Springer-Verlag (2002) 86–97
8. Takeda, M., Inenaga, S., Bannai, H., Shinohara, A., Arikawa, S.: Discovering most classificatory patterns for very expressive pattern classes. In: Proc. 6th International Conference on Discovery Science (DS'03). Volume 2843 of LNCS., Springer-Verlag (2003) 486–493
9. Bannai, H., Hyyrö, H., Shinohara, A., Takeda, M., Nakai, K., Miyano, S.: Finding optimal pairs of patterns. In: Proc. 4th Workshop on Algorithms in Bioinformatics (WABI'04). (2004) to appear.
10. Marsan, L., Sagot, M.F.: Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification. J. Comput. Biol. **7** (2000) 345–360
11. Palopoli, L., Terracina, G.: Discovering frequent structured patterns from string databases: an application to biological sequences. In: 5th International Conference on Discovery Science (DS'02). Volume 2534 of LNCS., Springer-Verlag (2002) 34–46
12. Arimura, H., Arikawa, S., Shimozono, S.: Efficient discovery of optimal word-association patterns in large text databases. New Generation Computing **18** (2000) 49–60
13. Arimura, H., Asaka, H., Sakamoto, H., Arikawa, S.: Efficient discovery of proximity patterns with suffix arrays (extended abstract). In: Proc. the 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01). Volume 2089 of LNCS., Springer-Verlag (2001) 152–156
14. Shinohara, A., Takeda, M., Arikawa, S., Hirao, M., Hoshino, H., Inenaga, S.: Finding best patterns practically. In: Progress in Discovery Science. Volume 2281 of LNAI., Springer-Verlag (2002) 307–317
15. Shinozaki, D., Akutsu, T., Maruyama, O.: Finding optimal degenerate patterns in DNA sequences. Bioinformatics **19** (2003) 206ii–214ii
16. Bussemaker, H.J., Li, H., Siggia, E.D.: Regulatory element detection using correlation with expression. Nature Genetics **27** (2001) 167–171
17. Bannai, H., Inenaga, S., Shinohara, A., Takeda, M., Miyano, S.: A string pattern regression algorithm and its application to pattern discovery in long introns. Genome Informatics **13** (2002) 3–11
18. Conlon, E.M., Liu, X.S., Lieb, J.D., Liu, J.S.: Integrating regulatory motif discovery and genome-wide expression analysis. Proc. Natl. Acad. Sci. **100** (2003) 3339–3344

19. Zilberstein, C.B.Z., Eskin, E., Yakhini, Z.: Using expression data to discover RNA and DNA regulatory sequence motifs. In: The First Annual RECOMB Satellite Workshop on Regulatory Genomics. (2004)
20. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)
21. Andersson, A., Larsson, N.J., Swanson, K.: Suffix trees on words. Algorithmica **23** (1999) 246–260
22. Kärkkänen, J., Ukkonen, E.: Sparse suffix trees. In: Proceedings of the Second International Computing and Combinatorics Conference (COCOON'96). Volume 1090 of LNCS., Springer-Verlag (1996) 219–230
23. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM Journal on Computing **22** (1993) 935–948
24. Inenaga, S., Kivioja, T., Mäkinen, V.: Finding missing patterns. In: Proc. 4th Workshop on Algorithms in Bioinformatics (WABI'04). (2004) to appear.
25. Kim, D.K., Sim, J.S., Park, H., Park, K.: Linear-time construction of suffix arrays. In: 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03). Volume 2676 of LNCS., Springer-Verlag (2003) 186–199
26. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. In: 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03). Volume 2676 of LNCS., Springer-Verlag (2003) 200–210
27. Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: 30th International Colloquium on Automata, Languages and Programming (ICALP'03). Volume 2719 of LNCS., Springer-Verlag (2003) 943–955
28. Kasai, T., Arimura, H., Arikawa, S.: Efficient substring traversal with suffix arrays. Technical Report 185, Department of Informatics, Kyushu University (2001)
29. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: The enhanced suffix array and its applications to genome analysis. In: Second International Workshop on Algorithms in Bioinformatics (WABI'02). Volume 2452 of LNCS., Springer-Verlag (2002) 449–463
30. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Proc. 4th Latin American Symp. Theoretical Informatics (LATIN'00). Volume 1776 of LNCS., Springer-Verlag (2000) 88–94
31. Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T.: Nearest common ancestors: a survey and a new distributed algorithm. In: 14th annual ACM symposium on Parallel algorithms and architectures (SPAA'02). (2002) 258–264
32. Hui, L.: Color set size problem with applications to string matching. In: Proc. 3rd Annual Symposium on Combinatorial Pattern Matching (CPM'92). Volume 644 of LNCS., Springer-Verlag (1992) 230–243
33. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14** (1995) 249–260
34. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: 12th Annual Symposium on Combinatorial Pattern Matching (CPM'01). Volume 2089 of LNCS., Springer-Verlag (2001) 181–192
35. Graber, J.: Variations in yeast 3'-processing cis-elements correlate with transcript stability. Trends Genet. **19** (2003) 473–476 `http://harlequin.jax.org/yeast/turnover/`.
36. Wang, Y., Liu, C., Storey, J., Tibshirani, R., Herschlag, D., Brown, P.: Precision and functional specificity in mRNA decay. Proc. Natl. Acad. Sci. **99** (2002) 5860–5865
37. Wickens, M., Bernstein, D.S., Kimble, J., Parker, R.: A PUF family portrait: 3'UTR regulation as a way of life. Trends Genet. **18** (2002) 150–157