Supplementary Material to TCBB Manuscript: Identification of Regulatory Modules in Time-Series Gene Expression Data using a Linear Time Biclustering Algorithm

Sara C. Madeira, Miguel C. Teixeira,

Isabel Sá-Correia and Arlindo L. Oliveira, *Member, IEEE*

I. Complexity and Implementation Issues

Since the complexity of the algorithm CCC-Biclustering depends strongly on the suffix tree construction and on the data structures used, we will revise here the Ukkonen´s algorithm for suffix tree construction at a higher level and give the intuition behind its linear time construction. We will also described the data structures used to achieve the bound $O(|R||C|)$ in the construction of the generalized suffix tree T used in the CCC-Biclustering algorithm. However, the reader is pointed to Gusfield [1] for a very good explanation about the linear time construction of suffix trees. The short explanation provided here is based on that.

Sara C. Madeira is with the Knowledge Discovery and BIOinformatic (KDBIO) team of INESC-ID, Lisbon, Portugal. She is also with University of Beira Interior, Covilhã, Portugal, and Instituto Superior Técnico, Technical University of Lisbon, Portugal. E-mail: smadeira@di.ubi.pt.

Miguel C. Teixeira is with the Biological Sciences Research Group, Centre for Biological and Chemical Engineering/IBB-Institute for Biotechnology and Bioengineering, Instituto Superior Técnico, Technical University of Lisbon, Portugal. E-mail: mnpct@ist.utl.pt.

Isabel Sá-Correia is with the Biological Sciences Research Group, Centre for Biological and Chemical Engineering/IBB-Institute for Biotechnology and Bioengineering, Instituto Superior Técnico, Technical University of Lisbon, Portugal. She is also with Instituto Superior Técnico, Technical University of Lisbon, Portugal. E-mail: isacorreia@ist.utl.pt.

Arlindo L. Oliveira is with the Knowledge Discovery and BIOinformatic (KDBIO) team of INESC-ID, Lisbon, Portugal. He is also with Instituto Superior Técnico, Technical University of Lisbon, Portugal. E-mail: aml@inesc-id.pt.

A. Ukkonen's Algorithm

Ukkonen's algorithm to construct suffix trees uses the concepts of *implicit suffix tree* and *suffix link* to achieve a linear time construction. We will first introduce these concepts and then describe the algorithm.

An *implicit suffix tree* for a string S is a tree obtained from the suffix tree T constructed for the string $S\$$ by removing every copy of the symbol $\$$ from de edge labels of the tree, then removing any node v that does not have at least two children. In particular, an implicit suffix tree for a prefix $S[1..i]$ of string S is similarly defined by taking the suffix tree for $S[1..i]\$$ and deleting $\$$ symbols, edges and nodes as above. The implicit suffix tree of the string $S[1..i]$ is denoted by T_i, $1 \geq i \geq |S|$. Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes a (possibly empty) substring. For any internal node v with string-label $x\alpha$, if there is another node $s(v)$ with string-label α, then a pointer from v to $s(v)$ is called a *suffix link*. The pair $(v, s(v))$ will denote the suffix link from v to $s(v)$. As a special case, if α is empty, $x\alpha$ has a suffix link leading to the root, $(v, root)$.

Ukkonen's algorithm starts by constructing an implicit suffix tree T_i for each prefix of $S[1...i]$ of a string S, starting from T_1 and incrementing i by one until $T_{|S|}$ is built, where $|S|$ is the number of characters in S. The true suffix tree from S is then constructed from $T_{|S|}$.

The algorithm is divided into $|S|$ phases (see Algorithm 1). In phase $i+1$, the implicit suffix tree T_{i+1} is constructed from T_i. Each phase $i+1$ is further divided into $i+1$ *extensions*, one for each of the $i+1$ suffixes of $S[1...i+1]$. In *extension* j of phase $i+1$, the algorithm first finds the end of the path from the root that is labeled with the substring $S[j...i]$. It then extends this substring by adding the character $S[i+1]$ to its end (unless $S[i+1]$ is already there). As such, in phase $i+1$, the string $S[1...i+1]$ is first inserted in the tree, followed by its suffixes $S[2...i+1], \ldots, S[|S|...i+1]$ (in extensions $1, \ldots, |S|$, respectively). The extension $i+1$ of phase $i+1$ extends the *empty* suffix of $S[1...i]$, that is, inserts the single character string $S[i+1]$ into the tree (unless it is already there).

The algorithm follows *three suffix extension rules* in order to make sure that the suffix $S[j...i+1]$ is in the tree after the extension of $S[j...i]$ with character $S[i+1]$:

(Rule 1) If path $S[j..i]$ ends at a leaf, concatenate $S[i+1]$ to the end of its edge label;

(Rule 2) If path $S[j...i]$ ends before a leaf, and doesn't continue by $S[i+1]$, connect the end of the path to a new leaf j by an edge labeled by character $S[i+1]$. If the path ended at the

Algorithm 1: High-Level Ukkonen's Algorithm [1]

input: Discretized gene expression matrix A

1 Construct the implicit suffix tree T_1. // T_1 is the single edge labeled by the character $S[1]$.
2 **for** i *from* 1 *to* $|S|-1$ **do**
  //PHASE $i+1$: constructs T_{i+1}.
  //Updates T_i (with all suffixes of $S[1...i]$) to T_{i+1} (with all suffixes of $S[1...i+1]$).
3 **for** j *from* 1 *to* $|i+1$ **do**
    //EXTENSION j: ensures that suffix $S[j...i+1]$ is in T_{i+1}.
4   Find the end of the path from the root that is labeled $S[j...i]$ in the current implicit suffix tree. If needed, extend that path by adding the character $S[i+1]$ to its end.

middle of an edge, split the edge and insert a new internal node as a parent of leaf j;

(Rule 3) If the path can be continued by $S[i+1]$ do nothing (the suffix $S[j...i+1]$ is already in the tree).

Given these suffix extension rules, once the end of a suffix $S[j...i]$ of $S[1..i]$ is in the current tree, the execution of the extension rules in order to ensure that suffix $S[1...i+1]$ is in the tree can be performed in constant time. The key issue in implementing Ukkonen's algorithm is then how to locate the end of all the $i+1$ suffixes of $S[1...i]$. This is achieved by using *suffix links* and *three implementation tricks*.

Lets start by an intuitive motivation about how suffix links can be used to speed up path traversals. Extension j (of phase $i+1$) finds the end of the path $S[j...i]$ in the tree (and extends it with character $S[i+1]$. Similarly, extension $j+1$ finds the end of the path $S[j+1...i]$. Assuming that v is an internal node with string-label $S[j]\alpha$ on the path $S[j...i]$, then we can avoid traversing the path α when locating the end of path $S[j+1...i]$, by starting the traverse from the suffix link of v, $s(v)$. This can be done because these suffix links always exist and are in fact easy to set. The certainty about the existence of the suffix link $(v, s(v))$ comes from the observation that if an internal node v is created during extension j (of phase $i+1$), then extension $j+1$ will find out the node $s(v)$. Why is this true? Let v be a node with string-label $x\alpha$. This node can only be created by extension *Rule 2*, that is, v can only be inserted at the end of path $S[j...i]$, which continued by some character $c \neq S[i+1]$. In this context, the paths $x\alpha c$ and αc have been inserted in the tree before phase $i+1$. This means that in extension $j+1$, node $s(v)$ is either found since it is already in the tree or created at the end of path $\alpha = S[j+1...i]$.

Consider now the extensions of phase $i+1$. Extension 1 extends path $S[1...i]$ with character

$S[i+1]$. This can be done easily since the path $S[1...i]$ always ends at leaf 1, and is thus extended by *Rule 1*. As such, extension 1 can be performed in constant time, if we maintain a pointer to the edge at the end of $S[1...i]$. What about the subsequent extensions $j+1$, $j = 1, \ldots, i$? Since extension j has located the end of the path $S[j...i]$ we can start from there, and walk up at most one node either to the root, or to a node v that has a suffix link $s(v)$ from it.

In case we have walked up to the root, we just have to traverse the path $S[j+1...i]$ explicitly downwards starting from the root.

In case we have followed the suffix link to node $s(v)$ let $x\alpha$ be the edge-label of v. This means $S[j...i] = x\alpha\beta$ for some $\beta \in \Sigma$. In this scenario, we just have to follow the suffix link of v, and continue by matching β downwards from node $s(v)$ (which is now labeled by α). Having found the end of path $\alpha\beta = S[j+1...i]$, we apply the extension rules to ensure that it is extended with the character $S[i+1]$. Finally, if a new internal node u was created in extension j, we set its suffix link to point to the end node of path $S[j+1..i]$.

However, we can speed up these explicit traversals by using the *implementation trick 1 (skip/count in Gusfield [1])*: each path $S[j...i]$, which is followed in extension j, is known to exist in the tree, as such, the path can be followed by choosing the correct edges, instead of examining each character. Let $S[k]$ be the next character to be matched on path $S[j...i]$. Now an edge labeled by $S[p...q]$ can be traversed simply by checking that $S[p] = S[k]$, and skipping the next $q-p$ character of S[j...i]. This improves the time to traverse a path from time proportional to its string-depth to time proportional to its node-depth.

With the suffix links and the first trick the total time of a phase is now $O(|S|)$. However, there are $|S|$ phases and the total time bound is still $O(|S|^2)$. In order to achieve the desired linear time bound we just need a simple implementation detail and two more implementation tricks.

The implementation detail concerns *edge-label compression*. In fact, with the current implementation, the edge-labels in the suffix tree might contain $O(|S|)$ characters which makes the space required for the suffix tree to be $O(|S|^2)$. As the time of the algorithm is at least as large as the size of its input, that many characters makes an $O(|S|)$ time bound impossible [1]. However there is a simple alternative scheme for edge-labeling: instead of explicitly write a substring $S[p...q]$ as edge-label, we can write only a *pair of indices* on the edge, (p, q), specifying the start and end positions of that substring in S. Since the algorithm has a copy of S, it can locate any particular character in S in constant time given its position in the string.

For example, when matching along an edge, the algorithm uses the index pair written on the edge to retrieve the needed characters from S and then performs the comparisons on those characters. The extension rules are also easily implemented in this labeling scheme: when the extension rule 2 applies in a phase $i+1$, we just have to label the newly created edge with the index pair $(i+1, i+1)$, and when rule 1 applies (on a leaf edge), we only need to change the index pair on that leaf edge from (p, q) to $(p, q+1)$. Since the number of edges is at most $2|S|-1$, the suffix tree uses only $O(|S|)$ symbols and $O(|S|)$ space.

The *implementation trick 2 ("show stopper" in Gusfield [1])* is based on the observation that some extensions can be found unnecessary to compute explicitly. In fact, Rule 3 is a "show stopper" since if the path $S[j...i+1]$ is already in the tree, so are the paths $S[j+1...i+1], \ldots, S[i+1]$. As such, phase $i+1$ can be finished at the first extension j that applies Rule 3.

The *implementation trick 3 ("once a leaf, always a leaf" in Gusfield [1])* is based on the observation that a node created as a leaf remains a leaf thereafter because no extension rule adds children to a leaf. In fact, if extension j created a leaf (numbered j), the extension j of any later phase $i+1$ applies Rule 1 (concatenating the next character $S[i+1]$ to end of the edge-label of j). As such, explicit applications of Rule 1 can be eliminated as follows: using edge-label compression described above and representing the end position of each terminal edge by a global value e standing for "the current end position". This means that in phase $i+1$, when a leaf edge is first created ad would normally be labeled with $S[p...i+1]$ instead of writing the pair of indexes $(p, i+1)$, as explained above, we write (p, e). The symbol e is a global index and is set to $i+1$ once in each phase. In phase $i+1$, since we know that rule 1 will apply in extensions 1 through j_i at least, we do not need explicit work to implement those j_i extensions. Instead, we only need constant time to increment the value of e and then do explicit work for (some) extensions starting with $j_i + 1$.

The *Single Phase Algorithm (SPA)* [1] summarizes the implementation of phase $i+1$:

Algorithm 2: Single Phase Algorithm (SPA) [1]

input: S and T_i

1 $e = i+1$ (implements extensions $1...j$ implicitly).
2 Compute extensions $j_{i+1}...j$ until $j > i+1$ or Rule 3 is applied in extension j.
3 $j_{i+1} = j-1$ (for the next phase).

Using suffix links, edge-compression and tricks 1, 2 and 3, Ukkonen'a algorithm builds the implicit suffix trees T_1 through $T_{|S|}$ in $O(|S|)$ total time. Moreover, in order to create the true suffix tree T we just have to use the final implicit suffix tree $S_{|S|}$. We first add a string terminal symbol \$ to the end of S and let Ukkonen's algorithm continue with this symbol. Next, we just have to replace every index e on every leaf node with the value $|S|$. This is achieved by a simple traversal of the tree, visiting each leaf each, which takes $O(|S|)$ [1].

In order to construct the generalized suffix tree for a set of string $\{S_1, ..., S_{|R|}\}$ all with the same length $|C|$ and sharing the same alphabet $\Sigma^{'}$, we first build the implicit suffix trees from $T_{S_1^1}$ through $S_1 T_{S_1^{|C|}}$ using Ukkonen's algorithm. We then insert the strings S_i on the tree starting with the last implicit suffix tree of S_{i-1}: $T_{S_{i-1}^{|C|}}$, $2 \leq i \leq |R|$. Finally we transform the last implicit suffix tree $T_{S_{|R|}^{|C|}}$ in a true suffix tree by adding a string terminal symbol $\$_i$ to the end of S_i and let Ukkonen's algorithm continue with these symbols one at the time. Assuming that the nodes in the implicit suffix tree organize their children by lexicographic order of the first character of their edge-labels, this is done in a way such that $\$_1 > \ldots > \$_{|R|}$ and every string terminator $\$_i$ is lexicographically smaller then any character in Σ'. This enables the insertion of leaves corresponding to terminators in the root node (and other nodes) always in the first position of the data structures storing the children of each node. This is done in $O(|R||C|)$, since there are $|R|$ strings of length $|C|$ and inserting each string in the suffix tree using Ukkonen's algorithm takes $O(|R||C|)$ as we have seen above.

B. Data Structures used in the Generalized Suffix Tree Construction

We use three types of nodes in the construction of the generalized suffix tree T: the root, internal nodes and leaf nodes. The root stores an array called $children$ with $|C||\Sigma| + |R|$ positions where each position is a pointer to the first of its "potential" children, which can either be internal nodes, leaf nodes or a null pointer. The array is sorted in lexicographic order of the first character of the edge-label in the nodes. The first R positions store the $|R|$ string terminators, the next positions store the nodes whose first character in the edge-label starts with $\Sigma'[1]....\Sigma'[|C|]...\Sigma'[|C|\Sigma]]$. In this setting, $children[j + |R|]$ is null if there is not a suffix of any of the strings S_i starting with the character in $\Sigma'[j]$, that is, if the potential node whose edge-label starts with the character $\Sigma'[j]$ does not exists.

Each internal node v stores a pointer to its first child, a pointer to its right sibling, a pointer

to the node we get by following its suffix link (if it exists), the index pair representing its edge-label (start and end position os the substring), its string-length, $P(v)$, the number of leafs in its subtree, $L(v)$ and a flag indicating if it corresponds to a maximal CCC-Bicluster or not. The first child of the node is the first element of a linked-list of nodes (either internal nodes or leaves nodes) corresponding to its children sorted in lexicographic order of the first character of their edge-labels. The right sibling of each internal node v is also the first element of a linked-list storing all its siblings (nodes whose parent is the parent of v).

Leaf nodes store the same information as internal nodes except the pointer to the first child.

II. Experimental Results with Synthetic Data

Table I shows the top CCC-Biclusters discovered sorted in ascending order of p-value. It is clear from the presented results that the CCC-algorithm coupled with the statistical significance test described in the paper is able to identify the CCC-Biclusters planted together with a number of highly overlapping CCC-Biclusters[1].

References

[1] D. Gusfield. *Algorithms on strings, trees, and sequences*. Computer Science and Computational Biology Series. Cambridge University Press, 1997.

[1]Lost genes are caused by the (artificial) way in which CCC-Biclusters were planted. When two or more CCC-Biclusters are overlapping, the expression patterns in the overlapping submatrices are those of the last planted CCC-Bicluster. For this reason, the genes in overlapping zones are lost for the previously planted CCC-Biclusters.

TABLE I

TOP CCC-BICLUSTERS RECOVERED WITHOUT FILTERING HIGHLY OVERLAPPING EXPRESSION PATTERNS (AFTER SORTING THE DISCOVERED CCC-BICLUSTERS USING THE STATISTICAL SIGNIFICANCE p-VALUE).

ID	*Expression Pattern*	*#Time-Points*	*#Genes*	*p-Value*	*Closest planted CCC-Bicluster*
24475	DDNUDDNDDNDD	12(35-46)	18	4.13E-56	MATCH 7
5790	UDNNDUNUDNDD	12(26-37)	19	1.37E-55	MATCH 8
17868	NUNUUNDDNDNU	12(21-32)	19	3.72E-55	MATCH 1
25020	DNDNDDNNNNDD	12(33-44)	18	6.10E-54	MATCH 10
2438	UDDUNUDDU	9(37-45)	23	8.43E-40	MATCH 6 LOST 1 GENE[1]
5937	UUDNNDUNUDNDD	13(25-37)	12	2.76E-37	OVERLAP 8
15158	NUNDNNDDUNN	11(30-40)	16	4.08E-37	MATCH 3 LOST 3 GENES[1]
34531	UNUDUDNDUU	10(2-11)	16	7.17E-34	MATCH 5
16797	NDUNDNUUU	9(25-33)	20	8.20E-33	MATCH 9
23592	DDUNUDDU	8(28-45)	24	1.38E-31	OVERLAP 6
15893	NNDUNUDN	8(28-35)	21	2.04E-27	OVERLAP 8
18237	NNUNUUNDDNDNU	13(20-32)	9	1.17E-26	OVERLAP
15616	NDUNUDND	8(29-36)	20	9.10E-26	OVERLAP 8
24476	DDNUDDNDDNDDU	13(35-47)	8	1.82E-24	OVERLAP 7
33996	UDUDNDUU	8(4-11)	18	2.31E-24	OVERLAP 5
24482	DDNUDDNDDNDDD	13(35-47)	8	2.781E-24	OVERLAP 7
5796	UDNNDUNUDNDDD	13(26-38)	8	4.46E-23	OVERLAP 8
38344	NNUDUNNNU	9(1-9)	15	5.29E-23	MATCH 2
17874	NUNUUNDDNDNUD	13(21-33)	8	1.00E-22	OVERLAP
15613	NDUNUDN	7(29-35)	23	2.12E-21	OVERLAP 8
24818	DDDNUDDNDDNDD	13(34-46)	7	6.69E-21	OVERLAP 7
14213	NDDNUDDNDDNDD	13(34-46)	7	1.17E-20	OVERLAP 7
25027	DNDNDDNNNNDDD	13(33-45)	7	3.12E-20	OVERLAP 7
17048	NNDUNDNUUU	10(24-33)	11	2.90E-23	OVERLAP 9
3997	UDNDNDDNNNNDD	13(32-44)	7	7.07E-20	OVERLAP 7
15157	NUNDNNDD	8(30-37)	17	1.93E-19	OVERLAP 3
5789	UDNNDUN	7(26-32)	22	1.67E-18	OVERLAP 8
5416	UNDNUUU	7(27-33)	21	1.70E-18	OVERLAP 9
5942	UUDNNDUNUDNDDD	13(32-44)	7	1.93E-18	OVERLAP 7
37706	NUDUNNNU	8(2-9)	16	2.23E-18	OVERLAP 2
4182	UNUDNDD	7(31-37)	20	3.04E-18	OVERLAP 8
13600	NDDNNNN	7(36-42)	19	4.07E-18	OVERLAP 10
23014	DNDDNDD	7(40-46)	19	7.71E-18	OVERLAP 7
40141	DUDNDUU	7(5-11)	19	1.03E-17	OVERLAP 5
14145	NDUUDDDD	8(34-41)	14	1.48E-17	MATCH 4 LOST 1 GENE[1]