12 More on Suffix Trees

This week we study the following material:

- WOTD-algorithm
- MUMs
- finding repeats using suffix trees

12.1 The WOTD Algorithm

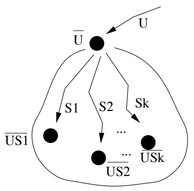
The suffix tree construction algorithm from Sect. 9.8 has the drawback that it needs to construct the *entire* tree before we can start querying. Hence, if we only have a few queries to pose, then it may be wasteful to compute the whole suffix tree.

The "write only, top down" algorithm due to R. Giegerich, S. Kurtz, and J. Stoye (2003) also has good memory locality, and can also be used for a *lazy* construction of the suffix tree, building only as much of the tree as is necessary to satisfy a given query.

Although it requires $O(n \log n)$ average run-time, in practice it is often competitive.

12.1.1 Basic idea: Compute tree recursively

Note that the subtree below a branching node $\overline{u}$ is determined by the set of all suffixes of $T\$$ that start with the prefix u:



So, if we know the set of *remaining suffixes*

$$R(\overline{u}) := \{s \mid us \text{ is a suffix of } T\$\},$$

then we can *evaluate* the node $\overline{u}$, i.e. construct the subtree below $\overline{u}$.

An *unevaluated* node is evaluated as follows: We partition the set $R(\overline{u})$ into groups by the first letter of the strings, i.e. for every letter $c \in \Sigma$, we define the *c-group* as:

$$R_c(\overline{u}) := \{w \in \Sigma^* \mid cw \in R(\overline{u})\}.$$

Consider $R_c(\overline{u})$ for $c \in \Sigma$. If $R_c(\overline{u}) \neq \emptyset$, then there are two possible cases:

1. If $R_c(\overline{u})$ contains precisely one string w, then we construct a new leaf edge starting at $\overline{u}$ and label it with cw.
2. Otherwise, the set $R_c(\overline{u})$ contains at least two different strings and let p denote their *longest common prefix (lcp)*. We create a new *c-edge* with label p whose source node is $\overline{u}$. The new unevaluated node $\overline{up}$ and set $R(\overline{up}) = \{w \mid pw \in R_c(\overline{u})\}$ will be (recursively) processed later.

The *wotd*-algorithm (*write-only, top-down*) starts by evaluating the root node, with $R(root)$ equal to the set of all suffixes of $T\$$. All nodes of $\mathcal{ST}(T)$ are then recursively constructed using the appropriate sets of remaining suffixes in a top-down manner.

12.1.2 Example

Consider as an example the following text:

$$T = \texttt{abab\$}$$

The *wotd*-algorithm proceeds as follows:

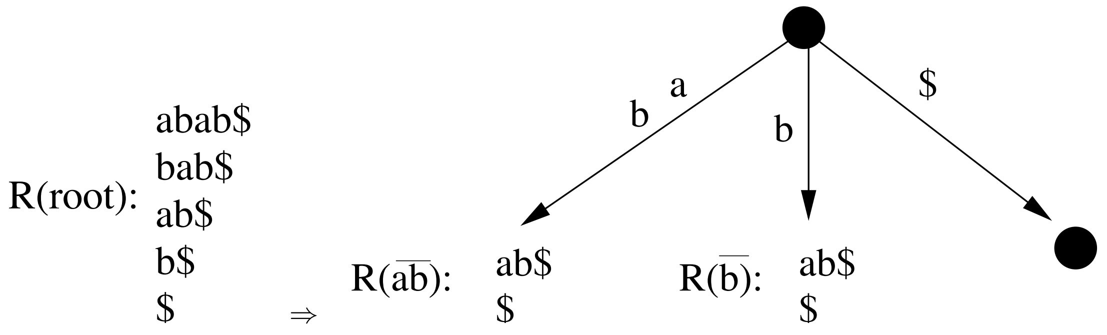
1. Evaluate the root node *root* using

   $$R(root) = \{abab\$, bab\$, ab\$, b\$, \$\}.$$

   There are three groups of suffixes:

   $$R_a(root) = \{abab\$, ab\$\}, R_b(root) = \{bab\$, b\$\} \text{ and } R_\$ = \{\$\}.$$

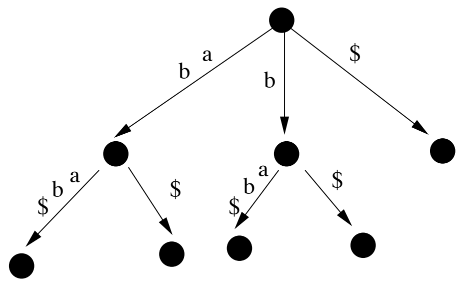
   The letter \$ gives rise to a leaf edge with label \$. The letter a gives rise to an internal edge with label ab, because $ab = lcp(R_a(root))$. Similarly, for b we obtain an internal edge with label b, because $b = lcp(R_b(root))$



2. For the node $\overline{ab}$ we have $R(\overline{ab}) = \{ab\$, \$\}$ and thus $R_a(\overline{ab}) = \{ab\$\}$ and $R_\$(\overline{ab}) = \{\$\}$. Because both latter sets have cardinality one, we obtain two new leaf edges with labels $ab\$$ and \$, respectively.

3. Similarly, for the node $\bar{b}$ we have $R(\bar{b}) = \{ab\$, \$\}$, and thus we obtain two new leaf edges with labels $ab\$$ and \$.

As a result of *wotd* we obtain the following full suffix tree:



12.1.3 Implementation of the suffix tree data-structure

An implementation of a suffix tree must represent its nodes, edges and edge labels. To be able to describe the implementation, we define a total ordering on the set of children of a branching node:

Let $\overline{uv}$ and $\overline{uw}$ be two different children of the same branching node $\overline{u}$ in $\mathcal{ST}(T)$. Recall that we defined $L(\overline{p})$ to be the leaf set under the node $\overline{p}$. We write

$$\overline{uv} \prec \overline{uw} \quad \text{iff} \quad \min L(\overline{uv}) < \min L(\overline{uw}).$$

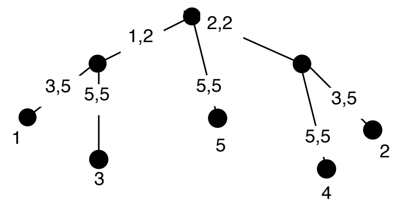
That implies that the first occurrence $\min L(\overline{uv})$ of $\overline{uv}$ in $T\$$ comes before the first occurrence $\min L(\overline{uw})$ of $\overline{uw}$ in $T\$$.

Representing edge labels efficiently

Recall our representation of edge labels:

Remark 1: Because an edge label s is a substring of the text $T\$$, we can represent it by a *pair of pointers* (i, j) into $T' = T\$$ such that $s = t'_i t'_{i+1} \dots t'_j$.

For example, for `abab$` we get



However, note that we have $j = n + 1$ for any leaf edge and so in this case the right pointer is redundant. Hence:

Remark 2: A leaf edge requires only one (left) pointer.

The following is not as easy to see:

Remark 3: An internal edge requires only one pointer *per node*.

This is made possible by defining a *left pointer* on the set of *nodes* (not edges) in such a way that these can be used to reconstruct the original left and right pointers of each edge, as follows:

Consider an edge $\overline{u} \xrightarrow{v} \overline{uv}$. Define the *left pointer lp* of $\overline{uv}$ as the position p of the first occurrence of uv in $T\$$ plus the length of u:

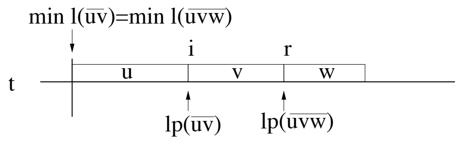
$$lp(\overline{uv}) = \min L(\overline{uv}) + |u|.$$

This gives the start position i of a copy of v in $T\$$.

To get the end position of v, consider the $\prec$-smallest child $\overline{uvw}$ of $\overline{uv}$. We have $\min L(\overline{uv}) = \min L(\overline{uvw})$, i.e. the corresponding suffix starts at the same position p. By definition, we have

$$lp(\overline{uvw}) = \min L(\overline{uvw}) + |uv| = \min L(\overline{uvw}) + |u| + |v| = lp(\overline{uv}) + |v|,$$

and the end position of v equals $lp(\overline{uvw}) - 1$.



The main data table

For each node $\overline{u}$, we store a reference *firstchild*$(\overline{u})$ to its smallest child.

We store the values of *lp* and *firstchild* together in a single (integer) table Tbl. We store the values of all children of a given node $\overline{u}$ consecutively, ordered w.r.t. $\prec$. (We will indicate the last child of $\overline{u}$ by setting its *lastchild*-bit.)

So, only the edge from a given node $\overline{u}$ to its first child is represented explicitly. Edges from $\overline{u}$ to its other children are given implicitly and are found be scanning consecutive positions in Tbl that follow the position of the smallest child.

We reference the node $\overline{u}$ using the index of the position in Tbl that contains the value $lp(\overline{u})$.

Storing an unevaluated node

We consider the *wotd*-algorithm as a process that evaluates the nodes of a suffix tree. It starts at the root and then evaluates all nodes recursively.

First we discuss how to store an unevaluated node $\overline{u}$.

To be able to evaluate $\overline{u}$, we (only) need to know the set of remaining suffixes $R(\overline{u})$. To make these available, we define a global array called *suffixes* that contains pointers to suffixes in $T\$$ and use it as follows:

For every unevaluated node $\overline{u}$, the *suffixes* array contains an interval of pointers to start positions in $T\$$ that correspond precisely to the suffixes contained in $R(\overline{u})$, in increasing order.

We can now represent $R(\overline{u})$ in Tbl using two numbers, $\mathit{left}(\overline{u})$ and $\mathit{right}(\overline{u})$, that define an interval of entries in the *suffixes* array.

If $\overline{u}$ is a branching node, then $\overline{u}$ will occupy two positions in Tbl, one for $lp(\overline{u})$ and followed by $\mathit{firstchild}(\overline{u})$. Until $\overline{u}$ is actually evaluated, we will use these two positions to store $\mathit{left}(\overline{u})$ and $\mathit{right}(\overline{u})$. We use a third bit called the *unevaluated*-bit to distinguish between unevaluated and evaluated nodes.

Evaluating a node $\overline{u}$

First note that the left pointer $lp(\overline{u})$ of the node $\overline{u}$ is given by the left-most entry of *suffixes* over the interval $[\mathit{left}(\overline{u}), \mathit{right}(\overline{u})]$.

Determine the length of the longest common prefix *lcp* of entries in *suffixes* over $[\mathit{left}(\overline{u}), \mathit{right}(\overline{u})]$ add it to all these entries.

The *lcp* is computed by stepping through a simple loop $j = 1, 2 \ldots$ and checking the equality of all letters $t_{\mathit{suffixes}[i]+j}$ for all start positions i in $[\mathit{left}(\overline{u}), \mathit{right}(\overline{u})]$. As soon as a difference is detected, the loop is aborted and j is the length of the *lcp*.

Sort and count all entries of *suffixes* in the interval $[\mathit{left}(\overline{u}), \mathit{right}(\overline{u})]$, by the first letter c of the suffixes as the sort key. (Do this stably, i.e. don't change the order of suffixes that start with the same letter.)

Each letter c that has count > 0 will give rise to a new node $\overline{v}$ below $\overline{u}$ and the suffixes in the c-group $R_c(\overline{u})$ determine the tree below $\overline{v}$.

For each non-empty c-group of $\overline{u}$, we store one child in the table Tbl, as follows:

Leaf case: A c-group containing only one string gives rise to a leaf node $\overline{v}$ and we write the number $lp(\overline{v})$ in the first available position of Tbl. This number $lp(\overline{v})$ is obtained as the single entry of *suffixes* that corresponds to the c-group.

Internal node case: A c-group containing more than one string gives rise to branching node $\overline{v}$ and we store $\mathit{left}(\overline{v})$ and $\mathit{right}(\overline{v})$ in the first two available positions of Tbl. The values of *left* and *right* are computed during the sort and count step.

12.1.4 Lazy vs. complete evaluation

To build the complete suffix tree, we proceed breadth-first, from left to right.

In a *lazy* approach, we only evaluate those nodes that are necessary to answer a query (and have not yet been evaluated).

12.1.5 Example

Input: Text:

Text:	a	b	a	b	\$
	1	2	3	4	5

Initial.: *suffixes*:

1	2	3	4	5

Tbl:

Evaluate(*root*)**:**

Sort and count:
$$\begin{array}{rcll} R_a(root) & = & \{1,3\}, & lcp = ab \\ R_b(root) & = & \{2,4\}, & lcp = b \\ R_\$(root) & = & \{5\} & \end{array}$$

The suffixes are ordered, *left* and *right* are entered in the table and the three bits ($u, *, \dagger$: *unevaluated*, *leaf*, *lastchild*) are set:

suffixes:

1	3	2	4	5

Tbl:

1	2	3	4	5				
u		u		$*\dagger$				

$$\begin{pmatrix} \text{Text}: & a & b & a & b & \$ \\ & 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Evaluate(1)**:**

Note $lp(1) = suffixes[Tbl[1]] = suffixes[1] = 1$ and $firstchild(1) = 6$, thus:

Tbl:

1	**6**	3	4	5				
		u		$*\dagger$				

Determine *lcp* of entries *suffixes*$[1\ldots2]$ (*lcp* =`ab`, thus length=2) and add it to values of *suffixes*$[1\ldots2]$:

suffixes:

3	5	2	4	5

Determine c-groups and then add appropriate nodes:
$$\begin{array}{rcl} R_a(1) & = & \{3\} \\ R_\$(1) & = & \{5\} \end{array}$$

Tbl:

1	**6**	3	4	5	**3**	**5**		
(u)		u		$*\dagger$	$*$	$*\dagger$		

$$\begin{pmatrix} \text{Text}: & a & b & a & b & \$ \\ & 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Evaluate(3)**:**

Note $lp(3) = suffixes[Tbl[3]] = suffixes[3] = 2$ and $firstchild(3) = 8$, thus:

Tbl:

1	6	**2**	**8**	5	3	5		
		u		$*\dagger$	$*$	$*\dagger$		

Determine *lcp* of entries *suffixes*$[3\ldots4]$ (*lcp* =`b`, thus length=1) and add it to values of *suffixes*$[3\ldots4]$:

suffixes:

3	5	3	5	5

Determine c-groups and then add appropriate nodes:
$$\begin{array}{rcl} R_a(3) & = & \{3\} \\ R_\$(3) & = & \{5\} \end{array}$$

Done!

Tbl:

1	6	**2**	**8**	5	3	5	**3**	**5**
		(u)		$*\dagger$	$*$	$*\dagger$	$*$	$*\dagger$

The table *Tbl* for $\mathcal{ST}(abab)$:

node	$\overline{ab}$		$\overline{b}$		$\overline{\$}$	$\overline{abab\$}$	$\overline{ab\$}$	$\overline{bab\$}$	$\overline{b\$}$
Tbl	1	6	2	8	5	3	5	3	5
Index	1	2	3	4	5	6	7	8	9
Bits					$*\,\dagger$	$*$	$*\,\dagger$	$*$	$*\,\dagger$

12.1.6 Finding occurrences in a WOTD tree

Suppose we are given a text *text* and have computed the WOTD tree T. How do we determine all occurrences of a given query string q?

This is done by navigating into the tree and matching the letters of the query with the letters of the edge labels until we have used-up all the letters of the query. Once we have established that the query is contained in the tree, we visit all nodes below the location at which the query was fullfilled and report one occurrence for each leaf.

Recall that the position of an occurrence is not stored in the tree, however, it can be obtained by keeping track of the *depth* of nodes which is the sum of lengths of all edge labels along the path from the root to the node.

12.1.7 Properties of the WOTD-algorithm

- Complexity: Space requirement? Worst case time complexity? (exercises...)

  The expected running time is $O(n \log_k n)$ and experimental studies indicate that the algorithm often performs in linear time for moderate sized strings.

- Good memory locality.

- Algorithm can be parallelized.

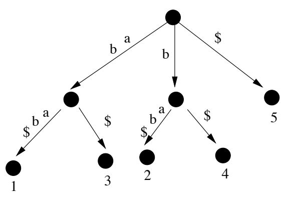
12.2 Applications of Suffix Trees

1. Searching for exact patterns
2. Minimal unique substrings
3. Maximum unique matches
4. Maximum repeats

12.2.1 Searching for exact patterns

To determine whether a string q occurs in a string t, follow the path from the root of the suffix tree $\mathcal{ST}(T)$ as directed by the characters of q. If at some point you cannot proceed, then q does not occur in t, otherwise it does.

Example: Text *abab*\$.



The query *abb* is not contained in *abab*.

(Navigating into the tree and matching the first two letters ab, we arrive at the node $\overline{ab}$, however there is no b-edge leaving from there.)

The query aba is contained in $abab$.

Determining whether q occurs in t requires $O(|q|)$ time.

12.2.2 Finding all occurrences

To find all positions where the query q is contained in t, annotate each leaf $\overline{s_i}$ of the suffix tree with the position i at which the suffix i starts in t.

Then, after matching q to a path in the tree, visit all nodes below the path and return the annotated values.

This works because any occurrence of q in t is the prefix of one of these suffixes.

The number of nodes below the path is at most twice the number of hits and thus finding and collecting all hits takes time $O(|q| + k)$, where k is the number of occurrences.

(Note that in the discussed lazy suffix tree implementation we do not use this leaf annotation but rather compute the positions from the lp values, to save space...)

12.2.3 Minimal Unique Substrings

Definition 12.2.1 (Minimal unique substrings problem) *Assume we are given a sequence $s \in \Sigma^*$, and a number $L > 0$. The* minimal unique substrings *problem consists of enumerating all substrings $u \in \Sigma^*$ of s satisfying the following properties:*

- *u occurs exactly once in s,*
- *$|u| \geq L$, and*
- *all proper prefixes of u occur at least twice in s.*

Example: Let $s =$ `abab` and $L = 2$. Then the minimal unique substrings are `aba` and `ba`. Note, that `bab` is not a minimal unique substring, since the proper prefix `ba` of `bab` is already unique, i.e. the minimality condition does not hold.

Question: How can suffix trees be used to solve the minimal unique substring problem?

Applications of this in "primer design".

12.2.4 Application: Maximum Unique Matches

Modern sequencing and computational technologies and advances in bioinformatics has made whole genome sequencing possible. One resulting challenge is the fast alignment of whole genomes.

Dynamic programming is too slow for aligning two large genomes.

Heuristics such as BLAST or FASTA are not designed to perform pairwise alignments of two very long sequences.

One very successful approach is based on identifying "maximal unique matches", which is based on the assumption that one expects to substrings occurring in two similar genomes.

Maximum unique matches (MUMs) are almost surely part of a good alignment of the two sequences and so the alignment problem can be reduced to aligning the sequence in the gaps between the MUMs.

Definition 12.2.2 (MUM problem) *Assume we are given two sequences $s, t \in \Sigma^*$, and a number $L > 0$. The* maximal unique matches *problem (MUM-problem) is to find all sequences $u \in \Sigma^*$ with:*

- $|u| \geq L$,
- *u occurs exactly once in s and once in t, and*
- *for any character $a \in \Sigma$ neither ua nor au occurs both in s and t.*

In other words, a MUM is a sequence u that occurs precisely once in s and once in t, and is both *right maximal* and *left maximal* with this property (meaning that ua and au both do not have the uniqueness property, for any letter a).

For example: if

$s =$`mostbeautifulandwildcorsica`

$t =$`greencleanandnuclearfree`,

then there is only one MUM of length ≥ 3, namely

`and`

This problem can be solved in $O(|s| + |t|)$ time using a so-called *generalized* suffix tree:

To find all MUMs, generate the *generalized* suffix tree T for $s\%t\$$, where % is a "separator" with $\% \notin s$ and $\% \notin t$. Any path in T from the root to some node $\overline{u}$ that has precisely two children, one in s and one in t, corresponds to a right maximal unique match.

To determine whether u is left maximal, too, simply check whether both preceding letters in s and t differ.

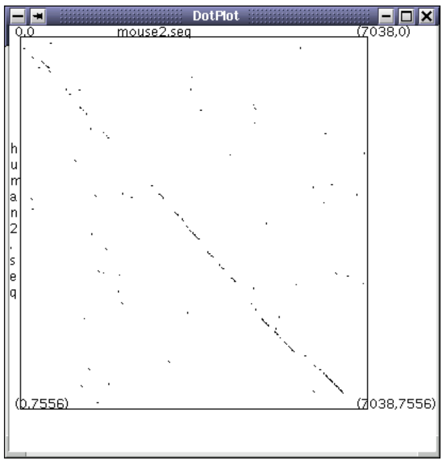
Example

For $s =$`aggac` and $t =$`agagcgac`, construct the suffix tree for $s\%t\$$:

The node $\overline{\texttt{gac}}$ has precisely two leaves as childern, one representing a suffix that starts in s and the other one that starts in t. Thus, the word `gac` occurs precisely once in both s and t, and is right maximal. The word is also left-maximal as the preceding characters in s (=`g`) and t (=`c`) differ.

Note, however, that the string `ag` is not a MUM, since the node $\overline{ag}$ has three children.

MUMs are apparent in dotplots:



Example

2376 Nucleic Acids Research, 1999, Vol. 27, No. 11

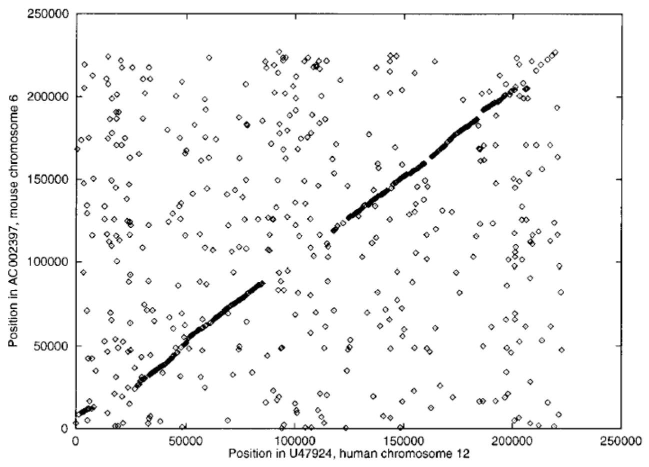
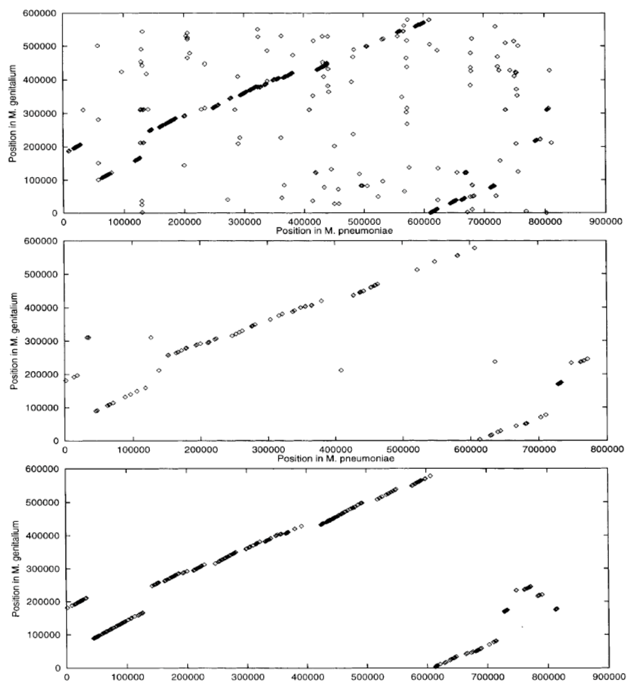


Figure 8. Alignment of a 222 930 bp subsequence of human chromosome 12p13, accession no. U47924, to a 227 538 bp subsequence of mouse chromosome 6, accession no. AC002397. Each point in the plot corresponds to an MUM of ≥15 bp.

Example



12.2.5 Applications

Applications of MUMs in sequence analysis:

- Detection of large-scale inversions in bacterial genomes
- Detection of whole-genome duplications, for example in the genome of *Arabidopsis thaliana.*
- Comparison of different assemblies of the same genome at different stages of sequencing and finishing.
- Comparison of assemblies of the same data using different assembly algorithms.

12.2.6 Detecting Repeats

A puzzling observation in the early days of molecular biology was that genome size does not correlate well with "organismal complexity". For example, *Homo sapiens* has a genome that is 200 times as large as that of the yeast *S. cerevisiae*, but 100 times as small as that of *Amoeba dubia.*

This so-called *C-value paradox* was largely resolved with the recognition that many genomes consist of a large amount of repetitive sequence.

Repeats in human

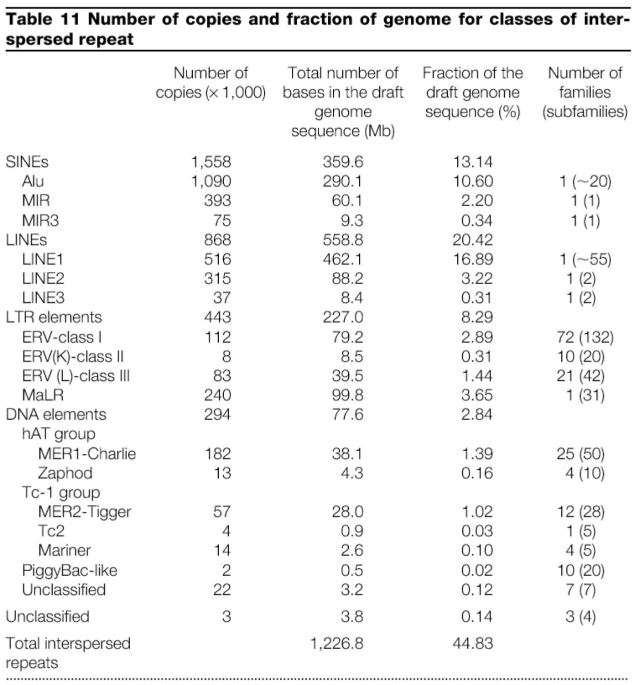


Table 11 Number of copies and fraction of genome for classes of interspersed repeat

	Number of copies (× 1,000)	Total number of bases in the draft genome sequence (Mb)	Fraction of the draft genome sequence (%)	Number of families (subfamilies)
SINEs	1,558	359.6	13.14	
Alu	1,090	290.1	10.60	1 (~20)
MIR	393	60.1	2.20	1 (1)
MIR3	75	9.3	0.34	1 (1)
LINEs	868	558.8	20.42	
LINE1	516	462.1	16.89	1 (~55)
LINE2	315	88.2	3.22	1 (2)
LINE3	37	8.4	0.31	1 (2)
LTR elements	443	227.0	8.29	
ERV-class I	112	79.2	2.89	72 (132)
ERV(K)-class II	8	8.5	0.31	10 (20)
ERV (L)-class III	83	39.5	1.44	21 (42)
MaLR	240	99.8	3.65	1 (31)
DNA elements	294	77.6	2.84	
hAT group				
MER1-Charlie	182	38.1	1.39	25 (50)
Zaphod	13	4.3	0.16	4 (10)
Tc-1 group				
MER2-Tigger	57	28.0	1.02	12 (28)
Tc2	4	0.9	0.03	1 (5)
Mariner	14	2.6	0.10	4 (5)
PiggyBac-like	2	0.5	0.02	10 (20)
Unclassified	22	3.2	0.12	7 (7)
Unclassified	3	3.8	0.14	3 (4)
Total interspersed repeats		1,226.8	44.83	

(Nature, vol. 409, pg. 880, 15. Feb 2000)

12.2.7 Repeats

A PubMed query for

```
(repeat OR repetitive) AND (protein OR nucleotide sequence)
```

on May 13, 2006 produced the following results:

- 62017 hits without limits to specific fields
- 18472 hits when restricting to title/abstract
- 1170 hits when restricting to title

Keywords associated with the hits:

`imprinting / RNA editing / diseases / repair / genome organization / viruses / protein families`

One can distinguish between

- local, small-scale repeats with "known" function or origin,
- simple repeats, local and interspersed, with "less known" function, and
- complex interspersed repeats with "unknown" function.

Examples of *local repeats* are:

- palindromic sequences (regulation of DNA transcription), eg. restriction enzyme recognition sequences,
- inverted repeats flanking transposons (orientation),

- repeats in viruses ,
- non-coding RNAs such as rRNAs and tRNAs (must be produced in large copy numbers),
- protein family members (eg. globins),
- clustered genes (eg. histones), and
- motifs and domains of proteins.

Examples of *simple repeats* are:

- often seen as tandem repeats: eg. `TTAGGG` up to several copies at the end of every human chromosome,
- simple tandem repeats of length 3 nucleotides are linked to certain diseases (Huntington, etc.), and
- satellite DNA (forensic applications).

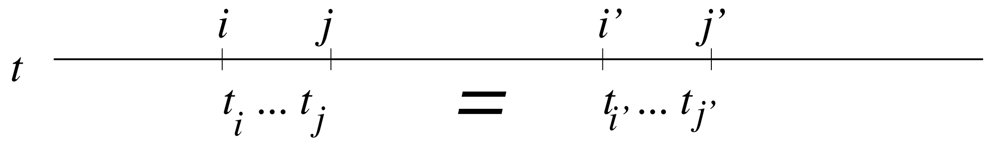
Examples of *interspersed repeats* are:

- SINEs - Short Interspersed Nuclear sEquences (Alu repeats, MIRs),
- LINEs, and
- LTR elements.

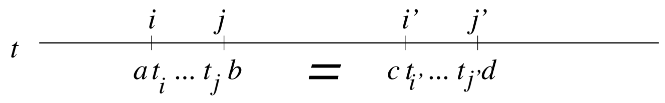
12.2.8 Maximum Repeats

Definition 12.2.3 (Repeat) *Assume we are given a sequence $t = t_1 t_2 \dots t_n$.*

Let a substring $t[i,j] := t_i \dots t_j$ be represented by the pair (i,j). A pair $R = (l,r)$ of different substrings $l = (i,j)$ and $r = (i',j')$ of t is called a repeat, *if $i < i'$ and $t_i \dots t_j = t_{i'} \dots t_{j'}$. We call l and r the* left *and* right instance *of the repeat R, respectively.*



Definition 12.2.4 (Maximal repeat) *A repeat $R = ((i,j),(i',j'))$ is called* left maximal, *if $i = 1$ or $t_{i-1} \neq t_{i'-1}$, and* right maximal, *if $j' = n$ or $t_{j+1} \neq t_{j'+1}$, and* maximal, *if it is both left and right maximal.*



maximum $\Leftrightarrow a \neq c$ and $b \neq d$

Note that the definition allows for overlapping repeats.

Example

	1	2	3	4	5	6	7	8	9	10	
The string	g	a	g	c	t	c	g	a	g	c	contains the following repeats of length ≥ 2:

$((1,4),(7,10))$	$gagc$	$\Rightarrow$	maximal
$((1,3),(7,9))$	gag	$\Rightarrow$	left maximal
$((2,4),(8,10))$	agc	$\Rightarrow$	right maximal
$((1,2),(7,8))$	ga	$\Rightarrow$	left maximal
$((2,3),(8,9))$	ag		
$((3,4),(9,10))$	gc	$\Rightarrow$	right maximal

Computing all Maximum Repeats

We will discuss how to compute all maximal repeats for a string t of length n. Generally we also want to permit a prefix or a suffix of t to be part of a maximal repeat. To model this we simply add a character to the start of t and one to the last of t that do no occur elsewhere in t, e.g.:

	1	2	3	4	5	6	7	8	9	10	11	12	13
$t =$	x	g	g	c	g	c	y	g	c	g	c	c	z

Let $\mathcal{ST}(T)$ be the suffix tree for t. A connection between suffix trees and maximal repeats is stated in the following

Lemma 12.2.5 *Let $\mathcal{ST}(T)$ be a suffix tree for string t. If a string r is a maximal repeat in t, then r is the label of an internal node $\overline{r}$ in $\mathcal{ST}(T)$.*

The following is maybe a bit surprising

Theorem 12.2.6 *There are at most n maximal repeated strings in any string of length n.*

Proof: Since $\mathcal{ST}(T)$ has n leaves, it has at most n internal nodes. Thus the theorem follows immediately from the preceding lemma. □

Thus because of the theorem above we will need only to look at internal nodes, but which specific internal nodes correspond to maximal repeats? The following algorithm will answer this question.

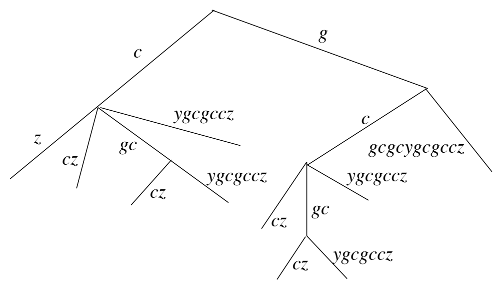
From now on we can ignore all leaf edges from the root (why?).

The algorithm proceeds in two phases:

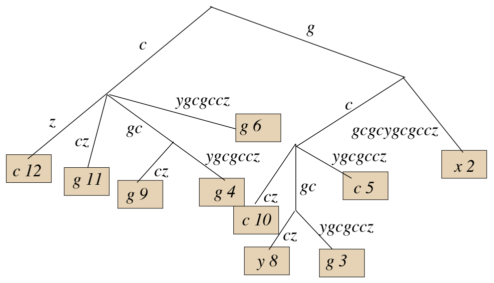
In the first phase, every leaf node $\overline{v}$ of $\mathcal{ST}(T)$ is annotated by (a, i), where i denotes the position of the suffix $v = t_i \dots t_n$ associated with $\overline{v}$ and $a = t_{i-1}$ is the letter that occurs immediately before the suffix.

Example:

		1	2	3	4	5	6	7	8	9	10	11	12	13	
Partial suffix tree for	$t =$	x	g	g	c	g	c	y	g	c	g	c	c	z	:



With leaf annotations:



Note that a substring r can only be a maximal repeat iff its branching node $\overline{r}$ has at least two leaves with different letters in their annotations. Thus in order to find and print those we need to combine the leaf annotations appropriately.

For every leaf node $\overline{v}$ and $c \in \Sigma$ we set:

$$A(\overline{v}, c) = \begin{cases} \{i\}, & \text{if } c = t_{i-1}, \text{ and} \\ \emptyset, & \text{else,} \end{cases}$$

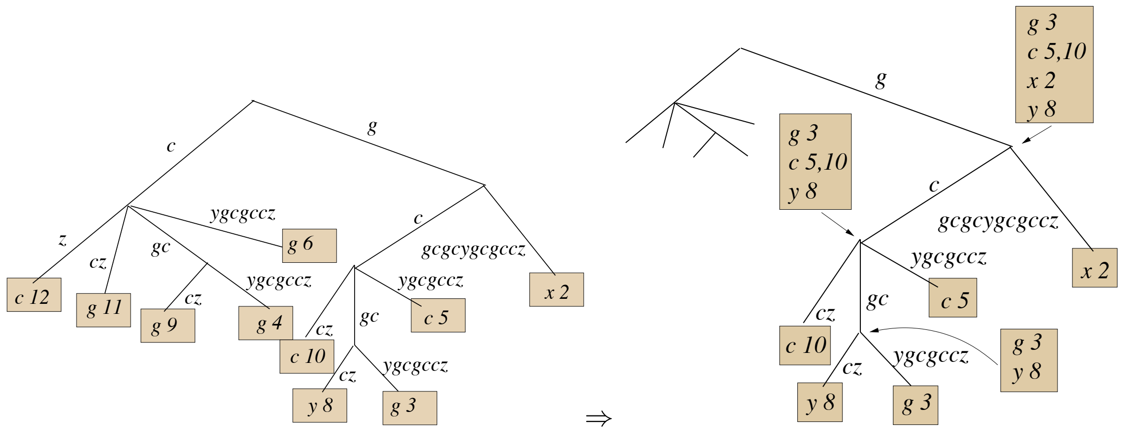
where i is the start position of the corresponding suffix v.

In the second phase of the algorithm, we extend this annotation to all branching nodes, bottom-up:

Let $\overline{w}$ be a branching node with children $\overline{v_1}, \ldots, \overline{v_h}$ and assume we have computed $A(\overline{v_j}, c)$ for all $j \in \{1, \ldots, h\}$ and all $c \in \Sigma$. For each letter $c \in \Sigma$ set:

$$A(\overline{w}, c) := \bigcup_{j=1}^{h} A(\overline{v_j}, c).$$

Note that this is a disjoint union and $A(\overline{w}, c)$ is the set of all start positions of w in t for which $t_{i-1} = c$.

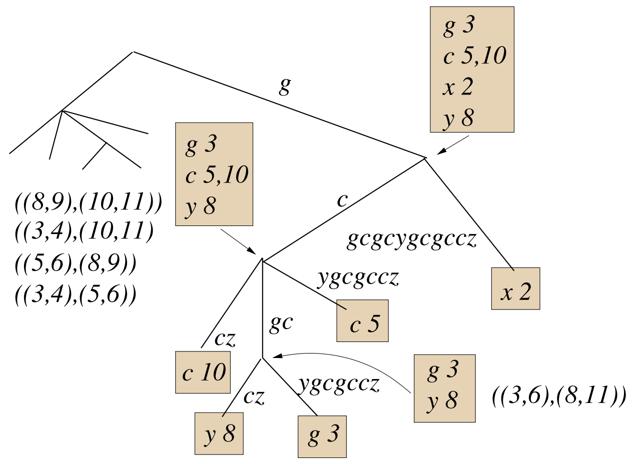


Reporting All Maximum Repeats

In a bottom-up traversal, for each branching node $\overline{w}$ we first determine $A(\overline{w}, c)$ for all $c \in \Sigma$ and then report all maximal repeats of the substring w that have length $\geq L$:

Let q be the string-depth of node $\overline{w}$, i.e., q is equal to the length of w. Then there is a maximal repeat for w, if $\overline{w}$ has a pair of children $\overline{v_f}$ and $\overline{v_g}$ such that there is a letter c and a letter d, with $c \neq d$ and $A(\overline{v_f}, c) \neq \emptyset$ and $A(\overline{v_g}, d) \neq \emptyset$. In this case we output $R((i, i+q-1), (j, j+q-1))$ for all i that are in $A(\overline{v_f}, c)$ and j that are in $A(\overline{v_g}, d)$.

The result of the algorithm for maximal repeats of length $L \geq 2$ for the example is:



The following summarizes the algorithm as pseudo code:

```
for each internal node $\overline{w}$ of string-depth $q \geq L$ do
    for each pair of children $\overline{v_f}$ and $\overline{v_g}$ of $\overline{w}$ with $\overline{v_f} \prec \overline{v_g}$ do
        for each letter $c \in \Sigma$ with $A(\overline{v_f}, c) \neq \emptyset$ do
            for each $i \in A(\overline{v_f}, c)$ do
                for each letter $d \in \Sigma$ with $d \neq c$ and $A(\overline{v_g}, d) \neq \emptyset$ do
                    for each $j \in A(\overline{v_g}, d)$ do
                        Print $((i, i+q-1), (j, j+q-1))$
end
```

Maximality of output

We need to prove that the algorithm computes *all* maximal repeats.

Lemma 12.2.7 *The algorithm prints precisely the set of all maximal repeats in t of length $\geq L$.*

Proof:

1. Each right maximal repeat R is represented by an internal node in the tree.
2. Vice versa, each reported repeat R is right-maximal.
3. Each reported repeat R is left-maximal, as $c \neq d$ in the above algorithm.
4. No maximal repeat is reported twice, as $\overline{v_f} \prec \overline{v_g}$ and all unions are disjoint. □

Performance analysis

The following result states that the maximal repeats algorithm is both time and space optimal.

Lemma 12.2.8 *Computation of all maximal repeats of length $\geq L$ can be done in $O(n+z)$ time and $O(n)$ space, where z is the number of maximal repeats.*

Proof: The suffix tree can be built in $O(n)$ time and space. We can annotate the tree in $O(n)$ time and space, if we use the fact that we only need to keep the annotation of a node until its father has been fully processed. (Also, we maintain the sets as linked links and then each disjoint-union operation can be done in constant time.) In the nested loop we enumerate in total all z maximal repeats in $O(z)$ steps. □

12.2.9 Bioinformatics Software for Repeats

- Suffix trees: Reputer - The Repeats Computer (http://www.genomes.de/)
- RepeatMasker: Repeat detection not based on suffix trees.

  RepeatMasker (http://www.repeatmasker.org/) is a program that screens DNA sequences for interspersed repeats and low complexity DNA sequences. The output of the program is a detailed annotation of the repeats that are present in the query sequence as well as a modified version of the query sequence in which all the annotated repeats have been masked (default: replaced by Ns). On average, almost 50% of a human genomic DNA sequence currently will be masked by the program. Sequence comparisons in RepeatMasker are performed by the program cross_match, an efficient implementation of the Smith-Waterman-Gotoh algorithm.

12.3 Summary

Applications of suffix trees in bioinformatics range from the search for many queries in a large text, the computation of minimal unique matches, MUMs, and to the detection of exact repeats.

A naive algorithm to build a suffix tree requires $O(n^2)$ time, with n being the length of the text. A query for an exact pattern q in the tree requires $O(|q|)$ time.

We can build a suffix tree in time that is linear in the size of the text, while the WOTD algorithm requires $O(n \log n)$ time to build, but can be constructed lazily.