4.1 Lazy suffix trees

This exposition has been developed by Knut Reinert based on slides from David Weese. It is based on the following sources, which are all recommended reading:

1. Giegerich, Kurtz, Stoye (2003) *Efficient implementation of lazy suffix trees*, Softw. Pract. Exper. 2003; 33:1035-1049 (DOI: 10.1002/spe.535)
2. David Weese (2013) *Indices and Applications in High-Throughput Sequencing*, PhD thesis

4.2 The main idea

A lazy suffix tree is a suffix tree whose nodes are created on demand, i.e. only when they are visited during a traversal in a top-down fashion. Hence the suffix tree construction is deferred to the traversal and only the necessary parts for the traversal or a search query are build.

This means, that there is essentially no preprocessing time. Instead, the preprocessing time is amortized and distributed over the accessions. In the following we assume that the reader is familiar with a suffix tree.

4.3 The WOTD algorithm

We consider a given non-empty text s of length n and a rooted, directed tree T that in every state of the algorithm is a subgraph of the suffix tree including its root, in the following referred to as *partial suffix tree*. Let R be a function that maps any string $\alpha \in \Sigma^*$ to the set of suffixes of $s\$$ that begin with α:

$$R(\alpha) := \left\{ \alpha\beta \;\middle|\; \alpha\beta \text{ is a suffix of } s\$ \right\} \setminus \{\$\}\,. \tag{4.1}$$

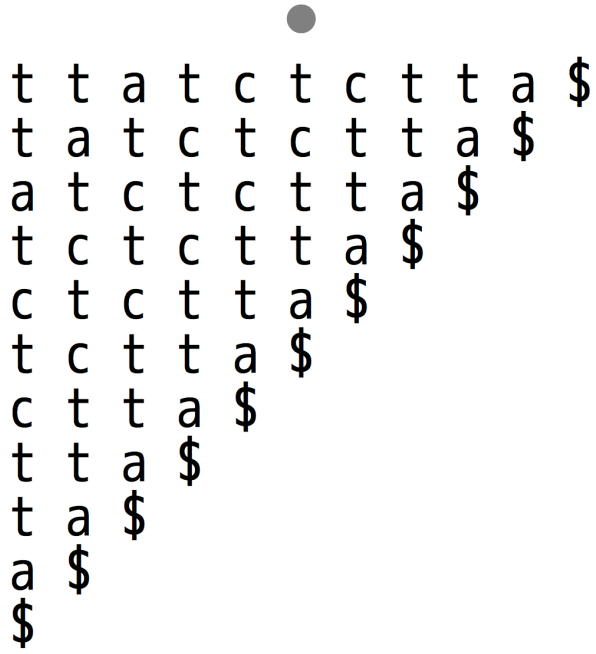
Given a branching suffix tree node $\overline{\alpha}$ (we denote for a string α it corresponding node in the suffix tree with $\overline{\alpha}$, if existent), $R(\alpha)$ contains the concatenation strings of the leaves below $\overline{\alpha}$.

```
// wotd_eager
input is a partially constructed suffix tree T and node ᾱ;
divide R(α) into subsets R(αc) of suffixes where character c
follows the α-prefix;
foreach c∈ Σ ∪ {$} and R(αc) ≠ ∅ do
  αcβ ← lcp(R(αc));
  if (| R(αc) |= 1)
    then
          add leaf $\overline{\alpha c\beta}$ as a child of ᾱ in T;
    else
          add inner node $\overline{\alpha c\beta}$ as a child of ᾱ in T;
          WOTDEAGER(()T, $\overline{\alpha c\beta}$);
  fi
od
```

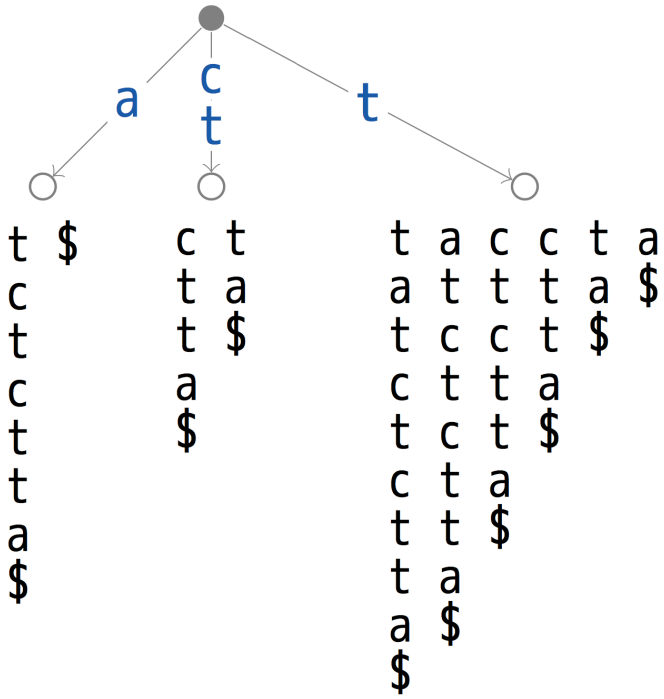
It is easy to see that the suffix tree can be constructed by calling WOTD_EAGER(()$T, \overline{\epsilon}$). However, this defeats the purpose and more efficient algorithms exist to construct a suffix tree (i.e. by Ukkonen).

A key property of the wotd-algorithm is that it constructs the suffix tree top-down and nodes from disjunctive subtrees can be expanded independently and in arbitrary order. That makes it possible to step-by-step expand single nodes instead of entire subtrees and allows turning the suffix tree construction into a lazy, on-demand construction.

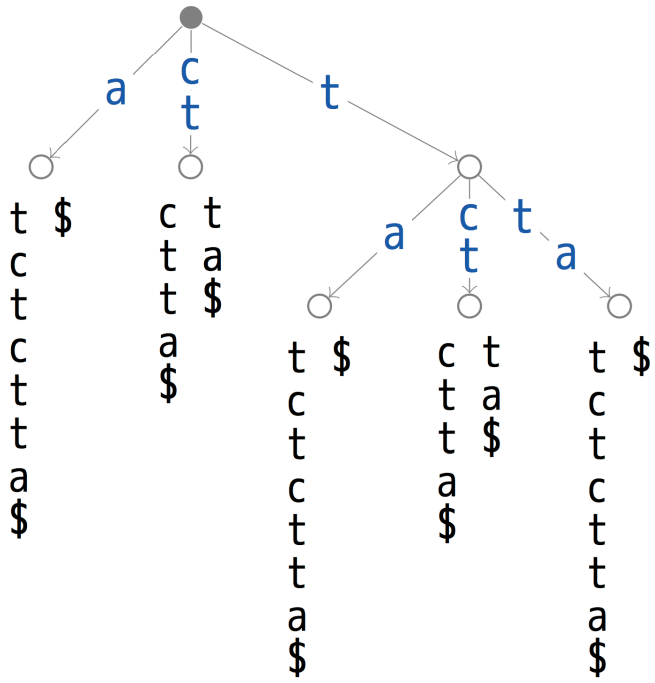
Nevertheless. Let construct it completely for $s = ttatctctta$.



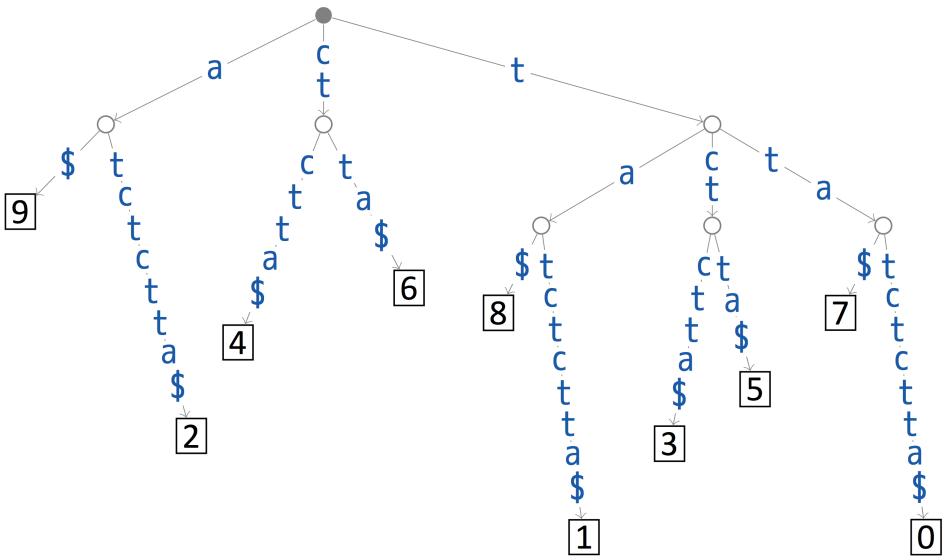
Initial state of the lazy suffix tree for $s = ttatctcta$.



Expanded root.



Expanded node $\bar{t}$.



Fully expanded tree.

How do we implement now the lazy suffix tree without using a tree like pointer structure?

We require a method to expand a suffix tree node and a data structure to represent a partial suffix tree whose nodes are either in *expanded* or *unexpanded* state. Further, it requires $R(\alpha)$ for the expansion of nodes $\overline{\alpha}$ and needs to provide the corresponding set of suffix start positions for all (even expanded) nodes $\overline{\alpha}$:

$$l(\alpha) := \left\{ i \in [0..n) \;\middle|\; \exists \beta \in \Sigma^* : \; s_i\$ = \alpha\beta \right\}$$

In the original lazy suffix tree construction the children $\overline{\alpha\beta}$ of an inner node $\overline{\alpha}$ are not in lexicographically order, instead they are ordered increasingly by $\min l(\alpha\beta)$, i. e. decreasingly by the length of the longest suffix in $R(\alpha\beta)$.

This order is well defined as the children $\overline{\alpha\beta_1}, \overline{\alpha\beta_2}, \ldots, \overline{\alpha\beta_m}$ of $\overline{\alpha}$ partition the set $l(\alpha)$ into non-empty, disjoint sets $l(\alpha\beta_1), l(\alpha\beta_2), \ldots, l(\alpha\beta_m)$.

4.4 Represent edge labels

Consider an edge from the expanded node $\overline{\alpha}$ to a child $\overline{\alpha\beta}$. As $l(\alpha\beta)$ is the set of occurrence begin positions of $\alpha\beta$, it holds that $\beta = s[\min l(\alpha\beta) + |\alpha|.. \min l(\alpha\beta) + |\alpha\beta|]$. Let lp be a function on tree nodes defined as:

$$lp(\overline{\alpha\beta}) := \min l(\alpha\beta) + |\alpha|, \quad \text{where } \overline{\alpha} \text{ is parent of } \overline{\alpha\beta}. \tag{4.2}$$

We then can substitute $\beta = s[lp(\overline{\alpha\beta})..lp(\overline{\alpha\beta}) + |\beta|)$. For now, assume that the tree representation stores lp-values for all children of an expanded node.

It remains to show how to determine $|\beta|$. In case $\overline{\alpha\beta}$ is a leaf in the suffix tree, $\alpha\beta$ is a suffix of $s\$$ and it holds $|\beta| = n + 1 - lp(\overline{\alpha\beta})$.

Otherwise, assume that $\overline{\alpha\beta}$ is expanded and let $\overline{\alpha\beta\gamma_1}$ be the first child of $\overline{\alpha\beta}$. By definition of the child order it holds $\min l(\alpha\beta) = \min l(\alpha\beta\gamma_1)$ and hence $|\beta| = lp(\overline{\alpha\beta\gamma_1}) - lp(\overline{\alpha\beta})$. If $\overline{\alpha\beta}$ is not expanded, $|\beta|$ can be computed via $|\beta| = |lcp(R(\alpha\beta))| - |\alpha|$.

4.5 Storing nodes

The nodes of the partial suffix tree are stored in a string T of integers, where the children of a node are stored in a *contiguous* block and in the same order as in the tree. Expanded and unexpanded nodes are treated differently.

An *expanded* inner node $\overline{\alpha}$ is represented by two adjacent entries, $lp(\overline{\alpha})$ and $firstchild(\overline{\alpha})$ (i.e. two integers per internal node).

The latter refers to the beginning of the block of child nodes in T. A leaf is represented by a single entry in T the value $lp(\overline{\alpha})$ (one integer per leaf).

To distinguish between inner nodes and leaves, a *leaf bit* (L) is split off the first entry. The last child of a node is marked by a *last-child bit* (LC) in the first entry.

Unexpanded nodes are marked by an *unexpanded bit* (U) in the second entry. To expand a node $\overline{\alpha}$, the suffixes $R(\alpha)$ are partitioned according to their character at position $|\alpha|$.

To this end, we store the corresponding suffix start positions in an additional integer string $suftab$ of length n initialized with $0, 1, \ldots, n-1$.

In T the two reserved entries of every unexpanded node store boundaries i, j of substrings of $suftab$ such that the following invariant holds:

The intervals $[i..j)$ are disjoint subsets of $[0..n)$ and for an unexpanded node $\overline{\alpha\beta}$, $suftab[i..j)$ contains the values $l(\alpha\beta) + |\alpha|$ in increasing order. Hence the lp-value of $\overline{\alpha\beta}$ equals $suftab[i]$ and is therefore *also available for unexpanded nodes*.

4.6 Stepwise expansion

In the beginning, the partial suffix tree consists of only the root node represented by two entries $0, n$ in T with *leaf* and *unexpanded bits* set.

Before expanding $\overline{\alpha\beta}$, the length of $|\beta|$ is unknown and must be determined by computing the lcp value $lcp(\{s_k \mid k \in suftab[i..j)\})$. This can be done by step-wise comparing all characters $s[suftab[i..j) + l]$ for $l = 1, 2, \ldots$ for equality. Then $|\beta|$ equals the smallest value l for which $a, b \in [i..j)$ exist with $s[suftab[a] + l] \neq s[suftab[b] + l]$.

The values $suftab[i..j)$ are then *increased* by $|\beta|$ and stably rearranged into subintervals G_x of the same character x, such that for $x \in \Sigma$ holds $\forall_{k \in G_x} s[suftab[k]] = x$ and $\dot{\bigcup}_{x \in \Sigma \cup \{\$\}} G_x = [i..j)$.

The groups correspond to the children of $\overline{\alpha\beta}$ and are appended to T in lp-order, i. e. increasingly by the value $\min_{k \in G_x} suftab[k]$.

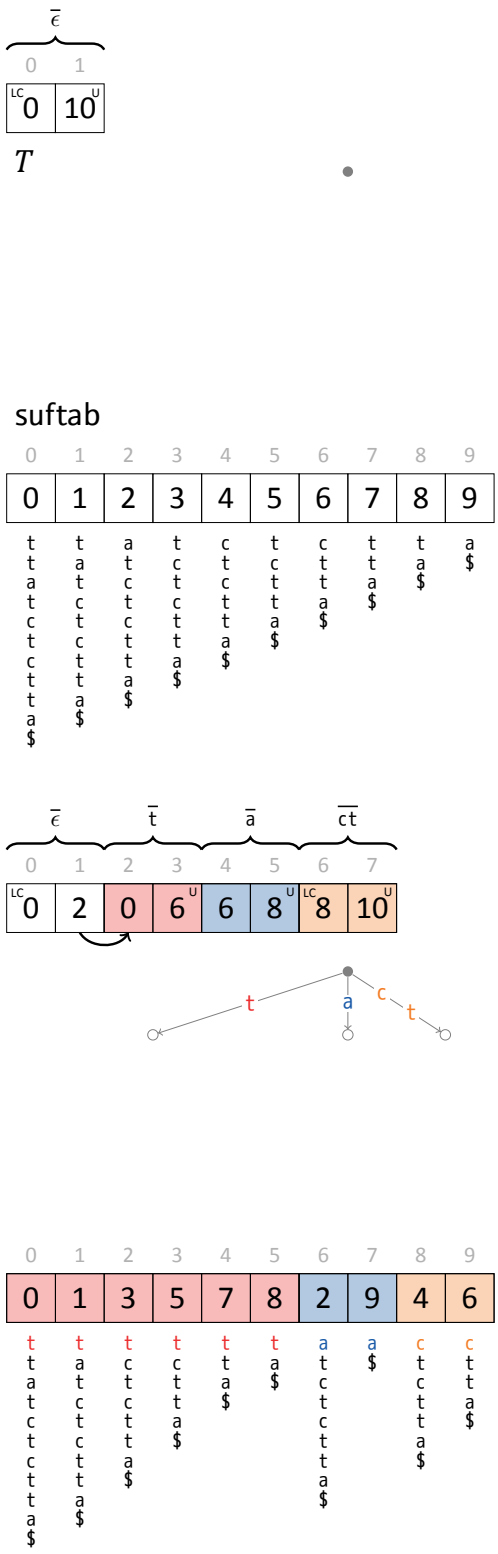
For each singleton group, a single lp-entry with a set *leaf bit* is stored.

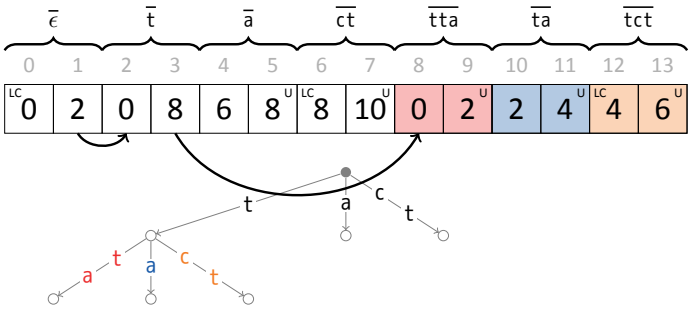
The remaining groups are branching nodes whose subinterval boundaries are stored and marked with a set *unexpanded bit*. The last group is marked by setting the *last-child bit*.

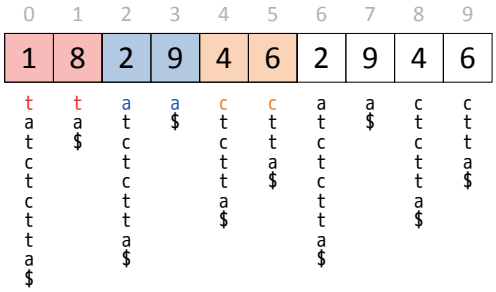
Finally, the *unexpanded bit* of the parent node is cleared and the two interval boundaries are replaced by $lp(\overline{\alpha\beta})$ and $firstchild(\overline{\alpha\beta})$, the position of the first child group appended to T.

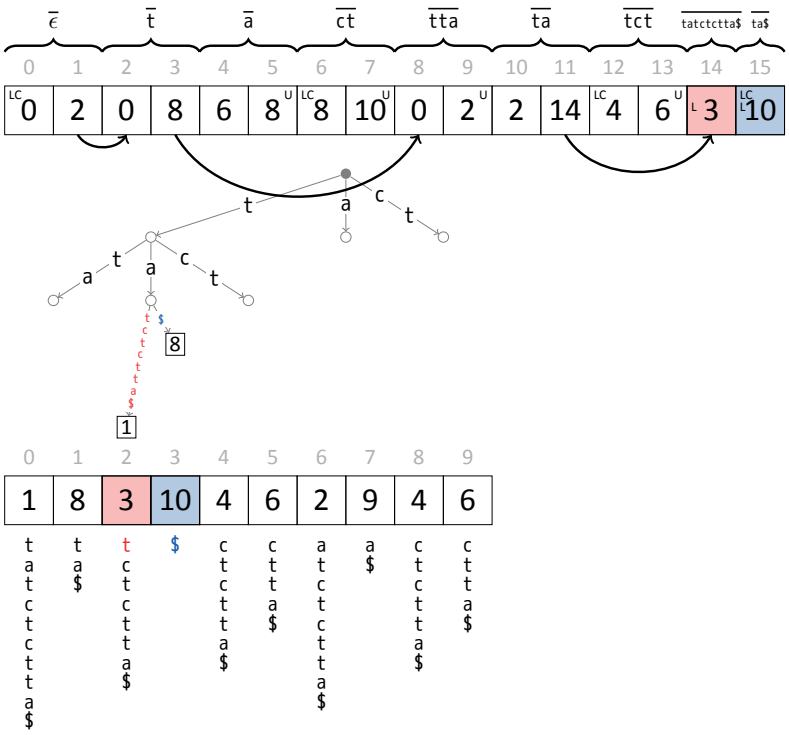
In the following we look at the stepwise construction of the lazy suffix tree for $s = ttatctctta$.

Note the ordering of the children according to $\min l(\alpha\beta)$. And note how the two entries per node are used differently depending on whether they represent an expanded or unexpanded node.









4.7 Analysis

The theoretical running time for constructing the whole suffix tree is $O(n^2)$ for the worst case (why n^2?) and $O(n \log_{|\Sigma|} n)$ on average.

In practice, the algorithm shows almost a linear running time and benefits from its good cache locality during the recursive descent. However, for repetitive strings the run time deteriorates.

The representation as described above requires $2q + n$ integers, where q is the number of non-root branching nodes. Since $q = n - 1$ in the worst case, this is an improvement of $2n$ integers over the best previous representation.

4.8 Experiments

Giegerich et al. tested an implementation of the lazy suffix tree.

In a first experiment they ran three different programs constructing suffix trees: wotdeager, mccl, and mcch. The latter two implement McCreight's suffix tree construction. mccl computes the improved linked list representation, and mcch computes the improved hash table representation of the suffix tree.

The following table shows the running times and the space requirements. They normalized w.r.t. the length of the files. That is, they show the relative time (in seconds) to process 10^6 characters (i.e., rtime = $(10^6 \cdot time)/n$), and the relative space requirement in bytes per input character. For wotdeager they show the space requirement for the suffix tree representation (stspace),a s well as the total space requirement including the working space.

Table I. Time and space requirement for different programs constructing suffix trees.

			wotdeager			*mccl*		*mcch*	
File	n	k	*rtime*	*stspace*	*space*	*rtime*	*space*	*rtime*	*space*
bib	111 261	81	**0.27**	8.30	**9.17**	0.54	9.61	0.81	14.54
book1	768 771	82	**0.72**	8.01	**9.09**	1.61	10.00	1.20	14.90
book2	610 856	96	**0.64**	8.25	**9.17**	1.18	10.00	1.05	14.53
alice29	152 089	74	**0.33**	8.25	**9.43**	0.72	10.01	0.92	14.54
lcet10	426 754	84	**0.56**	8.25	**9.24**	1.12	10.00	1.01	14.53
plrabn12	481 861	81	**0.62**	7.94	**8.93**	1.39	10.00	1.14	14.91
E.coli	4 638 690	4	1.83	9.14	**10.47**	**1.53**	12.80	1.77	17.29
bible	4 047 392	63	1.43	8.43	**9.57**	1.42	10.00	**1.28**	14.53
world	2 473 400	94	1.15	8.33	**9.34**	1.18	9.60	**1.04**	14.54
Average			0.84	8.32	9.38	1.19	10.22	1.13	14.92

In a second experiment they studied the behavior of different programs searching for many exact patterns in an input string. For the programs of the previous experiment, and for wotdlazy, they implemented search functions. wotdeager and mccl require $O(km)$ time to search for a pattern string of length m (k is the alphabet size). mcch requires $O(m)$ time.

Since the pattern search for wotdlazy is merged with the evaluation of suffix tree nodes, one cannot make a general statement about the running time of the search. They also considered suffix arrays, using the original program code developed by Manber and Myers. The suffix array program, referred to by mamy, constructs a suffix array in $O(n \cdot \log n)$ time.

Searching is performed in $O(m + \log n)$ time. The suffix array requires $5n$ bytes of space. For the construction, additionally $4n$ bytes of working space are required. Finally, they also considered the iterated application of an on-line string searching algorithm, their own implementation of the Boyer-Moore-Horspool algorithm, referred to by bmh. The algorithm takes $O(n + m)$ expected time per search, and uses $O(m)$ working space.

They generated patterns according to the following strategy: for each input string t of length n they randomly sampled ρn substrings $s_1, s_2, \ldots, s_{\rho_n}$ of different lengths from t and tested if they occurred in t or not. The proportionality factor ρ was between 0.0001 and 1. The lengths of the substrings were evenly distributed over the interval $[10, 20]$.

For $i \in [1, \rho_n]$, the programs were called to search for pattern p_i , where $p_i = s_i$, if i is even, and p_i is the reverse of s_i , otherwise (to simulate unsuccessful searches). The next table shows the relative running times for $\rho = 0.01$.

For wotdlazy they show the space requirement for the suffix tree after all ρn pattern searches have been performed (stspace), and the total space requirement.

Table II. Time and space requirement for searching $0.01n$ exact patterns.

			wotdlazy						*mamy*		
File	*n*	*k*	*rtime*	*stspace*	*space*	*wotdeager rtime*	*mccl rtime*	*mcch rtime*	*rtime*	*space*	*bmh rtime*
bib	111 261	81	**0.18**	0.93	**5.24**	0.36	0.54	0.72	4.22	9.00	0.81
book1	768 771	82	**0.61**	0.90	**5.22**	0.75	1.70	1.21	6.57	9.00	11.24
book2	610 856	96	**0.51**	0.91	**5.22**	0.64	1.26	1.06	6.78	9.00	8.46
alice29	152 089	74	**0.20**	0.93	**5.23**	0.33	0.79	0.92	5.26	9.00	1.32
lcet10	426 754	84	**0.42**	0.88	**5.22**	0.59	1.17	1.05	6.51	9.00	5.67
plrabn12	481 861	81	**0.48**	0.88	**5.22**	0.62	1.47	1.14	7.14	9.00	6.79
E.coli	4 638 690	4	1.80	0.84	**5.42**	1.86	**1.59**	1.82	13.07	9.00	131.30
bible	4 047 392	63	**1.28**	0.81	**5.22**	1.45	1.53	1.31	10.81	9.00	59.32
world	2 473 400	94	**0.95**	0.80	**5.09**	1.17	1.28	1.07	9.38	9.00	32.31
Average			0.71	0.88	5.23	0.86	1.26	1.15	7.75	9.00	28.58
Construction time						0.84	1.19	1.13	7.73		
Search time						0.02	0.07	0.02	0.02		