Tutorial Week 8

Definition 1. *The* suffix trie *of a string is the deterministic automaton that recognises the set of suffixes of the string and in which two different paths of same source always have distinct ends. Its computation starts with the longest suffix (the word itself), and continues by adding the upcoming suffixes one by one, in decreasing order of their length. However, this can yield a high complexity for the construction. Thus, to accelerate the process one adds to the data structure* suffix links *that, for each* fork *(node that has outgoing arc of degree 2, or has a single outgoing arc but it is terminal) corresponding to some factor av points to the vertex corresponding to v. This provides us with an algorithm that only depends on the number of vertices in the tree. To improve though on the running complexity of the creation of the data structure, one can prune the tree as to compact all non forks in a single edge, obtaining this was a* suffix tree*. In the end, as a final speed-up, one uses instead of actually strings as labels for the edges, only the positions where these factors occur, together with their lengths.*

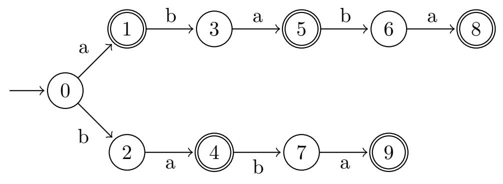
Exercise 1. *Consider the following list of sequences: ababa, abcacabb, abcacababc, and abacabacab. For each of them, construct the corresponding suffix trie. Next, construct the suffix tree of each string and append the corresponding suffix links.*

What is the complexity of each of the algorithms discussed here?

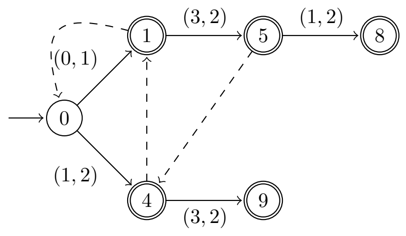
Solution: The list of ordered suffixes of *ababa* is:

Rank	i	Suff(*ababa*)
0	4	*a*
1	2	*aba*
2	0	*ababa*
3	3	*ba*
4	1	*baba*

The trie corresponding to the list of suffixes of *ababa* is depicted next:



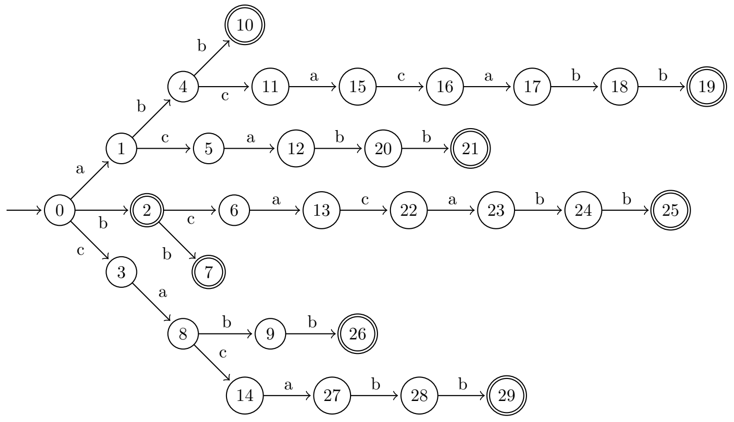
The following picture depicts the suffix tree associated to the string *ababa*, together with its corresponding suffix links:



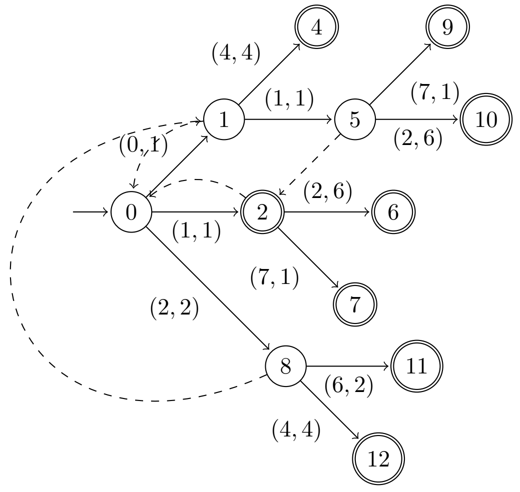
The list of ordered suffixes of *abcacabb* is:

Rank	i	Suff$(abcacabb)$
0		*abb*
1		*abcacabb*
2		*acabb*
3		*b*
4		*bb*
5		*bcacabb*
6		*cabb*
7		*cacabb*

The trie corresponding to the list of suffixes of *abcacabb* is depicted next:



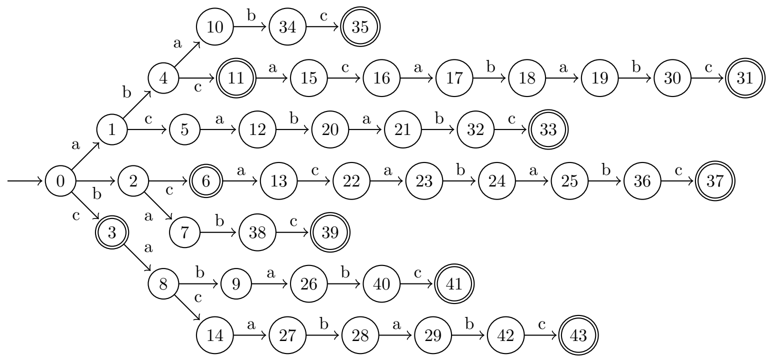
The following picture depicts the suffix tree associated to the string *abcacabb*, together with its corresponding suffix links:



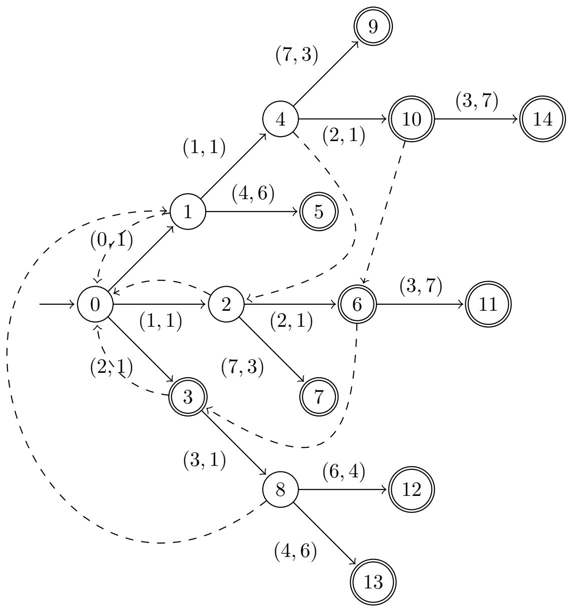
The list of ordered suffixes of *abcacababc* is:

Rank	i	Suff$(abcacababc)$
0	5	*ababc*
1	7	*abc*
2	0	*abcacababc*
3	3	*acababc*
4	6	*babc*
5	8	*bc*
6	1	*bcacababc*
7	9	*c*
8	4	*cababc*
9	2	*cacababc*

The trie corresponding to the list of suffixes of *abcacababc* is depicted next:



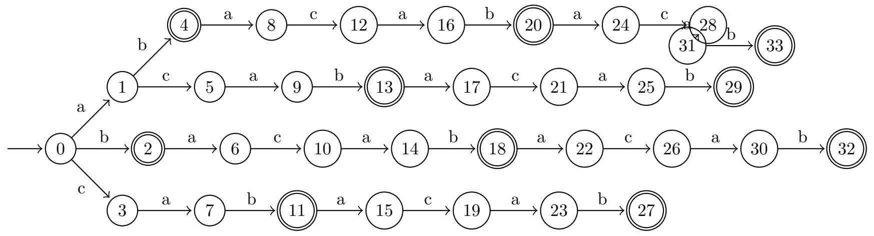
The following picture depicts the suffix tree associated to the string *abcacababc*, together with its corresponding suffix links:



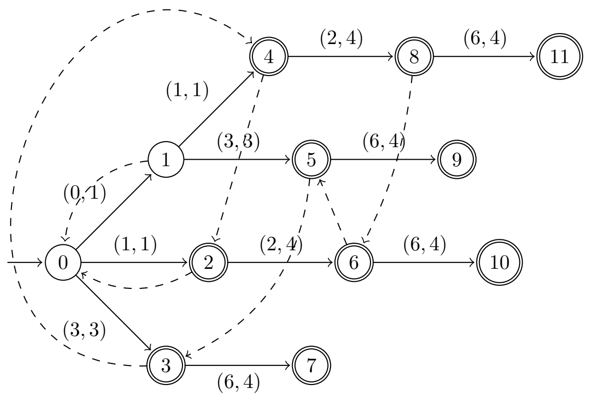
The list of ordered suffixes of *abacabacab* is:

Rank	i	Suff$(abacabacab)$
0		*ab*
1		*abacab*
2		*abacabacab*
3		*acab*
4		*acabacab*
5		*b*
6		*bacab*
7		*bacabacab*
8		*cab*
9		*cabacab*

The trie corresponding to the list of suffixes of *abacabacab* is depicted next:



The following picture depicts the suffix tree associated to the string *abacabacab*, together with its corresponding suffix links:



To construct a basic Suffix trie we need $\mathcal{O}n^2$ time, where n denotes the length of the string. When we do it in a smarter way, using suffix links, we will need $\mathcal{O}(|Q|)$ time, where $|Q|$ denotes the number of nodes of our automaton. The construction of the suffix tree can be accelerated when using pairs, as at no point there is the need to read the whole sequences, yielding a running time of $\mathcal{O}(n \log |A|)$, where $|A|$ denotes the number of elements in our alphabet. ■