EFFICIENTLY FINDING REGULATORY ELEMENTS USING CORRELATION WITH GENE EXPRESSION

HIDEO BANNAI[1], SHUNSUKE INENAGA[2,3], AYUMI SHINOHARA[2,3],
MASAYUKI TAKEDA[2,3], SATORU MIYANO[1]

[1] *Human Genome Center, Institute of Medical Science, University of Tokyo, 4-6-1 Shirokanedai, Minato-ku, Tokyo 108-8639, Japan*
{*bannai,miyano*}*@ims.u-tokyo.ac.jp*

[2] *Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan*
[3] *PRESTO, Japan Science and Technology Agency (JST)*
{*s-ine,ayumi,takeda*}*@i.kyushu-u.ac.jp*

We present an efficient algorithm for detecting putative regulatory elements in the upstream DNA sequences of genes, using gene expression information obtained from microarray experiments. Based on a *generalized suffix tree*, our algorithm looks for motif patterns whose appearance in the upstream region is most correlated with the expression levels of the genes. We are able to find the optimal pattern, in time linear in the total length of the upstream sequences. We implement and apply our algorithm to publicly available microarray gene expression data, and show that our method is able to discover biologically significant motifs, including various motifs which have been reported previously using the same dataset. We further discuss applications for which the efficiency of the method is essential, as well as possible extensions to our algorithm.

Keywords: suffix tree; pattern discovery; gene expression and regulatory elements.

1. Introduction

Gene expression is regulated by *transcription factors*, which bind to specific sequences usually in the upstream of the coding region of a gene. The binding sites for a given transcription factor are usually short regions of up to 15 base pairs, and is fairly conserved across genes which are regulated by the same transcription factor. In order to fully understand the mechanism underlying gene regulation, it is critical to discover the motifs that correspond to transcription factor binding sites.

Microarray data provides measurements of gene expression under varying experimental conditions. One popular use of this data is to first cluster the genes according to their gene expression measurements in several experiments – for example, within a time series – and to subsequently find motifs which are common to the upstream region of genes within each cluster[1,2,3]. However, this clustering approach has inherent limitations: not all genes in the cluster possess a common motif; more importantly, any one gene may possess multiple motifs that correspond to *different* transcription factors, so clustering the genes into disjoint sets may not allow these motifs to be found.

To overcome such problems, a method which does not require a pre-clustering

step has been presented by Bussemaker *et al.*[4] Concerning a single microarray experiment, they assume a linear relation between the log expression ratio of a gene, and the number of times a motif appears in the upstream region of the gene. Their method exhaustively looks for motifs of a certain length which best fits the expression data, using a simple motif model which does not consider any mismatches or ambiguity (which we shall call the *substring pattern class*). In order to enumerate all possible patterns, a naïve solution for this problem would result in exponential time in the length of the pattern to look for. However, that study presented no rigorous algorithm for finding the optimal pattern. Also, since the length of the pattern to look for is predefined, optimality of the patterns is not assured. Furthermore, finding the optimal pattern with such a naïve enumeration is virtually impossible because it is limited only by the longest sequence in the set of sequences.

The objective of this paper is to present a new algorithm that overcomes these difficulties: an algorithm that runs in linear time *and* gives optimal results. The problem setting described above can be formulated as a case of the *string pattern regression problem* which we have defined in a previous paper[5], provided that we consider the occurrence of the motif as a binary value - the motif either occurs or it does not. The rationality behind this simplification is based on the following observations: 1) we do not know *a priori*, if the log expression ratio of the gene and the number of times that a motif appears in the upstream region are linearly related, and 2) a motif of biologically plausible length will only appear a few times, if at all, in each upstream sequence. Thus, a binary indicator is likely to be a good estimate of occurrence counts anyway. We show that our method gives results similar to those obtained previously[4], but is remarkably fast.

We previously gave an enumerative branch-and-bound algorithm[5] which solves the string pattern regression problem optimally for a general motif model including patterns with don't care characters, wild-cards, mismatches, etc. However, the problem is still difficult to solve in general, and the computational cost is still quite large when considering gene regulatory regions which can be $600 \sim 800$bp long. Our linear time algorithm takes a completely different approach, based on a linear time algorithm to solve the *color set size problem*, which finds for all possible substring patterns appearing in a set of strings, the number of strings in the set which possess the pattern[6,7]. We extend this algorithm so that the expression level ratio of the genes can be taken into account.

In Section 2, we give the problem definition and a brief introduction to background concepts. Section 3 describes the linear time algorithm for optimally solving the string pattern regression problem for the substring class. In Section 4, we show results of applying our algorithm to publicly available microarray data[8], and show its effectiveness, as well as its efficiency. Finally, we discuss other applications and extensions of our algorithm in Section 5.

2. Preliminaries

Notation

Let $\mathbf{R}$ be the set of real numbers. Let Σ be a finite set of characters called the *alphabet*. A concatenation of the characters of the alphabet is called a *string*, and let Σ^* denote the set of strings of finite length. The length of a string w is denoted by $l(w)$. The empty string is denoted by ε, that is, $l(\varepsilon) = 0$. Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. For any set T, let $|T|$ denote its cardinality.

A *pattern class* is a pair $\mathcal{C} = (\Pi, m)$, where Π is a set called the *pattern set* and $m : \Sigma^* \times \Pi \to \{-1, 1\}$ is the *pattern matching function*. An element $p \in \Pi$ is called a *pattern*. For a pattern $p \in \Pi$ and string $t \in \Sigma^*$, we say p *of class* $\mathcal{C}$ *matches* t if $m(t, p) = 1$, and p *of class* $\mathcal{C}$ *does not match* t, otherwise.

For example, the *substring pattern class* that we will use in this paper, is defined with the pattern set Σ^* and the pattern matching function *strstr* given by:

$$\mathit{strstr}(t, p) = \begin{cases} 1 & \text{if } p \text{ is a substring of } t \\ -1 & \text{otherwise} \end{cases}.$$

For a pattern p, we denote by $L_{\mathcal{C}}(p) = \{t \in \Sigma^* : m(t, p) = 1\}$ the set of strings in Σ^* which p of class $\mathcal{C}$ matches.

Problem Definition

We assume that we are given a set of data having two attributes: a string attribute and an objective numerical attribute. Let $D \subset \Sigma^* \times \mathbf{R}$ denote the data set.

For a pattern $p \in \Pi$, the data set D can be split into two sets: $D_p = \{(s, r) \in D \mid s \in L_{\mathcal{C}}(p)\}$, which represents the subset of D for which the pattern p matches the string attribute, and $\overline{D_p} = D - D_p$, where p does not match the string attribute. The aim of our approach is to find the pattern for which the values of the objective numeric attribute of D_p is "best distinguished" from that of $\overline{D_p}$. There can be many measures for the *goodness* of such splits. One possibility is the measure used in this paper, which is to minimize the mean squared error. We define the problem as follows:

Definition 1 (string pattern regression[5]). Given a data set $D \subset \Sigma^* \times \mathbf{R}$ and a pattern class $\mathcal{C} = (\Pi, m)$, the *string pattern regression problem* is to find $p \in \Pi$ such that the mean squared error defined below is minimized:

$$MSE(D, p) = \frac{\sum_{(s,r)\in D_p}(r - \mu(D_p))^2 + \sum_{(s,r)\in \overline{D_p}}(r - \mu(\overline{D_p}))^2}{|D|}$$

where $\mu(D') = \sum_{(s,r)\in D'} r/|D'|$ represents the average of the objective numerical attribute values of the data for any $D' \subseteq D$.

Minimizing the mean squared error can be achieved by maximizing the interclass variance (ICV)[9], defined by:

$$ICV(D,p) = |D_p|(\mu(D) - \mu(D_p))^2 + |\overline{D_p}|(\mu(D) - \mu(\overline{D_p}))^2.$$

Without loss of generality, we can subtract the average of all the numeric attributes from each numeric attribute so that their average will be 0, that is, $\mu(D) = 0$. Under this conversion, let $x_p = |D_p|$ and $y_p = \sum_{(s,r)\in D_p} r$ for a pattern p. Since $\mu(D_p) = \frac{y_p}{x_p}$ and $\mu(\overline{D_p}) = \frac{-y_p}{|D|-x_p}$, the interclass variance can be written as:

$$ICV(D,p) = f_{|D|}(x_p, y_p) = \begin{cases} 0 & \text{if } x_p = 0 \text{ or } x_p = |D| \\ y_p^2 \left(\frac{1}{x_p} + \frac{1}{|D|-x_p}\right) & \text{otherwise} \end{cases} \quad (1)$$

This measure is convenient for our calculations, and will be used for the remaining of the paper.

The string pattern regression problem is NP-hard in general, since it can contain the *consistency problem*[10,11] as a special case. A branch-and-bound algorithm which solves the string pattern regression problem exactly has been given in a previous work[5]. It is applicable to various classes of patterns such as subsequence patterns, approximate patterns, patterns with wild-cards (VLDC patterns[12,13]), etc. However, we will show in Section 3 that for the substring pattern class, the problem can be solved in linear time in the total length of the string attributes, by a clever use of generalized suffix trees[7].

Suffix Tree

A *suffix tree*[7] of string s is a rooted tree consisting of nodes (internal node or leaf) and edges, where each edge is labeled by a non-empty substring of s. Each internal node has at least 2 children, and the first character of the label on the edges to its children must be distinct. For a node v, the string obtained by concatenating the labels on the path from the root to v is denoted by $p(v)$. For any leaf v, $p(v)$ must represent a distinct suffix of s. Note that for any substring of s, the suffix tree will contain such a path from the root, which may end at a node, or perhaps on an edge. If v is an internal node, $p(v)$ represents a common prefix of suffixes of s (corresponding to leaves in the subtree below the node).

Figure 1 (A) shows an example of a suffix tree for the string "cocoa". The following algorithmic result is well known, assuming an alphabet of fixed size:

Lemma 1 (suffix tree construction[14,15]). *A suffix tree for a string s can be constructed in time linear in the length of the string.*

It is also possible to construct a *generalized suffix tree* (GST), that represents all suffixes of a set of strings, in time linear in the total length of the strings. A conceptually simple way to do this would be to create a suffix tree for a string obtained by concatenating each of the strings in the set with delimiters (e.g. for $S = \{s_1, \ldots, s_k\}$, create the suffix tree for string $s_1\$_1 s_2\$_2 \cdots s_k\$_k$, where $\$_i$ $(i = 1, \ldots, k)$

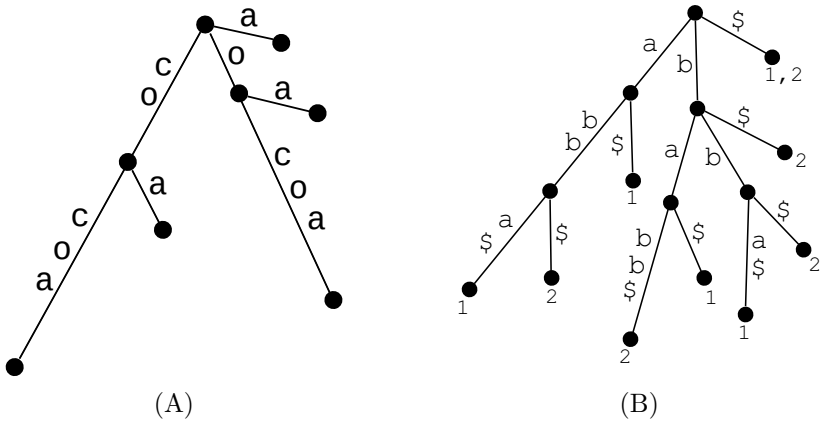

Fig. 1. (A) A suffix tree for the string "cocoa". (B) A generalized suffix tree (GST) for the strings "abba$"(1) and "babb$"(2). The character '$' represents a unique delimiter character that does not appear in any other part of the strings. Each leaf on the GST contains a list of numbers corresponding to the strings which contain that suffix. Note that the actual representation of the labels on an edge of a suffix tree is not as depicted, but is a pair of integers that identifies a substring of the given string.

are delimiters which do not appear in any of the strings). Since all strings end with a distinct character, and we do not need to consider substrings spanning across the delimiters, the tree is not constructed beyond the delimiters (i.e. paths end once a $\$_i$ appears). Therefore, each leaf in the GST corresponds to a $\$_i$ ending the path (which in turn corresponds to a string in the set) and is thus labeled. Although we omit details, it is possible to avoid the apparent increase in the alphabet size by considering all $\$_i$'s as the same character \$ and marking i on the leaves. An example of a GST is shown in Figure 1 (B). Note that the number of nodes and edges of the generalized suffix tree is linear in the length of the concatenated string, and therefore traversals on the tree can also be done in time linear in the total length.

Lowest Common Ancestor

The *lowest common ancestor* of any two nodes in a tree is the first node at which the paths from the nodes (to the root) intersect. The following is also a well known result concerning the computational complexity of *lca* queries:

Lemma 2 (efficient *lca* query[16,17]). *A query for the* lca *of any two nodes of a given tree can be answered in constant time, with linear (in the size of the tree) time preprocessing.*

3. Linear Time Algorithm for Solving the String Pattern Regression Problem for the Substring Pattern Class

For the given data set $D \subset \Sigma^* \times \mathbf{R}$, let $S = \{s \mid (s, r) \in D\}$ denote the k strings in D ($k = |D|$), $N = \sum_{s \in S} l(s)$ the total length of the strings in S, and T the generalized suffix tree of S. The string pattern regression problem is equivalent to finding the node $\hat{v}$ in T, which corresponds to the pattern $p(\hat{v})$ giving the maximum ICV, that is

$$\hat{v} = \arg\max_{v} \{ICV(D, p(v)) \mid v \text{ is a node of } T\}.$$

It is sufficient to consider only the $O(N)$ patterns represented by the internal nodes of T, because:

- patterns which do not have a corresponding path on the tree do not match any of the strings in S, and therefore do not need to be considered.
- patterns corresponding to paths which end on an edge of the tree can be extended to the next node at the end of the edge, and will still match the same set of strings, and therefore have the same ICV score.

From Equation (1), the ICV for a node v in T can be calculated if we know the number of data $x_{p(v)}$, as well as the sum of the numeric attributes $y_{p(v)}$ of the data, for which the pattern $p(v)$ matches. It has been shown[6] that $x_{p(v)}$ for all v can be computed in linear time. This fact alone does not give us $y_{p(v)}$ in linear time, but careful examination of the algorithm reveals that it can indeed be extended for incorporating such sums of real valued data associated with each string.

We first describe this algorithm by Hui[6,7] that calculates $x_{p(v)}$ in $O(N)$ time. $x_{p(v)}$ is the total number of distinctly labeled leaves in the subtree of v. To calculate this value, one can imagine a single bottom-up traversal from all of the leaves, where each leaf node will have a value of 1, and each internal node will have the summed value of its child nodes. However, if we simply sum the values as we ascend, we would be adding "too much", that is, the value calculated for each node would not be representing the number of distinctly labeled leaves in the subtree of the node. This is because if there is more than one leaf with the same label in the subtree, it would be accounted for more than once. A simple way to prevent this would be to prepare a k bit vector at each node, where each bit corresponds to a leaf label, representing whether or not the subtree contains a leaf with the label (i.e. whether the pattern represented by the node occurs in the string corresponding to the label). At each ascent, we would 'or' the bit vectors. However, this results in a $O(kN)$ algorithm. We can do away with the k bit vector if we do the bottom-up traversal k times, where, for the ith ($1 \leq i \leq k$) traversal, we would start from only the leaves corresponding to s_i, and only add the values once to a given node in each traversal. However, this requires k traversals of $O(N)$ each, which is still $O(kN)$ time. $O(kN)$ can be written as $O(k^2\bar{l})$, where $\bar{l}$ is the average length of the strings, and can be computationally intensive when k becomes large.

The idea to avoid the extra factor of k is to calculate exactly how much was "too much", and subtract it from the total sum including the redundancy. We shall call this the *correction factor*[7]. Let L be the list of leaves in the order that appears in a depth-first traversal of the generalized suffix tree, and let L_i $(1 \leq i \leq k)$ be the subsequence of leaves in L corresponding to the string s_i, in the same order. Note that with the depth-first traversal, the leaves in the subtree of a node v appear as a single contiguous sub-list in L. This means that the total number of times a subtree of v contains the *lca* of a consecutive pair of leaves in L_i, is equal to one less the number of leaves of L_i which are in the subtree of v. Therefore, by calculating the *lca* for each consecutive pair of leaves in L_i for each i $(1 \leq i \leq k)$ and counting how many times each node was selected as an *lca*, the correction factor at each node can be computed with a single bottom-up traversal from the leaves. This is illustrated in Figure 2.

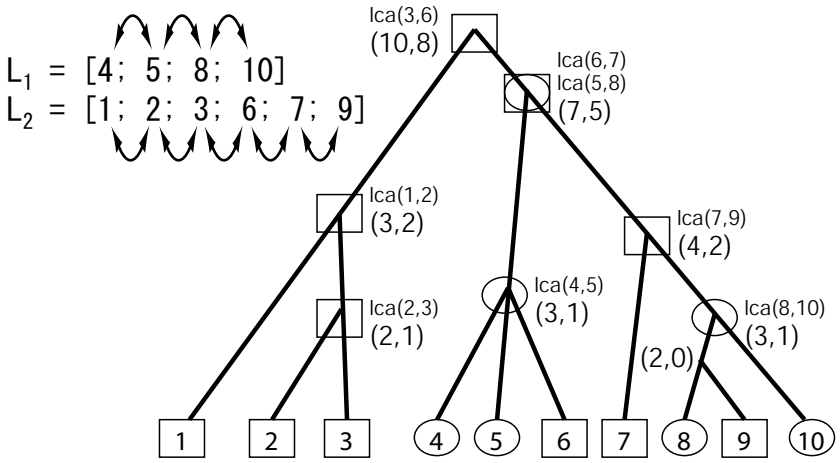

Fig. 2. Calculating the correction factors with *lca* queries on a GST. The GST represents two strings, identified with the box and oval labels at the leaves, and the *lca* for consecutive nodes of each string is depicted. The pair of numbers (x_1, x_2) at each node is calculated by a single bottom-up traversal of the tree, where x_1 represents the total number of leaves, and x_2 represents the total number of nodes that were counted as *lca*'s, in the subtree starting with the node. The number of distinct strings that the pattern corresponding to the node matches, is obtained by $x_1 - x_2$.

Now that we are ready, the algorithm can be summarized as follows: Assume each node holds a pair of integers (x_1, x_2), where x_1 is the total number of leaves in its subtree, and x_2 is the correction factor for that node. A leaf should have $(x_1, x_2) = (1, 0)$, while internal nodes are initialized to $(x_1, x_2) = (0, 0)$.

(1) Construct GST T for the set of strings $S = \{s_1, \ldots, s_k\}$ $(O(N))$.
(2) Do a depth first traversal on T to make k lists of leaves $(L_1, \ldots, L_k)$, one list

for each string s_i $(1 \leq i \leq k)$ $(O(N))$.
(3) Preprocess T for *lca* queries $(O(N))$.
(4) For each list L_i $(1 \leq i \leq k)$ obtained in step 2, calculate the *lca*'s for consecutive nodes in the list. Each time a node is chosen as an *lca*, increment the correction factor x_2 of the node by 1 ($O(N)$ since the total length of the lists is $O(N)$, and each query is constant time).
(5) Do a bottom-up traversal from the leaves, adding x_1 and x_2 of the children to those of its parents $(O(N))$.

Since all steps can be conducted in $O(N)$, the total complexity is also $O(N)$.

We next show that the above algorithm can be extended to incorporate numeric values associated with each string. Each node will hold another pair of values (y_1, y_2), where y_1 is the sum of all numeric values for all leaves in the subtree of the node, and y_2 is the correction factor for the sum. Steps 4 and 5 are modified as follows:

(4') Calculate the *lca*'s for consecutive nodes in the list obtained in step 2. Each time a node is chosen as an *lca*, 1) increment the correction factor x_2 of the node by 1, and 2) add the numeric attribute value of the corresponding string to y_2 of the node.
(5') Do a bottom-up traversal from the leaves, adding x_1, x_2, y_1, and y_2 of the children to those of its parents.

We can calculate the ICV for each node with $f_k((x_1 - x_2), (y_1 - y_2))$, and we can find the node which maximizes this value, by a final traversal of all nodes in the tree which can be done again in $O(N)$. Altogether, we have solved the string pattern regression problem for the substring pattern class in $O(N)$ time.

Theorem 1. *The string pattern regression problem can be solved in $O(N)$ time, for the substring pattern class, where N is the total length of the string attributes of the input.*

Note that although we defined the score to optimize as ICV in Equation (1), any other scoring function of x_p and y_p can be used.

Finding Multiple Motifs

Assuming a model where each motif contributes additively to the numeric value, we can perform a greedy iteration for finding multiple motifs[4], by first finding the best pattern which contributes most to the numeric value, then subtracting the effect that the pattern has on the numeric attributes. The procedure is then repeated with the modified numeric attributes to find the next pattern with the most influence.

More precisely, for the best pattern $\hat{p}$, we subtract $\mu(D_{\hat{p}})$ from the numeric attributes of data in $D_{\hat{p}}$, and $\mu(\overline{D_{\hat{p}}})$ from numeric attributes of data in $\overline{D_{\hat{p}}}$, and finally adjust the values so that the $\mu(D) = 0$ again. Notice that the suffix tree

does not change between iterations, and can be reused for the next iteration. Also, since the tree does not change, the preprocessing for the *lca* queries need only be conducted once. Therefore, subsequent iterations can be computed much faster than the first.

4. Application to Gene Expression Data

We show the effectiveness and efficiency of our algorithm by applying it to publicly available gene expression data by Spellman *et al.*[8], which are microarray data obtained from time course experiments concerning the cell cycle of *S. cerevisiae*. We implemented the algorithm in the Objective Caml language[18]. We use the on-line suffix tree construction algorithm of Ukkonen[15], and an efficient *lca* query method by Bender and Farach-Colton[17], based on range minimum queries. Runtime performance was measured on a dual Xeon 2.2GHz PC with 2GB of memory.

Motifs Detected

The same dataset has been analyzed previously by several other methods, and we evaluate the motifs discovered by our algorithm by comparing it to the output of a previously proposed, very similar method[4] mentioned in Section 1. We give here the results for the 14-minute time point in the α-synchronized cell-cycle experiment[8]. There were 6177 genes in the combined data, but we discard genes with no expression ratio measurements, as well as those whose sequence we could not retrieve from SCPD[19]. For the 5976 genes that could be used, we take the 600 ($-600 \sim -1$) basepair sequence upstream of the gene from SCPD, and pair them with their log expression ratios. We run the algorithm for 20 iterations (i.e. output 20 motifs), only considering motifs whose lengths are less than or equal to 7 (the same length as used in the previous analysis[4]). Although we need not limit the motif length since we are able to obtain the optimal pattern in $O(N)$ time, we limited our output since allowing longer motifs seemed to make the algorithm capture very long motifs common in very few genes (2 or 3) whose expression level ratios were similarly relatively higher or lower than the rest.

Table 1 shows the first 20 motifs discovered through the iterative procedure. Notice that a positive value for y_p suggests that the motif *enhances* transcription, whereas a negative value suggests that it *inhibits* transcription. Comparing with the 11 motif result[4] obtained from the same expression data, we mark the motifs with edit distance less than or equal to 1, from the closest motif of the 11. We also mark *sup* (resp. *sub*) if the string is a superstring (resp. substring) of a motif in the 11. We can see that 4 exactly equivalent motifs appear, as well as 6 others which were very close (although AAGGGG and CAGGGG correspond to the same motif AGGGG), showing that the motifs obtained from our method are comparable to those obtained from the previous method[4]. Motifs corresponding to known elements, MCB (ACGCGT), STRE (AGGGG and CCCCT), and SFF (GTMAACAA), are also shown in the table.

Table 1. The first 20 motifs detected from the 14-minute time point in the α-synchronized cell-cycle microarray experiment of Spellman *et al.* The superscript on the patterns represent the edit distance from (*sup* (resp. *sub*) represents that it is a superstring (resp. substring) of) the closest motif in the 11 motif result of Bussemaker *et al.*, based on the same data.

iteration	pattern p	ICV	total occurrence	x_p	y_p
1	$\mathrm{AAATTTT}^{1}$	18.12	1845	1549	−144.2
2	ACGCGT^{0} (MCB)	12.54	414	371	+66.06
3	AAGGGG^{sup} (STRE)	9.553	662	616	+72.65
4	$\mathrm{CGATGAG}^{0}$	8.27	348	331	−50.86
5	$\mathrm{CTCATCG}^{0}$	5.54	326	315	−40.67
6	$\mathrm{TGACGCG}^{sup}$	4.28	106	104	+20.91
7	$\mathrm{TGAAAAA}^{sup}$	3.45	2018	1678	−64.55
8	ATAAGGA	3.30	396	387	+34.54
9	CCCC^{sub} (STRE)	3.00	4877	2669	+66.56
10	CCTGGAA	2.62	189	186	+21.74
11	TTCAAAA	2.41	1134	1020	−45.18
12	AAAAGG	2.40	2845	2227	+57.97
13	$\mathrm{TAAACAA}^{0}$ (SFF)	2.29	761	698	−37.56
14	CAGGGG^{sup} (STRE)	2.28	263	247	+23.24
15	TTTTTC	2.20	6582	3751	−55.40
16	GCTGGGT	2.10	95	94	−13.94
17	CGAACCA	1.96	121	121	+15.25
18	CTGGGCT	1.98	94	94	−13.52
19	GGCACAC	1.90	98	97	+13.47
20	CATCTCA	1.87	363	321	−23.82

From gene expressions of other microarray experiments at different time points (details not shown), we were able to find other known motifs, such as the SCB element (CGCGAAA). We also obtained ATGCGAA, a motif similar to the 'histone' motif $(\mathrm{ATGCGAAR})^{8}$ in the 8th iteration of the 35-minute time point of α-synchronized cell-cycle experiment, while it is reported that it was not detected in the previous analysis[4].

We note that the sequence data we used seems to be slightly different from those used in the previous analysis[4], since the same patterns showed different total occurrence counts.

Runtime Performance

For the experiments described in the previous subsection, the time for the first iteration took around 102 seconds: the construction of the suffix tree around 65 seconds, and preprocessing for the *lca* queries around 15 seconds. The rest of the

time was spent calculating the correction factors and finding the node giving the optimal pattern. Each subsequent iteration takes around 19 seconds each, for a total of 467 seconds for the 20 iterations.

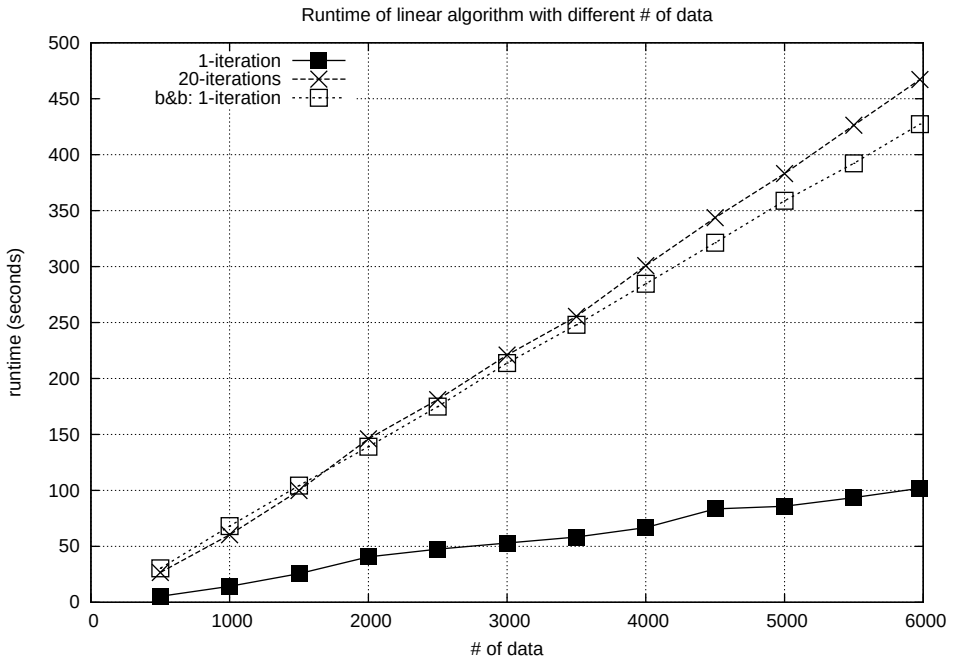

Fig. 3. The running times on a random subset of the total data, varying the sizes of the subset. The times for one iteration (filled box), 20 iterations (cross) are shown for the linear time algorithm (finding the optimal pattern), and the time for one iteration (clear box) using the branch-and-bound algorithm by Bannai *et al.* with the string pattern class (limiting the pattern length to 7) is shown. Our linear time algorithm is much faster for one iteration. The difference grows for subsequent iterations, since we can reuse the suffix tree structure between iterations.

To demonstrate that our algorithm actually runs in linear time, we also measured the runtimes of our algorithm when applied to random subsets of the total set of genes. Figure 3 shows the runtime of our algorithm when we vary the sample size, and we can see that the runtime grows linearly, as expected.

The runtimes using a naïve string matching algorithm with the branch-and-bound enumeration of patterns[5] is also shown. The maximum length of the pattern in the enumeration was limited to 7. This provides an *optimistic* lower bound for the computation time of the previous method[4] when using an enumerative strategy. Note that this does not include the time for calculation that would be required for the linear fitting between the occurrence counts and expression ratios. Also, it is not apparent whether the branch-and-bound strategy can be applied to the original

scoring function.

The branch-and-bound algorithm is fairly efficient, but our linear time algorithm is at least 4 times faster for one iteration. Furthermore, since we can reuse the suffix tree between iterations, we can do 20 iterations with the linear time algorithm while we only do one iteration with the branch-and-bound algorithm, meaning that our new algorithm is almost 20 times faster.

5. Discussion

We present several possible extensions to our algorithm which can be made without largely sacrificing its efficiency.

5.1. *Scoring Function*

In Section 4 we noted that when not limiting the pattern length, the algorithm seemed to come up with very long patterns common to the upstream of 2 or 3 genes. In fact, some genes seemed to have exactly the same upstream regions. Although the biological meanings of such long common sequences remain to be investigated, perhaps choosing a more appropriate scoring function could prevent such effects if it is deemed undesirable. Note that this is an interesting observation which could be made since our method is able to find the optimal pattern with respect to a certain scoring function, having no limit in the length of the pattern. As noted in Section 3, although we defined the score to optimize as ICV in Equation (1), any other function of x_p and y_p can be used. Also notice that for each node v of the generalized suffix tree, the length of the pattern $l_p = |p(v)|$ can be recorded on each node in linear time by a simple top down traversal of the tree. The scoring function could also incorporate this information and take the form $f(x_p, y_p, l_p)$. Furthermore, if some assumptions on the monotonicity of f with respect to l_p can be made (that is, if for any $l : l_1 \leq l \leq l_2$, we have $\max f(x_p, y_p, l) = \max\{f(x_p, y_p, l_1), f(x_p, y_p, l_2)\}$), it is not difficult to see that the string pattern regression problem for scoring function f can also be solved in linear time, provided that f can be computed in constant time.

Another possible extension could be in the number of microarray experiments. We have only considered the case where we use one gene expression data per run, but by a suitable extension to the scoring function, our method can use multiple microarrays obtained at different conditions. The data for each gene would consist of the upstream sequence together with a *vector* of gene expression ratios, and the algorithm would find the pattern p with optimal score $f(x_p, \mathbf{y}_p)$, where $\mathbf{y}_p$ is the sum of the gene expression ratio vectors of genes which contain p. The runtime would increase by a factor of the number of microarray data used.

5.2. *Substring Patterns which Match in Reverse Complement*

It is known that for several transcription factors, the binding site can occur on either strand of the upstream DNA region. Therefore, it can be biologically mean-

ingful to consider pattern matching functions where a string matches if the string itself is contained as a substring, *or* its reverse complement (i.e. exchange A $\leftrightarrow$ T, C $\leftrightarrow$ G, and reverse the direction of the pattern) is contained as a substring of the sequence. This is partially indicated from the results of Table 1, where we can see, for example, that (2:"ACGCGT") is self-complement, and also (4:"CGATGAG") and (5:"CTCATCG"), are complements of each other.

Such reverse complement patterns can be handled in our algorithm as follows: when given a set of strings $S = \{s_1, \ldots, s_k\}$, construct the generalized suffix tree for the string $s_1\$_1 s_1'\$_1 s_2\$_2 s_2'\$_2 \cdots s_k\$_k s_k'\$_k$, where s_i' represents the reverse complement string of s_i, and $\$_i$ $(i = 1, \ldots, k)$ are delimiters which do not appear in any of the strings. The tree is not built deeper than any $\$_i$. Since $x_{p(v)}$ and $y_{p(v)}$ of a node v depends on the number of distinct $\$_i$'s which appear in the subtree of v, running the algorithm on the new GST will produce the correct results as in the original case. The time complexity will increase by a factor of 2, but is still linear in the length of the total length of the input sequences.

Table 2 shows the first 20 motifs obtained with this extension, run on the same data as in Table 1. Similar motifs are found again, but with some notable additions: the motif (13: "CGGGTAA/TTACCCG") which is known as the REB1 binding site, and (16: "AACCCA/TGGGTT") which is known as the RAP1 binding site.

5.3. *Other Applications*

The algorithm described in this paper was successfully employed[20] as a subroutine to extend a previous Bayesian network inference method[21] that estimates a network of genes from microarray data.

We roughly describe the method below: First, estimate an initial network using the previous method[21]. Consider a certain gene g as a possible transcription factor, and use our algorithm to find a motif in the upstream regions that is most correlated with the *likelihood* (which is a numeric which depends on the structure of the current network) that a given gene is a child of g. The rationale for this is that the initial gene network estimation is *fairly* good, and genes which are truly the child of g should have relatively higher likelihoods, while other genes should have relatively lower likelihoods. Our motif detection algorithm finds a sequence common to genes with higher likelihood of being regulated by g, and not appearing in genes with lower likelihood. After the motif is obtained, increase the *prior probability* that a gene which possesses this motif is actually a direct child of g. After updating the prior probabilities, the network is re-estimated, and the process repeated until convergence.

The integration of network inference and motif discovery is shown to improve the accuracy of the inferred network, compared to the inference method which does not use the sequence information. The key to this integration is that our method can consider the inaccuracy of the network structure with the likelihoods. Also, since the inference of the network involves numerous iterations of motif discovery

Table 2. The first 20 motifs detected from the 14-minute time point in the α-synchronized cell-cycle microarray experiment of Spellman *et al*, allowing reverse complement matches. The superscript on the patterns represent the edit distance from (*sup* represents that it is a superstring of) the closest motif in the 11 motif result of Bussemaker *et al.*, based on the same data.

iteration	pattern p	ICV	total occurrence	x_p	y_p
1	CGATGAG^0/CTCATCG^0	20.14	674	630	−106.5
2	ACGCGT^0/ACGCGT^0 (MCB)	12.42	828	371	+65.73
3	AAAATTT^0/AAATTTT^1	10.82	3755	2252	−123.3
4	AGGGG^0/CCCCT^0 (STRE)	8.98	3155	2346	+113.2
5	CGCGTCA/TGACGCG^{sup}	5.75	185	170	+30.81
6	ATTTTTC^{sup}/GAAAAAT	5.56	2990	2359	−89.12
7	CATCTCA/TGAGATG	3.52	645	564	−42.37
8	ATGCAGG/CCTGCAT	3.24	286	278	+29.31
9	CATCGCA/TGCGATG	3.11	483	450	−35.97
10	GGCACAC/GTGTGCC	2.80	187	183	+22.27
11	AAAACAA/TTGTTTT	2.77	2820	2140	−61.73
12	AAAAAAT/ATTTTTT^1	2.59	5560	3497	−61.36
13	CGGGTAA/TTACCCG (REB1)	2.53	676	639	−38.00
14	CAGGGAG/CTCCCTG	2.50	157	156	+19.47
15	TAAACAA^0/TTGTTTA (SFF)	2.30	1449	1259	−47.80
16	AACCCA/TGGGTT (RAP1)	2.31	1371	1211	−47.26
17	AAGCTCG/CGAGCTT	2.39	260	254	+24.11
18	GAGGAGA/TCTCCTC	2.20	509	486	+31.32
19	CAGTGAG/CTCACTG	2.00	238	230	−21.03
20	CCTGGAA/TTCCAGG	1.92	370	351	+25.19

and network structure inference, the efficiency of the motif discovery algorithm is essential.

6. Conclusion

We presented an efficient, linear time algorithm which can be used for finding putative regulatory elements in the upstream regions of genes, by combining sequence information together with the information of gene expression obtained from microarray experiments. This simple method seems to balance algorithmic efficiency with biological relevance: we were able to rediscovery several biologically meaningful motifs very efficiently. We have shown that the same algorithm can be extended in several ways without sacrificing its efficiency, and have successfully demonstrated a variation of the algorithm considering reverse complement matches. We also mention that the algorithm has also been shown to be effective in other situations where it is required to find, efficiently, patterns whose appearance is most correlated with

a numeric value.

Acknowledgments

The authors would like to acknowledge the anonymous referees for their helpful comments. This research was supported in part by Grant-in-Aid for Encouragement of Young Scientists (B), and Grant-in-Aid for Scientific Research on Priority Areas (C) "Genome Biology" from the Ministry of Education, Culture, Sports, Science and Technology of Japan.

References

1. S. Tavazoie, J. D. Hughes, M. J. Campbell, R. J. Cho and G. M. Church, "Systematic determination of genetic network architecture," *Nat. Genet.* **22**, 281–285 (1999).
2. J. Vilo, A. Brazma, I. Jonassen, A. Robinson and E. Ukkonen, "Mining for putative regulatory elements in the yeast genome using gene expression data," In *Proc. of the 8th International Conference on Intelligent Systems fo Molecular Biology (ISMB)*, 384–394 (2000).
3. U. Ohler and H. Niemann, "Identification and analysis of eukaryotic promoters: recent computational approaches," *Trends Genet.* **17**, 50–60 (2001).
4. H. J. Bussemaker, H. Li and E. D. Siggia, "Regulatory element detection using correlation with expression," *Nat. Genet.* **27**, 167–171 (2001).
5. H. Bannai, S. Inenaga, A. Shinohara, M. Takeda and S. Miyano, "A string pattern regression algorithm and its application to pattern discovery in long introns," *Genome Informatics* **13**, 3–11 (2002).
6. L. Hui, "Color set size problem with applications to string matching," In *Proc. of the 3rd Annual Symposium on Combinatorial Pattern Matching (CPM)* LNCS **644**, 230–243 (1992).
7. D. Gusfield, *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.
8. P. T. Spellman, G. Sherlock, M. Q. Zhang, V. R. Iyer, K. Anders, M. B. Eisen, P.O. Brown, D. Botstein and B. Futcher, "Comprehensive identification of cell cycle-regulated genes of the yeast *Saccharomyces cerevisiae* by microarray hybridization," *Mol. Biol. Cell* **9**, 3273–3297 (1998).
9. Y. Morimoto, H. Ishii and S. Morishita, "Efficient construction of regression trees with range and region splitting," *Machine Learning* **45**, 235–259 (2001).
10. T. Jiang and M. Li, On the complexity of learning strings and sequences, In *Proc. 4th ACM Conf. on Computational Learning Theory*, 367–371, (1991).
11. S. Miyano, A. Shinohara, T. Shinohara, Which classes of elementary formal systems are polynomial-time learnable?, In *Proc. 2nd Workshop on Algorithmic Learning Theory*, 139–150 (1991).
12. S. Inenaga, H. Bannai, A. Shinohara, M. Takeda and S. Arikawa, "Discovering best variable-length-don't-care patterns," In *Proc. 5th International Conference on Discovery Science* LNCS **2534**, 86–97 (2002).
13. M. Takeda, S. Inenaga, H. Bannai, A. Shinohara, and S. Arikawa, "Discovering Most Classificatory Patterns for Very Expressive Pattern Classes", In *Proc. 6th International Conference on Discovery Science*, LNCS **2843**, 486–493 (2003).
14. P. Weiner, "Linear pattern matching algorithms," In *Proc. 14th IEEE Annual Symp. on Switching and Automata Theory*, 1–11 (1973).

15. E. Ukkonen, "On-line construction of suffix trees," *Algorithmica* **14**, 249–260 (1995).
16. D. Harel, and R. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.* **13**, 338–355 (1984).
17. M. A. Bender and M. Farach-Colton, "The LCA problem revisited," In *Proc. Latin American Theoretical Informatics (LATIN)* LNCS **1776**, 88–94 (2000).
18. X. Leroy, D. Doligez, J. Garrigue and J. Vouillon, The Objective Caml system: `http://caml.inria.fr/` (1996–).
19. J. Zhu and M. Zhang, "SCPD: A promoter database of the yeast *Saccharomyces cerevisiae*," *Bioinformatics* **15**, 607–11 (1999).
20. Y. Tamada, S. Kim, H. Bannai, S. Imoto, K. Tashiro, S. Kuhara and S. Miyano, "Estimating gene networks from gene expression data by combining Bayesian network model with promoter element detection," *Bioinformatics*, **19** (Supplement: *Proceedings of the European Conference on Computational Biology 2003*), ii227–ii236 (2003).
21. S. Imoto, T. Goto and S. Miyano, "Estimation of genetic networks and functional structures between genes by using Bayesian networks and nonparametric regression," In *Proc. Pacific Symposium on Biocomputing (PSB)* **7**, 175–186 (2002).