BUNDLED SUFFIX TREES

Luca Bortolussi[1] Francesco Fabris[2] Alberto Policriti[1]

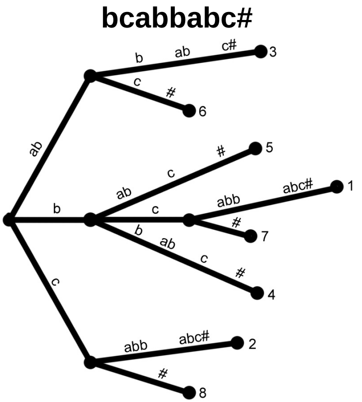
[1]Department of Mathematics and Computer Science
University of Udine

[2]Department of Mathematics and Computer Science
University of Trieste

IFIP TCS 2006, Santiago, Chile, 23rd–24th August 2006

Outline

Suffix Trees



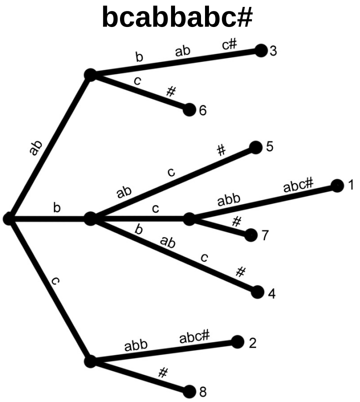
A Suffix Tree is a data structure revealing the internal structure of a string.
They occupy $O(n)$ space and can be built in $O(n)$ time.

They are efficient for:

- *Exact String Matching*
- *Longest Exact Common Substring Problem*
- *Identifying Exactly Repeated Patterns*

- Gusfield D., *Algorithms on strings, trees and sequences*, Cambridge University Press, 1997.
- E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249-260, 1995.

Limitations of Suffix Trees



Suffix Trees cannot deal naturally with approximate string matching problems. (Hamming or Edit distance)

Two difficult problems:

- *Longest Common Approximate Substring Problem*
- *Extraction of approximately repeated patterns*

- Gusfield D., *Algorithms on strings, trees and sequences*, Cambridge University Press, 1997.
- Landau G.M., Vishkin U., Efficient String Matching with k Mismatches, *Theoretical Computer Science*, 43, 239-249, 1986.

Extending Suffix Trees

THE TARGET

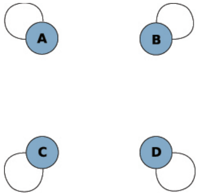
Extending Suffix Trees in order to solve *in a simple way* some classes of *approximate string matching problems*.

Bundled Suffix Trees

Bundled Suffix Trees extend suffix Trees.

- They incorporate *approximate information*;
- They can be used *like Suffix Trees* for:
  - Longest Common Approximate Substring Problem
  - Extraction of approximately repeated patterns

Approximate Matching

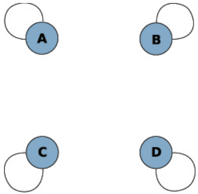


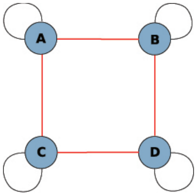
Character matching is a relation among letters
(in fact, it is the equality relation)

We model *approximate matching* as a non-transitive relation among letters:

two strings "match" if all their letters are in relation.

Approximate Matching





Character matching is a relation among letters
(in fact, it is the equality relation)

We model *approximate matching* as a non-transitive relation among letters:

two strings “match” if all their letters are in relation.

Non-Transitive Relation: An Example

Modeling a relation based on Hamming Distance

- Start from a *basic alphabet* (e.g. binary: $\mathcal{A} = \{0, 1\}$)
- Construct an alphabet composed of **macrocharacters** (e.g. $\overline{\mathcal{A}} = \{00, 01, 10, 11\}$)
- Two letters $x, y \in \overline{\mathcal{A}}$ are in *relation* if and only if $d_H(x, y) \leq D$ (e.g. $D = 1$).

The Relation Graph

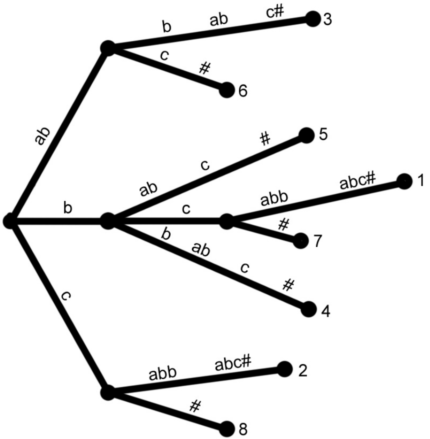
$$
\begin{array}{ccc}
00 & \leftrightarrow & 01 \\
\updownarrow & & \updownarrow \\
10 & \leftrightarrow & 11
\end{array}
$$

- Relation is *non-transitive*
- It encapsulates a (*restricted*) form of distance.

Bundled Suffix Tree: An Example

bcabbabc

$a \leftrightarrow b \leftrightarrow c$



- We start from the *suffix tree* for the string.
- Let's compare suffix 3 and suffix 1:

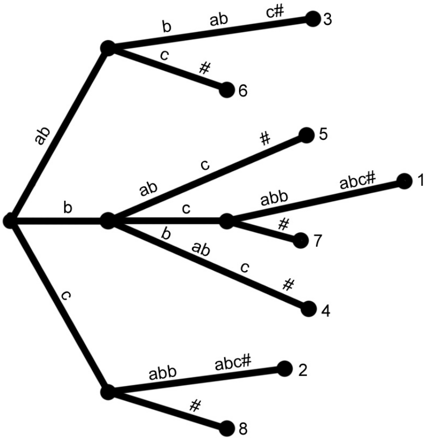
b	c	a	b	b	a	b	c
$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\not\Updownarrow$		
a	b	b	a	c	c		

- After **bcabb** in the tree, we put a red node with label 3.
- Due to symmetry, there is also a red node with label 1 after **abbab**.

Bundled Suffix Tree: An Example

bcabbabc

$a \leftrightarrow b \leftrightarrow c$



- We start from the *suffix tree* for the string.
- Let's compare suffix 3 and suffix 1:

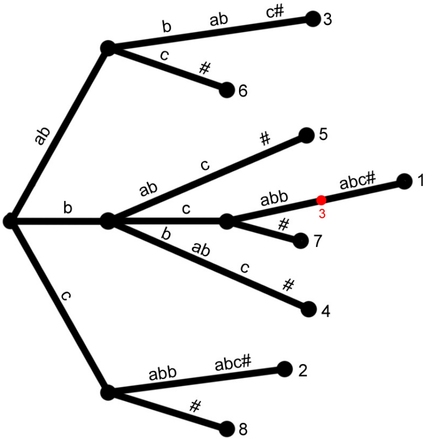
b	*c*	*a*	*b*	*b*	*a*	*b*	*c*
$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\not\Updownarrow$		
a	*b*	*b*	*a*	*c*	*c*		

- After **bcabb** in the tree, we put a red node with label 3.
- Due to symmetry, there is also a red node with label 1 after **abbab**.

Bundled Suffix Tree: An Example

bcabbabc

$a \leftrightarrow b \leftrightarrow c$



- We start from the *suffix tree* for the string.
- Let's compare suffix 3 and suffix 1:

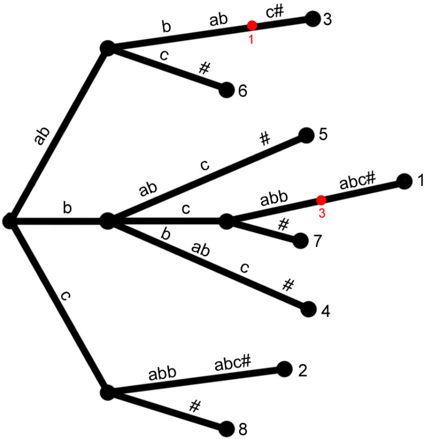
b	*c*	*a*	*b*	*b*	*a*	*b*	*c*
$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\not\Updownarrow$		
a	*b*	*b*	*a*	*c*	*c*		

- After **bcabb** in the tree, we put a red node with label 3.
- Due to symmetry, there is also a red node with label 1 after **abbab**.

Bundled Suffix Tree: An Example

bcabbabc

$a \leftrightarrow b \leftrightarrow c$

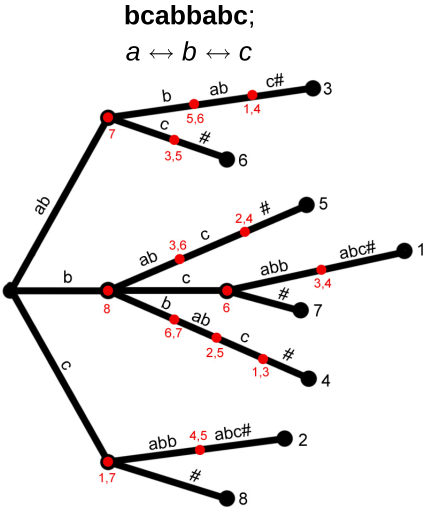


- We start from the *suffix tree* for the string.
- Let's compare suffix 3 and suffix 1:

b	c	a	b	b	a	b	c
$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\Updownarrow$	$\cancel{\Updownarrow}$		
a	b	b	a	c	c		

- After **bcabb** in the tree, we put a red node with label 3.
- Due to symmetry, there is also a red node with label 1 after **abbab**.

Bundled Suffix Tree: An Example



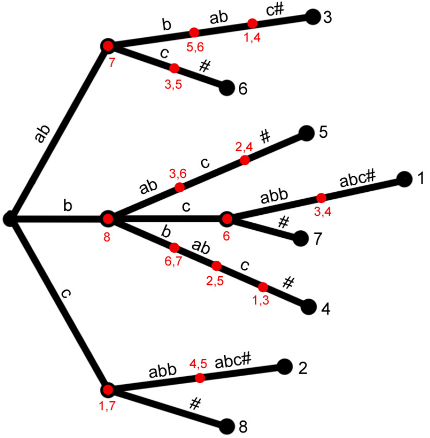
If we do this process for every couple of suffixes, we build a Bundled Suffix Tree!

Note that this data structure is *in the middle* between a *suffix tree* and a *suffix trie*.

Bundled Suffix Tree: An Example

bcabbabc;

$a \leftrightarrow b \leftrightarrow c$



Bundled Suffix Trees can be used to:

- solve the Longest Common Approximate Substring Problem *with respect to a given relation* (just find the lowest red node).
- extract information about approximately repeated patterns.

How Big?

The number of red nodes inserted depends on:

- the relation
- the structure of the text.

In the worst case, the number of red nodes is quadratic in the length of the text S. Example

On average, the number of red nodes is limited by

$$m^{1+\delta}, \ \delta = \log_{1/p^+} C.$$

(m is the length of the text, p^+ is the normalized frequency of the most common letter in S, C depends on the relation)

$1+\delta$ is slightly greater than one! Example

How Fast?

Naive Algorithm

- The naive algorithm for building a BuST tries to "match" *every suffix* of the text*along every branch* of the suffix tree, until a "mismatch" is found.
- It can be *quadratic* in the *worst case*.
- An analysis based on the *average shape of a suffix tree* shows that its average complexity is bounded by $m^{1+\delta'}$ (δ' just slightly greater that δ).

- W. Szpankowski. A Generalized Suffix Tree and its (Un)expected Asymptotic Behaviors. *SIAM J. Comput.* 22(6): 1176-1198 (1993)
- P. Jacquet, B. McVey, W. Szpankowski. Compact Suffix Trees Resemble PATRICIA Tries: Limiting Distribution of Depth, *Journal of the Iranian Statistical Society*, 3, 139-148, 2004.

Faster

Efficient Algorithm

We found an "McCreight-like" algorithm that is linear in the size of the output.

Intuitions

- It processes the suffixes backwards.
- It is based on the concept of inverse suffix links. Show Details
- It identifies the red nodes for suffix i by processing the red nodes for suffix $i + 1$. Show Details

Experimental Results

- We have implemented the naive algorithm for the construction of BuST.
- We have tested it with relations induced by hamming distance, defined over DNA-macrocharacters.
- With macrocharacters of size 4 ($X \leftrightarrow Y \Leftrightarrow d_H(X, Y) \leq 1$) the algorithm can process texts of length 100K in few seconds.
- The number of red nodes grows tamely. Show Details

Measures of surprise: exact case

z-score

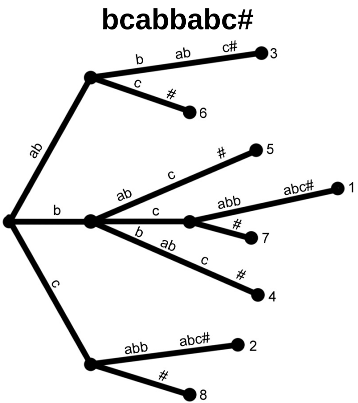
$$\delta(\alpha) = \frac{f(\alpha) - E(\alpha)}{N(\alpha)}$$

- $f(\alpha)$ is the observed frequency of α
- $E(\alpha)$ is the expected frequency of α
- $N(\alpha)$ is a normalization factor (e.g. the variance or its first-order approximation).

Monotonicity

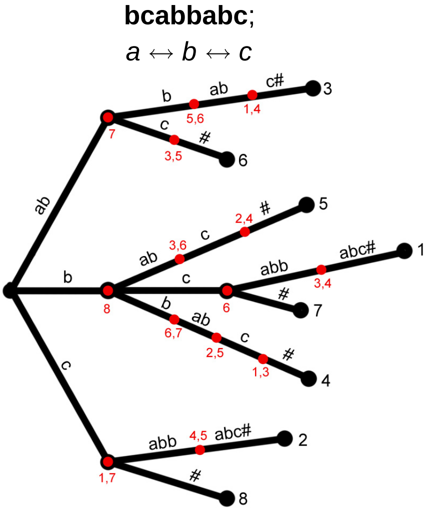
- If $f(\alpha) = f(\alpha\beta)$ then $\delta(\alpha) \leq \delta(\alpha\beta)$.
- δ needs to be computed only for *maximal strings* at a fixed frequency. These are exactly the strings ending at nodes of the Suffix Tree.

Computing the z-score



- Using a Suffix Tree, we can compute and store the z-score for all “interesting” substrings of a given text in linear time and space (given that we can compute E and N in linear time and space).
- A. Apostolico, M.E. Block, S. Lonardi. Monotony of surprise and the large-scale quest for unusual words. *Journal of Computational Biology*, 7(3-4), 2003.

Measures of Surprise in the Approximate World



- Let's consider as occurrences of β in α all the substrings β' that are in relation with β.
- Reasoning as in the exact case, we can use a BuST to compute the z-score for all interesting substrings of α in *time and space proportional to the BuST's size*.

Measures of Surprise in the Approximate World

If we use an Hamming-like relation built on macrocharacters, we are counting all the occurrences of a string with *distance bounded by a threshold proportional to the string's length*.

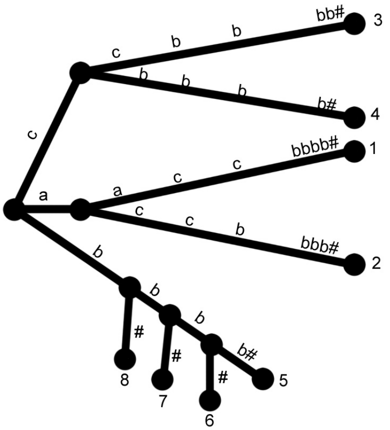
Pros and Cons

- **Pros**:
  - the algorithm runs in time proportional to the number of maximal substrings (w.r.t. δ).
  - BuST provides a compact way to store and retrieve this information.
- **Cons**:
  - the macrocharacters introduce rigidity (we can count compute the z-score only for strings of length multiple of the macrocharacter's size).
  - the distance must be distributed evenly among macrocharacters.

Conclusions

- We have introduced Bundled Suffix Trees, a new data structure extending suffix trees.
- Given a relation among characters encoding some sort of approximate information, a BuST reveals the inner structure of the strings w.r.t. this relation (all this information is internal w.r.t. the processed string).
- BuST can be used for all the problems related to the inner structure of the string, like computation of approximated frequency.
- The structure is based on a very general concept of non-transitive relation among characters. The use of Hamming-like relation on tuples is just a possible example.
- Its *size is slightly more than linear on average*, and there's a *fast (McCreight-like) algorithm* to build it.

Quadratic BuST



- Let's consider the text

$$\underbrace{a\ldots a}_{m}\underbrace{c\ldots c}_{m}\underbrace{b\ldots b}_{2m},$$

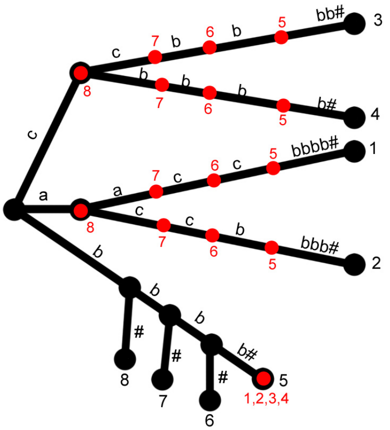
over $\{a,b,c,d\}$, with

$$\begin{array}{ccc} a & \leftrightarrow & b \\ \updownarrow & & \updownarrow \\ d & \leftrightarrow & c \end{array}$$

- The number of nodes surrounded by the red box is quadratic in m!

Return

Quadratic BuST



- Let's consider the text

$$\underbrace{a\ldots a}_{m}\underbrace{c\ldots c}_{m}\underbrace{b\ldots b}_{2m},$$

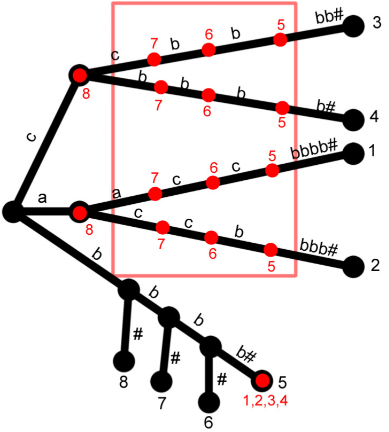
  over $\{a, b, c, d\}$, with

$$\begin{array}{ccc} a & \leftrightarrow & b \\ \updownarrow & & \updownarrow \\ d & \leftrightarrow & c \end{array}$$

- The number of nodes surrounded by the red box is quadratic in m!

Return

Quadratic BuST



- Let's consider the text

$$\underbrace{a\ldots a}_{m}\underbrace{c\ldots c}_{m}\underbrace{b\ldots b}_{2m},$$

over $\{a, b, c, d\}$, with

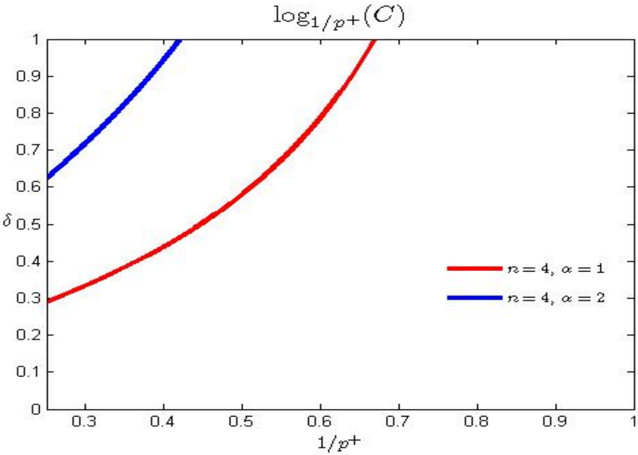
$$\begin{array}{ccc} a & \leftrightarrow & b \\ \updownarrow & & \updownarrow \\ d & \leftrightarrow & c \end{array}$$

- The number of nodes surrounded by the red box is quadratic in m!

Return

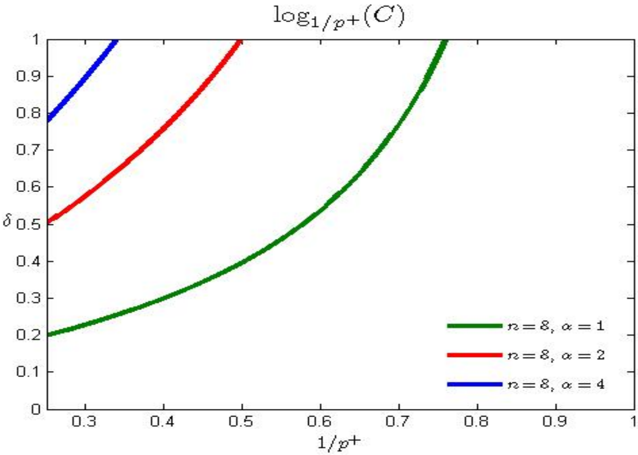
The exponent δ

Return

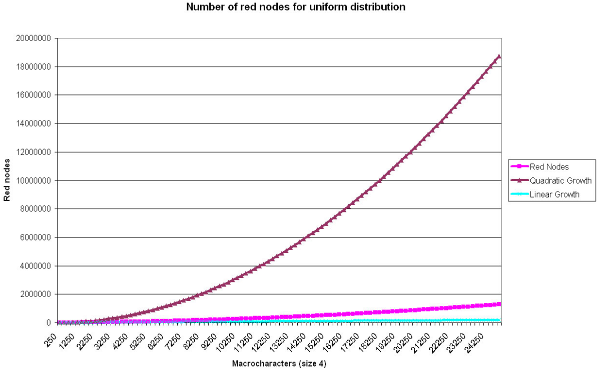


The exponent δ

Return



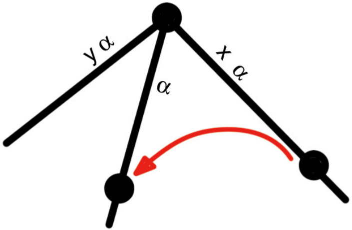
Test



Number of macrocharacters of length 4 over DNA alphabet.
Test strings are generated according to a uniform p.d.

Return

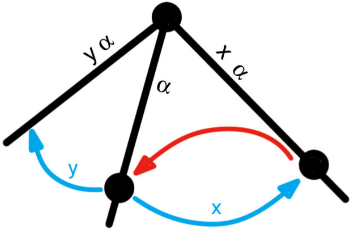
Inverse Suffix Links



- A crucial role in the fast construction of suffix trees is played by suffix links.
- Suffix links are pointers from nodes with path label $x\alpha$ to nodes with path label α.
- Whenever there is a node with path label $x\alpha$, there's also a node with path label α.

Return

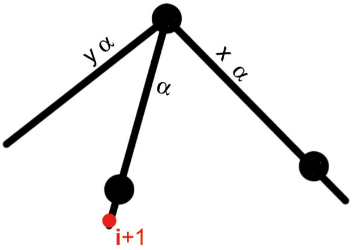
Inverse Suffix Links



- Inverse suffix links are pointers from nodes with path label α to positions in the tree labeled $x\alpha$, for each x in the alphabet such that $x\alpha$ is a substring of S.
- They can point in the middle of an arc.
- If a ISL takes from α to $x\alpha$, it is labeled with x.

Return

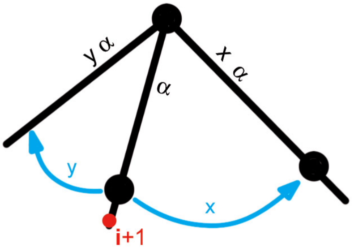
The Algorithm



- Red nodes for suffix $S[i]$ can be computed from *red nodes for suffix* $S[i+1]$, using *Inverse Suffix Links*.
- Suppose a red node for suffix $S[i+1]$ is just under a “black” node with path label α.
- From this node, we can cross all inverse suffix links labeled with characters in relation with $S(i)$.
- With a skip and count trick, we can identify the positions of red nodes for $S[i]$.

Return

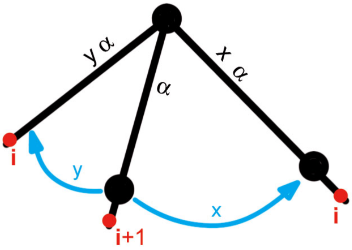
The Algorithm



- Red nodes for suffix $S[i]$ can be computed from *red nodes for suffix* $S[i+1]$, using *Inverse Suffix Links*.
- Suppose a red node for suffix $S[i+1]$ is just under a "black" node with path label α.
- From this node, we can cross all inverse suffix links labeled with characters in relation with $S(i)$.
- With a skip and count trick, we can identify the positions of red nodes for $S[i]$.

Return

The Algorithm



- Red nodes for suffix $S[i]$ can be computed from *red nodes for suffix* $S[i+1]$, using *Inverse Suffix Links*.
- Suppose a red node for suffix $S[i+1]$ is just under a “black” node with path label α.
- From this node, we can cross all inverse suffix links labeled with characters in relation with $S(i)$.
- With a skip and count trick, we can identify the positions of red nodes for $S[i]$.

Return