451: Suffix Trees

G. Miller, K. Sutner
Carnegie Mellon University
2020/11/19

According to the Prague Stringology Club:

> *. . . a science on algorithms on strings and sequences. It solves such problems like exact and approximate pattern matching, searching for repetitions in various texts, etc. There are many areas that utilize the results of the stringology (information retrieval, computer vision, computational biology, DNA processing, etc.).*

http://www.stringology.org/

Factors

We are interested in storing all the factors of a given word w. This is obviously useful in a text search; we'll see other applications next time. We will consider a number of approaches:

- suffix tries
- suffix trees
- suffix automata
- suffix arrays

The algorithms tend to be a bit messy and combinatorial in nature, in particular the linear time ones.

Recall: Factors

Given a factorization

$$w = x\,y\,z$$

of a word $w \in \Sigma^\star$, $x/y/z$ is a prefix/factor (infix)/suffix of w, respectively.

We write

$$w[i{:}j] = w_i w_{i+1} \dots w_j$$

for the factor from position i to j, $1 \le i \le j \le n$. $w[i{:}]$ stands for the suffix $w[i{:}n]$, and $w[{:}i]$ for the prefix $w[1{:}i]$,

We write

$$\mathsf{pref}(w) \qquad \mathsf{fact}(w) \qquad \mathsf{suff}(w)$$

for the finite languages of all prefixes/factors/suffixes of $w \in \Sigma^\star$. For $n = |w|$ we have $|\mathsf{pref}(w)| = |\mathsf{suff}(w)| = n + 1$, but $|\mathsf{fact}(w)|$ can be quadratic.

Recall: Sorting Strings

Suppose we have list of strings $W = (w_1, w_2, \ldots, w_m)$ over an alphabet Σ. The size of W is

$$\mathsf{sz}(W) = n + \sum_i |w_i|$$

As usual, sorting requires $\Omega(n \log n)$ comparisons which are $O(\max|w_i|)$. In particular sorting $\mathsf{suff}(w)$ is $\Theta(n^2 \log n)$.

By comparing single letters, we can sort in $O(s)$ steps where s is the size of the input:

- Insert the words into a trie, then do a pre-order traversal.
- Use radix sort with MSD first.

So we can sort the suffixes of a word of length n in quadratic time.

Warning: This is fine for the standard case of fixed, small size alphabets, but in general we pick up another factor $\log|\Sigma|$ (uniform versus logarithmic model).

Recall: Tries

Fix some finite alphabet Σ (of non-astronomical size so we may assume a letter is size 1).

A trie[1] T over Σ is a rooted, edge-labeled tree. The edge labels are letters in Σ, for each node there is at most one out-edge labeled $a \in \Sigma$. Write $\mathsf{lab} : E \to \Sigma$ for the edge labels.

Any path $\pi = e_1, e_2, \ldots, e_k$ in the trie corresponds to a word over $\Sigma^\star$:

$$\mathsf{lab}(\pi) = \mathsf{lab}(e_1)\mathsf{lab}(e_2)\ldots\mathsf{lab}(e_k)$$

We are interested in paths starting at the root, and ending at a terminal (essential) node. The collection of associated words is a finite language, in symbols $\mathcal{L}(T)$.

[1]Should be pronounced like "tree" for retrieval. To preserve student sanity I will say "try." Apologies to Ed Fredkin.

Challenge: We want to build a trie $\mathfrak{T}(w)$ that stores all factors of a word w.

Clearly this will be useful for text search, but there is actually an amazing number of applications, more later.

Given some sort of finite-state-machine type of approach and the observation $\mathsf{fact}(w) = \mathsf{pref}(\mathsf{suff}(w))$ it suffices to just store all suffixes.

The brute-force approach is to use a standard trie and simply insert all the suffixes $w[i{:}]$, using a vanilla insert operation.

Let's say we insert in order $w = w[1{:}], w[2{:}], \ldots, w[n{:}]$.

```
defun VanillaSuffixTrie($w : \Sigma^\star$)

    initialize trie

    forall $i = 1, \ldots, |w|$ do
        vanilla-insert $w[i{:}]$            start at root each time
    od
```

Lemma

Brute-force construction of $\mathfrak{T}(w)$ is $\Theta(|w|^2)$.

This is quadratic even when $w = a^n$.

Terminology

There are two somewhat orthogonal ways to discuss the objects in this lecture:

Data Structures Just think of building yet another useful data structure that has many applications in string processing.

Automata Theory Instead recognize these data structures as thinly disguised finite state machines, and import all the standard machinery from there.

In the first setting, special nodes in a factor trie that correspond to suffixes are called essential. In the second, they are terminal states. Similarly one talks about the root versus the initial state, and so on.

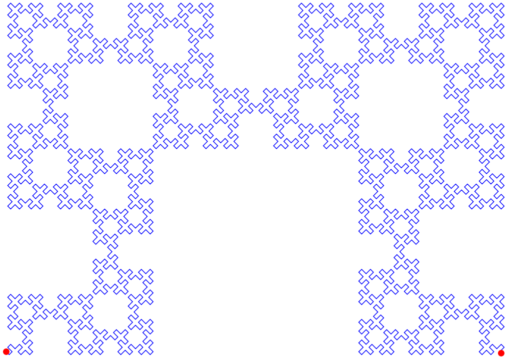
These may seem utterly irrelevant, but having a good conceptual framework is very important.

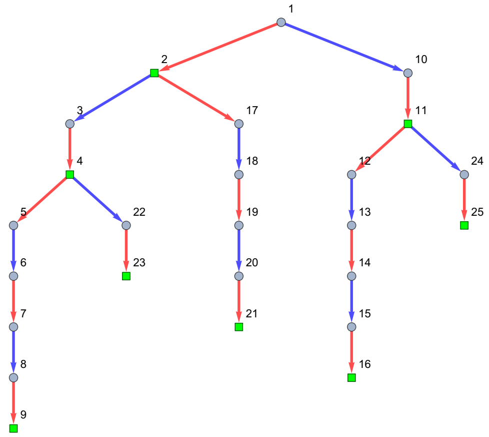
Example: Fibonacci Word

Fibonacci words are defined by

$$F_1 = b \qquad F_2 = a \qquad F_n = F_{n-1}\,F_{n-2}$$

These have lots of interesting properties and provide useful testcases. A turtle-generated fractal based on F_{20}





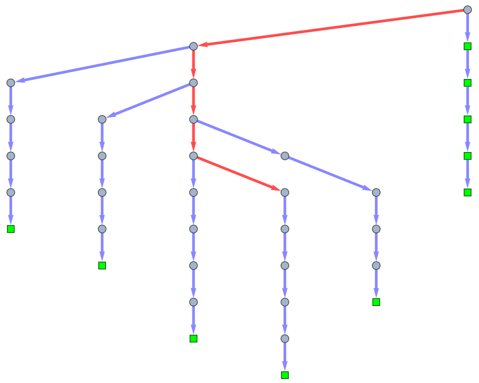
word $F_6 = abaababa$

edge colors: a red, b blue

essential nodes are square/green

node labels indicate insertion order

Example: Quadratic Trie



word $a^n b^n$ for $n = 5$

edge colors: a red, b blue

essential nodes are square/green

Grumpy complaint: why can't I tell the rendering algorithm to place the red edges on the leftmost branch?

Endmarkers

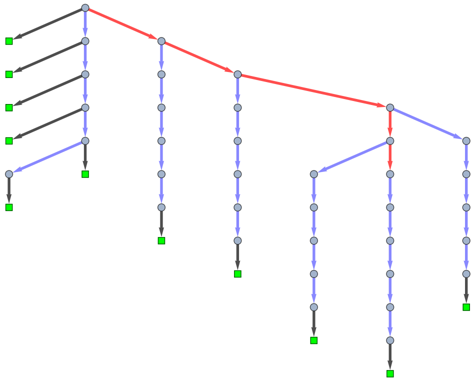
Claim: There will be essential internal nodes iff some proper prefix of a suffix (aka factor) is also a suffix.

To push all the essential nodes to the leaves one can attach an endmarker, a symbol that appears nowhere else in the given word.

$$w_1 w_2 \dots w_n \#$$

Note that some authors adopt a slightly dangerous convention: since it is boring to write the dang $\#$ over and over again, they simply omit it. It's just a phantom. We will not do this.

Example: Quadratic Trie

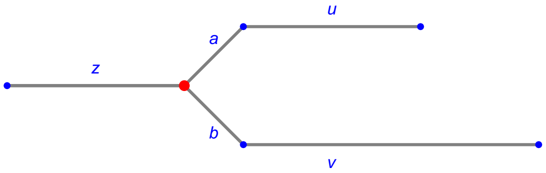


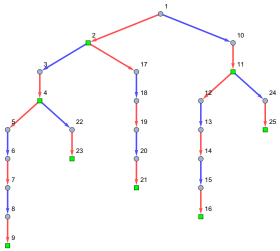
this time with endmarker: $a^n b^n \#$

All the essential nodes are now leaves.

back

If the word W has two suffixes of the form zau and zbv, the state associated with z has outdegree at least two: a fork. In this case we call the strings z a head, and au/bv a tail.





For F_6 we have 3 forks: a, aba, ba.

Convention

If we think of a trie as a finite state machine, it is entirely natural to conflate input $x \in \Sigma^\star$ with the actual state $\delta(q_{\mathsf{ini}}, x)$ where δ is the transition function. Since we are dealing with deterministic machines, there is no problem with this.

Of course, in a real implementation, a state would be a `uint` or some such.

Unless you care deeply about types (and your own sanity).

Suffix Links

For long words, quadratic time is obviously a problem, in particular when the number of factors is sub-quadratic.

Here is a first step towards better algorithms: we would like to make the running time of the construction depend on the size of the resulting trie.

Suppose we have a fork/head az with tails u and v. Note that z must be another fork (with shorter head and same tails). How about computing a link (a pointer) from az to z, in the hope of speeding up the trie construction. This called the suffix link function, $\mathsf{sl}(az) = z$.

As a word function, this is trivial, but we want to compute it cheaply for all forks in the trie. Actually, the real algorithm constructs a few more links (fork plus parent, final nodes of outdegree 1).

Insertion Order

Say we have a head az and tails u and v, $|u| < |v|$. Recall that we insert longest-first.

Proposition

We have the following insertion order

$$azv \prec zv \preceq azu \prec zu$$

Proof. It is clear that azv is first, and zu is last.

Suppose $azu \prec zv$. Then $|azu| > |zv|$, so that $|v| = |u| + 1$. Both are suffixes, so $v = bu$.

Similarly, $zv = zbu = azu$ and it follows that $a = b$, $z \in a^\star$. So $zv = azu$.

□

The inner loop of the auxiliary function makeLinks follows/builds a chain of suffix links until the initial state is reached, or a transition $p_1 \xrightarrow{w_k} q_1$ is found that already has links defined for both endpoints.

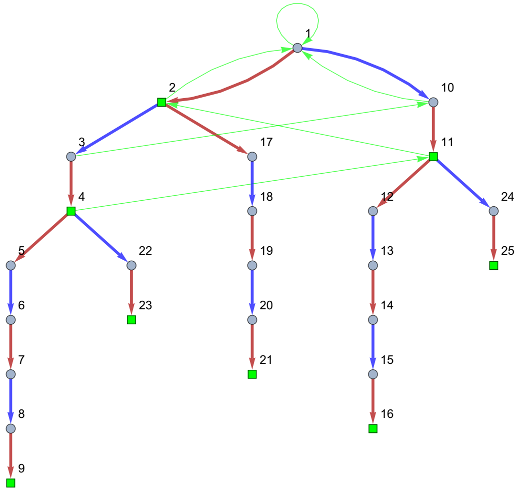
$$\begin{tikzcd}
p \arrow[r, dashed] \arrow[d, "a"'] & \mathsf{sl}(p) \arrow[r, dashed] \arrow[d, "a"'] & \mathsf{sl}^2(p) \arrow[d, "a"'] \\
q \arrow[r, dashed] & \mathsf{sl}(q) \arrow[r, dashed] & \mathsf{sl}^2(q)
\end{tikzcd}$$

The main function SuffixTrie exploits these links to speed up the insertion process for the suffixes (as opposed to just using vanilla insert at the bottom of the code).

```
defun SuffixTrie($w : \Sigma^\star$)

$\mathsf{sl}(q_{\mathsf{ini}}) = q_{\mathsf{ini}}$
$(r, k) = (q_{\mathsf{ini}}, 1)$

forall $i = 1, \ldots, n$ do
    $k = \max(i, k)$
    $(r, k) =$ makeLinks($\mathsf{sl}(r), k$)

    vanilla-insert $w[k{:}]$ at $r$
od
```

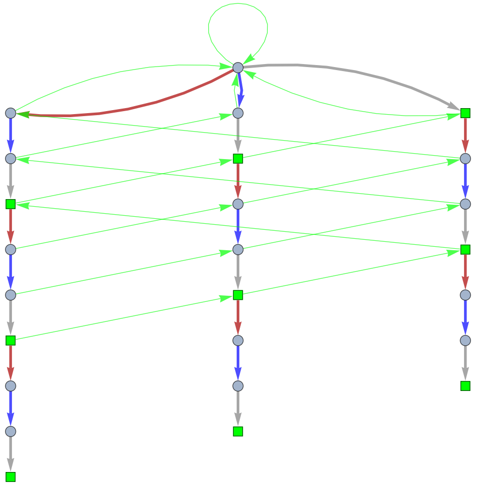
```
defun makeLinks($p : Q, k : \mathbb{N}$)

while $k \leq n$ and $\delta(p, w_k) \downarrow$ do
    $q = \delta(p, w_k)$
    $(p_1, q_1) = (p, q)$
    while $p_1 \neq q_{\mathsf{ini}}$ and $\mathsf{sl}(q_1) \uparrow$ do
        $\mathsf{sl}(q_1) = \delta(\mathsf{sl}(p_1), w_k)$
        $(p_1, q_1) = (\mathsf{sl}(p_1), \mathsf{sl}(q_1))$
    od
    if $\mathsf{sl}(q_1) \uparrow$ then $\mathsf{sl}(q_1) = q_{\mathsf{ini}}$
    $(p, k) = (q, k{+}1)$
od
return $(p, k)$
```

EXAMPLE $F_6 = abaababa$



Consider head aba, tails $ababa$ and ba.

i	r	k	links	r, k change	inserts
1	1	1		$(1,1) \to (1,1)$	$1 \xrightarrow{a} 2, 1$
2	1	2		$(1,2) \to (1,2)$	$1 \xrightarrow{b} 10, 2$
3	1	3	2:1	$(1,3) \to (2,4)$	$2 \xrightarrow{a} 17, 4$
4	2	4	3:10, 10:1, 4:11, 11:2	$(1,4) \to (4,7)$	$4 \xrightarrow{b} 22, 7$
5	4	7		$(11,7) \to (11,7)$	$11 \xrightarrow{b} 24, 7$
6	11	7		$(2,7) \to (4,9)$	
7	4	9		$(11,9) \to (11,9)$	
8	11	9		$(2,9) \to (2,9)$	



If one allows internal terminals, links also have to handle terminal nodes with outdegree 1.

Lemma

The suffix link algorithm constructs $\mathfrak{T}(w)$ in $O(s)$ steps where s is the number of nodes in $\mathfrak{T}(w)$ (output-optimal).

Proof.

The overhead in the main loop of SuffixTrie is $O(n)$ over the whole execution of the algorithm.

Calls to makeLinks are also $O(s)$ overall: count the number of suffix links created.

The nested loop of SuffixTrie is also $O(s)$ overall, since new states are created there.

$\square$

Saving Space

Suffix tries can be quadratic in size, so one would like to find more compact representations. There are two main approaches:

Label Compaction Allow edges to be labeled by words rather than just letters.

Subtree Sharing Collapse nodes with identical subtrees (including labels).

The first method is all tree surgery. The second corresponds to minimization in finite state machines.

One nice feature of these methods: both can be applied together, in either order.

Word Labels

In a standard trie, edges are labeled by letters. It is entirely natural to use word labels instead, $\mathsf{lab} : E \to \Sigma^+$. If q has indegree/outdegree 1, then contract

$$p \xrightarrow{x} q \xrightarrow{y} r \quad \rightsquigarrow \quad p \xrightarrow{xy} r$$

Call the trie deterministic if there is no node with two distinct out-edges e and e' such that $\mathsf{lab}(e) = au$ and $\mathsf{lab}(e') = av$, $a \in \Sigma$, $u, v \in \Sigma^\star$.

In a deterministic trie, we can still search for a path in essentially the same manner as in plain tries.

If all edge labels are factors of a fixed word, word edge labels can easily be implemented as a pair of pointers using constant space.

Exercise

Figure out how to implement label compaction. How does insertion and deletion work in this setting?

Definition

The suffix tree $\mathfrak{S}(w)$ for a word w of length n is a compacted, deterministic trie with n leaves corresponding to the suffixes.

There is a more general definition that allows for internal essential nodes, but we will use the restricted form: we can use our endmarker trick to make sure a suffix tree always exists.

Note that all internal nodes must have outdegree at least two, so there are at most $n-1$ of them.

We can easily build a suffix tree in quadratic time: first build a trie, then do compaction. expl

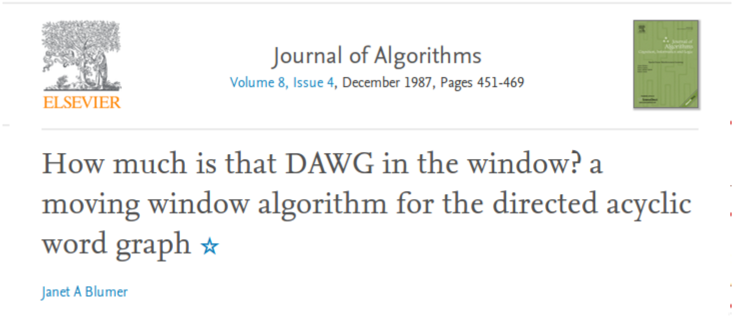
Innocent Question: Can we do this in linear time?

First, let us consider a closely related problem: since fact(w) is finite and hence regular, there must be a minimal DFA recognizing the factors of w. We will drop the sink (partial DFA).

Question: Can we build the minimal PDFA for fact(w) in linear time?

The minimal PDFA for suff(w) is called the suffix automaton for w, written $\mathfrak{A}(w)$.

Finite state machines without loops are also called DAWGs: directed acyclic word graphs.



ELSEVIER

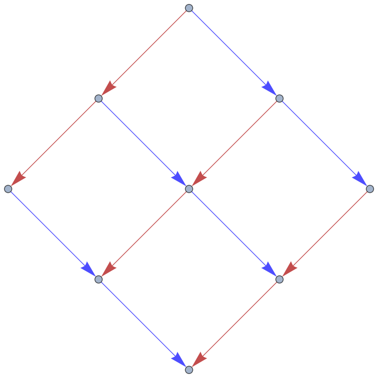
Journal of Algorithms

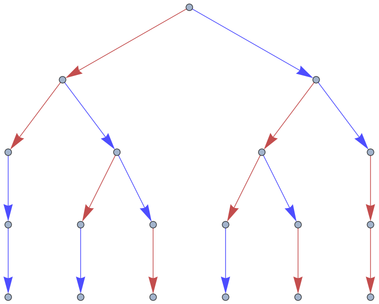
Volume 8, Issue 4, December 1987, Pages 451-469

How much is that DAWG in the window? a moving window algorithm for the directed acyclic word graph ☆

Janet A Blumer

We can unfold any acyclic PDFA into a trie. If the PDFA was minimal, sharing subtrees takes us back to the machine.





Two descriptions for $\{aabb, abab, abba, baab, baba, bbaa\}$.

Recall: Myhill-Nerode Congruence

The right context (or left quotient) of a word x wrto the language $\mathsf{suff}(w)$ is

$$\mathcal{R}_w(x) = \{ z \in \Sigma^\star \mid xz \in \mathsf{suff}(w) \}$$

This produces a right congruence on $\Sigma^\star$: $x \equiv_w y$ if $\mathcal{R}_w(x) = \mathcal{R}_w(y)$: an equivalence relation of finite index such that

$$x \equiv_w y \text{ implies } \forall\, u\, (xu \equiv_w yu)$$

It is well-known that the minimal DFA for any regular language has size the number of right contexts of that language: we can think of the right contexts as the states of an (abstract) automaton.

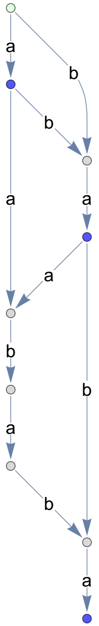
For some regular languages, this leads directly to an efficient algorithm to construct the minimal PDFA for the language.

Example: $F_6 = abaababa$

Clearly $\mathcal{R}_w(x) \neq \emptyset$ iff x is a factor of w. Ignore non-factors.

Minimal suffix automaton for F_6, let $L = \{abaababa, baababa, aababa, ababa, baba, aba, ba, a, \varepsilon\}$

1	L
2	$\{baababa, ababa, baba, ba, \varepsilon\}$
3	$\{aababa, aba, a\}$
4	$\{baba\}$
5	$\{ababa, ba, \varepsilon\}$
6	$\{aba\}$
7	$\{a\}$
8	$\{ba\}$
9	$\{\varepsilon\}$



Congruence

$L = \{abaababa, baababa, aababa, ababa, baba, aba, ba, a, \varepsilon\}$

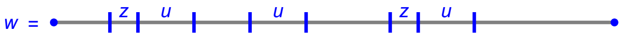
$\mathcal{R}_w(x)$	$\equiv_w$ classes
L	$\{\varepsilon\}$
$\{baababa, ababa, baba, ba, \varepsilon\}$	$\{a\}$
$\{aababa, aba, a\}$	$\{ab, b\}$
$\{ababa, ba, \varepsilon\}$	$\{aba, ba\}$
$\{baba\}$	$\{abaa, baa, aa\}$
$\{aba\}$	$\{abaab, baab, aab\}$
$\{ba\}$	$\{abaaba, baaba, aaba\}$
$\{a\}$	$\{abaabab, baabab, aabab, abab, bab\}$
$\{\varepsilon\}$	$\{abaababa, baababa, aababa, ababa, baba\}$

This explains the minimal PDFA on slide 33.

The following claims are easy to establish using pictures like the one below.

Claim 1: Let $|u| \leq |v|$. Then either

- $\mathcal{R}_w(u) \cap \mathcal{R}_w(v) = \emptyset$ or
- $\mathcal{R}_w(v) \subseteq \mathcal{R}_w(u)$ and u is a suffix of $v = zu$.



Clearly $\equiv_{wa}$ is a refinement of $\equiv_w$. Here is a more detailed description:

Claim 2:

$$\mathcal{R}_{wa}(z) = \begin{cases} \mathcal{R}_w(z)\, a \cup \{\varepsilon\} & \text{if } z \text{ is a suffix of } wa, \\ \mathcal{R}_w(z)\, a & \text{otherwise.} \end{cases}$$

Splitting Classes

Lemma

Let z be the longest suffix of wa that appears on w and let z' be the longest factor of w for which $z' \equiv_w z$. Then for all factors u and v of w: if $u \equiv_w z$ then

$$u \equiv_{wa} \begin{cases} z & \textit{if } |u| \leq |z|, \\ z' & \textit{otherwise.} \end{cases}$$

Otherwise $u \equiv_w v \iff u \equiv_{wa} v$.

Note that z and z' depend only on wa, not the individual congruence classes.

Also, $z = z'$ implies that the equivalence classes of $\equiv_w$ and $\equiv_{wa}$ are the same. This happens in particular when a does not appear in w (so $z = \varepsilon$).

Example

For $W = a^n$ the congruence classes are just suff(W). Increasing n adds 1 class.

For $W = a^n b^m$, $0 < n, m$, the congruence classes are

$$\begin{array}{ll} \{a^i\} & i = 0, \ldots, n \\ \{\, a^i b^j \mid i \in [n] \,\} & j = 1, \ldots, m \\ \{b^j\} & j = 1, \ldots, m-1 \end{array}$$

Hence there are $n + 2m$ classes. Increasing m adds 2 classes.

Exercise

What happens with $a^n b^n a$?
How about the number of classes of a Fibonacci word?

The state/transition complexity of an automaton $\mathcal{A}$ is the number of states/transitions; written $\mathrm{sc}(\mathcal{A})$ and $\mathrm{tc}(\mathcal{A})$.

Theorem

Let $n = |w| \geq 2$. The suffix automaton for w has $\mathrm{sc}(\mathfrak{A}) \leq 2n - 1$ *and* $\mathrm{tc}(\mathfrak{A}) \leq 3n - 3$.
Moreover, it can be constructed in linear time and space.

Sketch of proof.

For $n = 1$ a 2-state machine clearly works. Essentially by the last lemma, moving from w to wa will increase the number of states by at most 2, done by induction.

One can show that the transition complexity is at most $\mathrm{sc}(\mathfrak{A}) + n - 2$.

For the actual construction, one uses an idea similar to the suffix links from above.

□

The first linear time algorithm dates back to Weiner 1973, but a less convoluted method is

Theorem (Ukkonen 1996)

A suffix tree can be constructed in linear time.

The idea is to first construct an implicit suffix tree: remove all the # labeled edges, then perform compaction if necessary. ISTs are built inductively for all prefixes of W. In the end the IST for W is converted into a real suffix tree.

ISTs sound like a bold step in the wrong direction: the IST for W has fewer leaves than the suffix tree for $W\#$ iff some suffix is a prefix of another—the endmarker was introduced exactly to avoid this.

But in the end everything works fine.

Details

Alas, the details are too messy for us. If you are interested, take a look at

M. Lothaire
Applied Combinatorics on Words
Cambridge University Press, 2005

D. Gusfield
Algorithms on Strings, Trees, and Sequences
Cambridge University Press, 1997