TUCS

Special course in Computer Science: Advanced Text Algorithms

Lecture 4: Suffix trees

Eugen Czeizler
Department of IT, Abo Akademi
http://combio.abo.fi/teaching/textalg/

(slides originally by I. Petre, E. Czeizler, V. Rogojin)

Suffix trees

- Suffix tree = data structure exposing the internal structure of a string.

- Suffix trees store the suffixes of a text.

- Any factor of a string can be extended to a suffix of the text.

- By storing efficiently the suffixes of a text, we get direct access to all the factors of that text.

Suffix trees

- Suffix trees have numerous applications, often providing linear-time solutions to challenging string problems, e.g.,:
  - Exact pattern matching
  - The longest common substring
  - The longest repeated substring inside a given text
  - Frequent common substrings of a set of strings, i.e., that *occur in a large number of distinct strings*
  - Suffix arrays
  - Genome-scale projects

Definitions

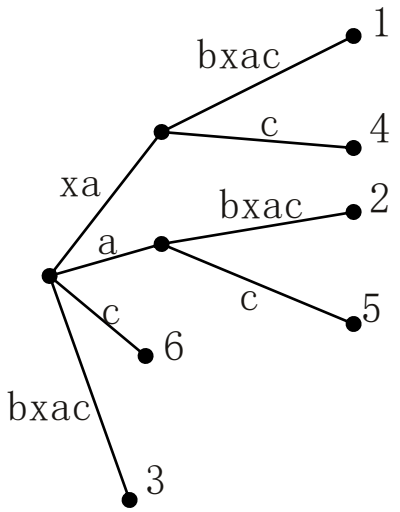
- A suffix tree T for a string S[1..m] of length m is a directed rooted tree with:
  - exactly m leaves numbered 1,..,m
  - at least two children for each internal node (with the root as a possible exception)
  - each edge labeled by a nonempty factor of S
  - no two edges out of a node beginning with the same character.

- Let L(v) denote the label of a node v, i.e., the concatenation, in order, of the strings labelling the edges on the path from the root to the node v.

- Key feature of suffix trees: L(i)=S[i..m] for each leaf i.

Example of a suffix tree

- For the string xabxac we can associate the following suffix tree:



- L(5)=ac, i.e., the suffix starting on position 5.

Suffix trees

- Question: Does a suffix tree always exist?
- Answer: No!

- If a suffix of S matches a prefix of another suffix, then we cannot construct a suffix tree obeying all the previous rules.

- Example: If we cut the last character from the string of the previous example, i.e., we take xabxa, then the suffix xa is also a prefix of the suffix xabxa. So, the path labelled by xa will not end at a leaf.

Suffix trees

- To avoid this problem, assume that the string S has a "*termination character*" that occurs only at the end of S (or insert one, if needed)

- By doing this we make sure that:

  - no suffix can appear as a prefix of another suffix

  - suffixes label complete paths leading from the root of the graph to the leaves

First application of suffix trees

Suffix trees can be used to solve the exact pattern matching problem:

1. Construct the suffix tree T for the text[1..n] (We'll discuss later how to do this in O(n) time)
2. Match the characters of pat along the unique path from the root:
   - The pattern pat occurs in the text at position j if and only if pat is a prefix of text[j..n]
   - That is, pat labels an initial part of the path from the root of the graph to the leaf labelled by j
   - The matching path is unique since the edges leaving from a node are labelled by strings beginning with different characters.
   - The time to match pat to a path is in O(m), with m=|pat|

First application of suffix trees

a) If pat can be fully matched, let k be the number of leaves below the path labelled by pat. Each of these leaves indicates the start position of an occurrence of pat, and they can be collected in time O(k).
   - Traverse the tree starting at the end of the matching path using, e.g., a depth-first approach, and gather all the encountered leaves.

b) If pat doesn't match completely, then pat does not occur in the text

- The total time complexity of the algorithm is O(n+m+k)

A naive approach to construct a suffix tree

- We start with a root and a leaf numbered 1, connected by an edge labeled S$.

- We introduce in the graph successively the suffixes S[i..m]$ with i increasing from 2 to m.

- To insert the suffix K_i = S[i..m]$, we follow the path from the root matching the characters of K_i until the first mismatch, at character K_i[j] (which is bound to happen since no suffix of S$ is a prefix of another suffix)
    a) If the matching cannot continue from a node, denote that node by w
    b) Otherwise the mismatch occurs in the middle of an edge, say (u,v), which has to be split

A naive approach to construct a suffix tree

- Suppose the mismatch occurs in the middle of an edge e = (u, v), and let the label of that edge be $a_1 \ldots a_l$

- If the mismatch occurred on character a_k, i.e., $a_k \neq K_i[j]$, then create a new node w, and replace the edge e by edges (u,w) and (w,v) labeled by $a_1 \ldots a_{k-1}$ and $a_k \ldots a_l$

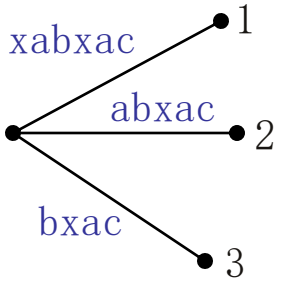
- Finally, in both cases (a) and (b), we create a new leaf numbered i, and connect w to it by an edge labelled by $K_i[j .. |K_i|]$

Example

- We want to build the suffix tree associated to the string S=xabxac

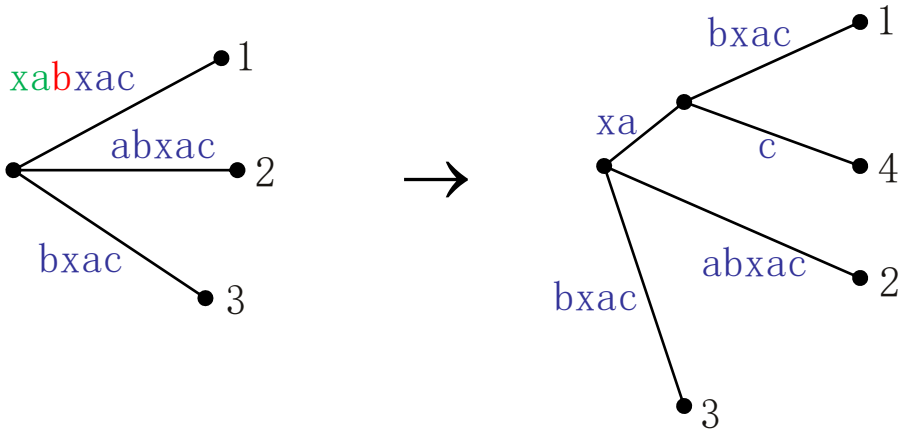
- We start with the graph:

- When we add the suffixes S[2..6]=abxac and S[3..6]=bxac, the mismatches occur in the root of the graph. So, we add 2 nodes and 2 edges:



Example (2)

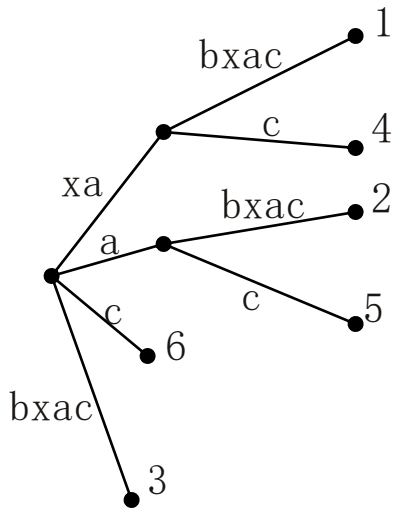
- Entering S[4..6] = xac causes the first edge to split:



- The same happens for the second edge when we introduce S[5..6] = ac in the graph

Example (3)

- After entering also the last suffix S[6]=c we obtain the suffix tree:



- Introducing a suffix S[i..m]$ in the tree is done in

$$\Theta(|\, S[i..m]\$\,|)$$

- So the total time complexity of the algorithm is

$$\Theta(\Sigma_{i=2}^{m+1} i) = \Theta(m^2)$$

Ukkonen's algorithm

- Ukkonen's algorithm constructs a suffix tree for the string S[1..m] in linear time (O(m)).

- We begin with a high-level description of the method, and then describe how to implement it to run in linear time.

Ukkonen's algorithm

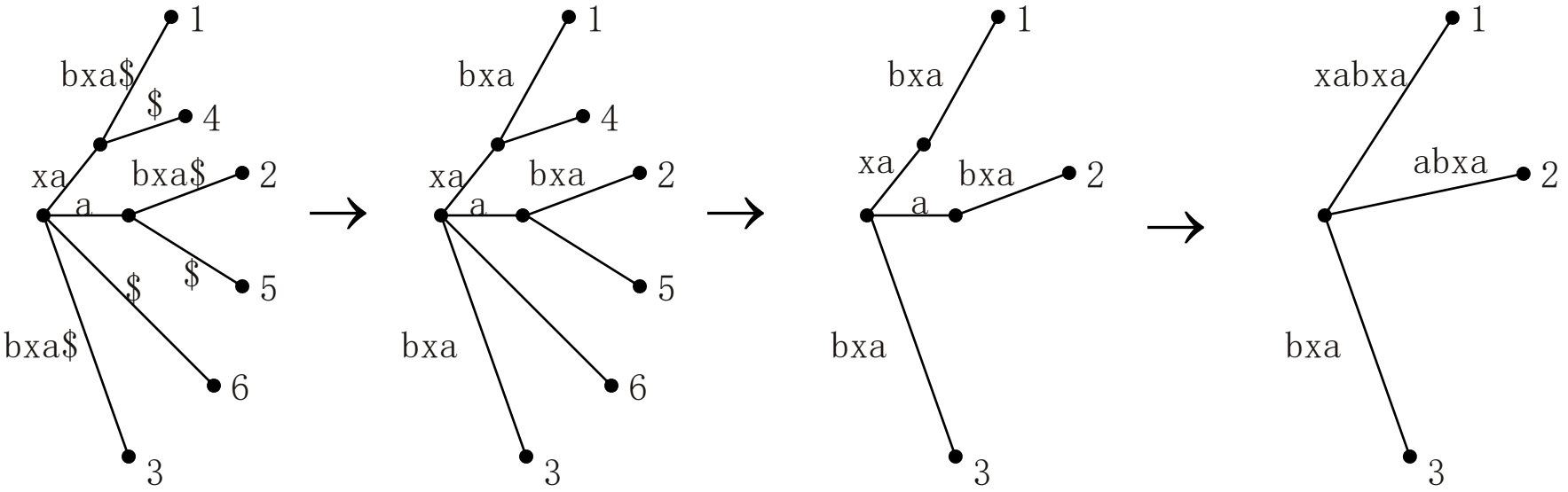
- The method builds, as intermediate results, *implicit suffix trees* for each *prefix* S[1]; S[1..2];..; S[1..m]

- The implicit suffix tree of a string S is obtained by constructing the suffix tree for the string S*$ and then remove:*
  - *All copies of the terminal symbol $ from the edge labels*
  - *All edges that have no labels*
  - *All nodes having only one child*

- All suffixes are included in an implicit suffix tree, but not necessarily as labels of complete paths leading to the leaves.

Example of implicit suffix tree

- Take the string S=xabxa having the following suffix tree.



1) Cut all $ from the edge labels

2) Cut all edges without labels

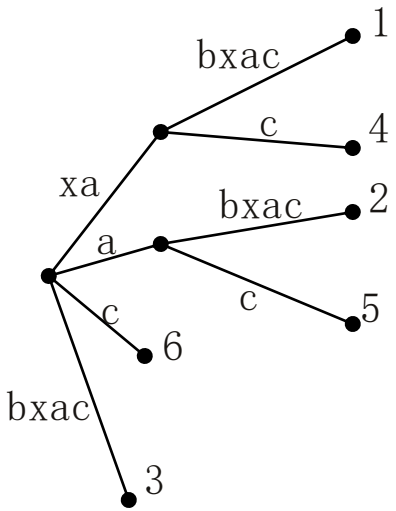
3) Cut all nodes having only one child

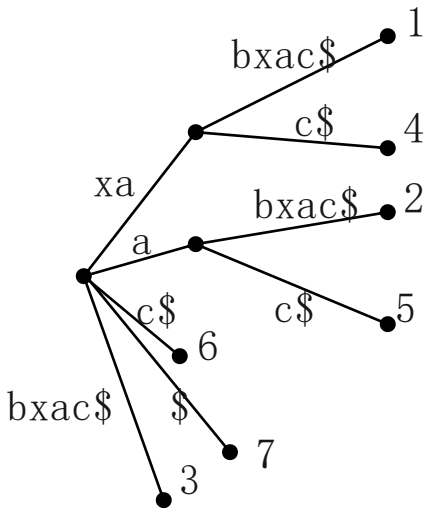
- some suffixes occur as labels of incomplete paths (not leading to a leaf, or even to an internal node)

Implicit suffix tree

- If S ends with a character that occurs also somewhere else in S, then the implicit suffix tree for S has fewer leaves than the suffix tree of S$.

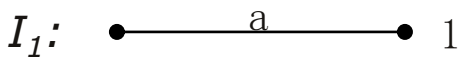
- If S ends with a character that does not occur anywhere else in S, then the implicit suffix tree is essentially the same as the suffix tree (only without $).





Implicit Suffix Trees of Prefixes

- Denote by I_j the implicit suffix tree of the prefix S[1..j]
- I_1 is just a single edge labeled by S[1] leading to leaf 1
- Example: The implicit suffix tree for the first prefix of the word axabxb:



Implicit Suffix Trees of Prefixes

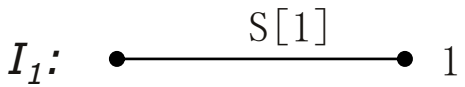
- I_j contains each suffix S[1..j]; S[2..j]; ... ; S[j] of S[1..j] as a label of some path (possibly ending in the middle of an edge)

- Let us call *(string) paths* the labels of such (partial) paths

- That is, a (string) path is
  - a string that can be matched along the edges, starting from the root, or equivalently
  - a prefix of any node label

Ukkonen's Algorithm on a High Level

- Given a string S[1..m], we start with $T := I_1$



- Then update T to trees $I_2,\ldots,I_{m+1}$ in m phases.
  - S[m+1] is the terminal symbol \$, so the final value of T is a true suffix tree, which contains all suffixes of S (extended with \$)

- Phase i+1 updates T from I_i (containing all suffixes of S[1..i]) to I_{i+1} (containing all suffixes of S[1..i+1])

- Each phase i+1 consists of extensions j = 1,..,i+1;

- Extension j ensures that suffix S[j..i+1] is in I_{i+1}

Extension j of Phase i + 1

- Phase i+1 starts with $T = I_{i,}$ containing all suffixes of S[1..i]

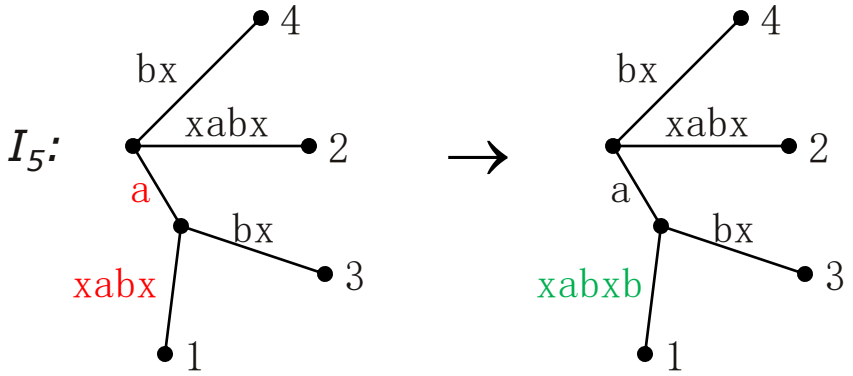
- For each $1 \leq j \leq i+1$, the algorithm finds the end of the path S[j..i] and extends it (if necessary) to ensure that the suffix S[j..i+1] is in the tree.

- Question: How do we extend the path?

- After locating the end of the path S[j..i], we have tree possible cases, and we have 1 extension rule for each case:

Suffix Extension Rules

- Consider phase 6 for the string S[1..6]=axabxb;
- Now $T=I_5$ contains all suffixes of S[1..5] = axabx:

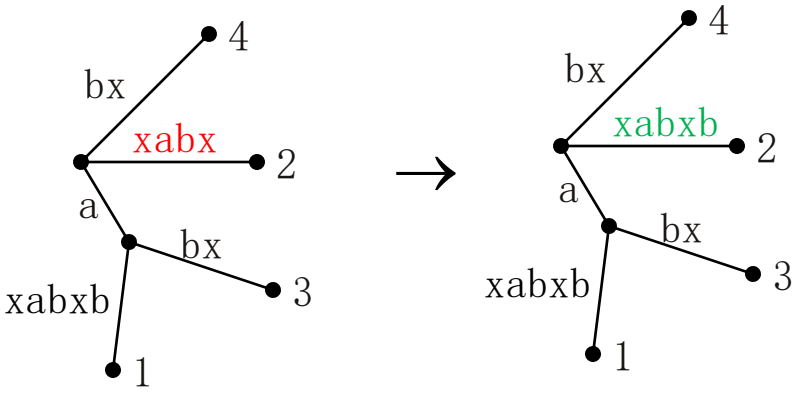


- Extension 1: we want to locate the end of string S[1..5]=axabx and then extend it by S[6]

- Rule 1: If in the current tree the path S[j..i] ends at a leaf.
  - Then we concatenate S[i+1] to its edge label.

Suffix Extension Rules

- $T=I_5$ contains all suffixes of S[1..5] = axabx:

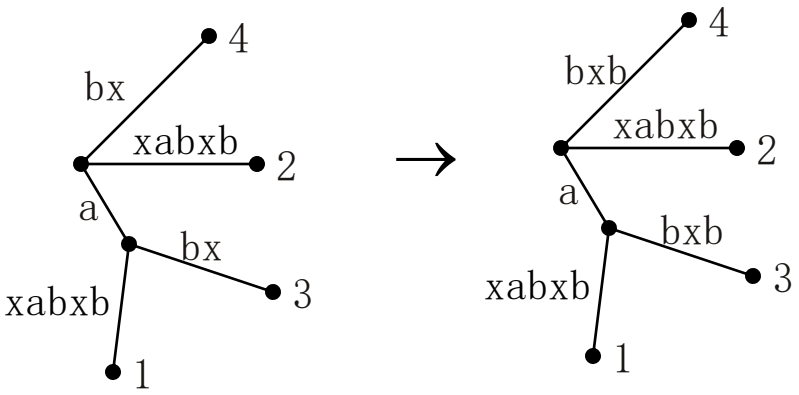


- Extension 2: we want to locate the end of string S[2..5]=xabx and then extend it by S[6]=b

- Rule 1: If in the current tree the path S[j..i] ends at a leaf.
  - Then we concatenate S[i+1] to its edge label.

Suffix Extension Rules

- $T=I_5$ contains all suffixes of S[1..5] = axabx:

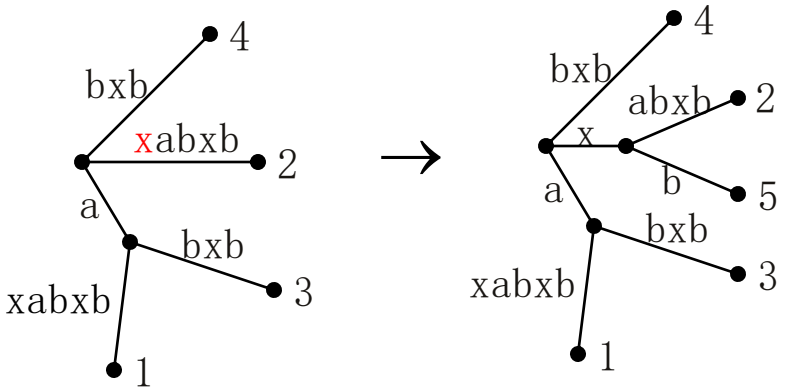


- Extensions 3,4 (for the strings S[3..5]=abx, S[4..5] =bx) are done similarly using Rule 1.

- Rule 1: If in the current tree the path S[j..i] ends at a leaf.
  - Then we concatenate S[i+1] to its edge label.

Suffix Extension Rules

- Extension 5: locate the end of S[5]=x and then extend it by S[6]=b,

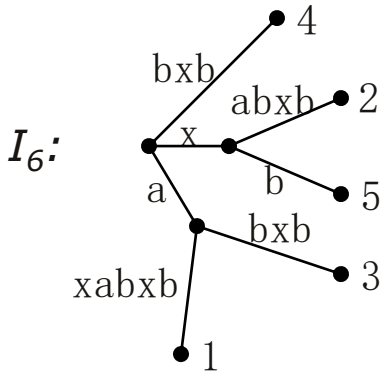


- create a node at the end of S[5] and a leaf labeled 5
- create an edge from the new node to the leaf labeled by S[6]=b

- Rule 2: If no path from the end of S[j..i] starts with character S[i+1], but there is at least one labeled path from the end of S[j..i].
  - Then, we create a new leaf labeled j, and an edge starting from the end of S[j..i] to the new leaf labeled by S[i+1].
  - If the path ended at the middle of an edge, we split the edge and insert a new node as a parent of leaf j.

Suffix Extension Rules

- Extension 6: we locate the end of S[6..5]=ε and then we extend it by S[6]=b



- Rule 3: Some path from the end of S[j..i] starts with S[i+1].
  - Then, we do nothing since the suffix S[j..i+1] is already in the tree.

Algorithm complexity

- The key issue in implementing Ukkonen's algorithm is locating the end of all the i+1 suffixes of S[1..i].

- Once the end of such a suffix is located, the execution of any of the extension rules takes constant time.
  - So, applying them once in each extension takes time $\Theta(i)$

- Locating the ends of the paths S[1..i];..; S[i+1..i] by traversing them explicitly takes time $\Theta(\Sigma_{l=0}^{i} l) = \Theta(i^2)$

- So, the total time for all phases i=2;..;m + 1 is

$$\Theta(\Sigma_{l=2}^{m+1} i^2) = \Theta(m^3)$$

Reducing the Complexity

- How to reduce this complexity?

- To get total time down to $\Theta(m^2)$ we need to avoid or speed up path traversals.

- Spending even constant time for each extension requires time $\Theta(\Sigma_{i=2}^{m+1} i) = \Theta(m^2)$

- To get total time down to $\Theta(m)$ we also need to avoid performing some extensions at all.

- We will consider first speeding up the path traversals.

Locating Ends of Paths

- The extensions of phase i+1 need to locate the ends of all the i+1 suffixes of S[1..i]

- How to do this efficiently?

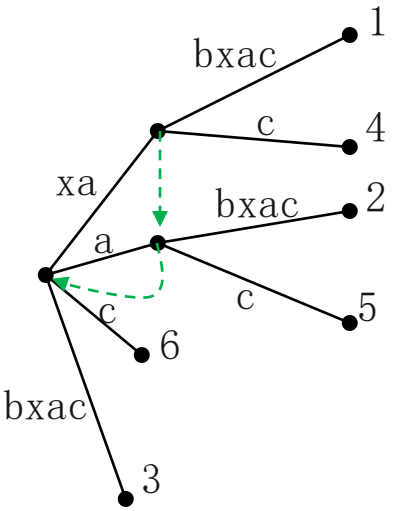
- Let $x\alpha$ denote an arbitrary string, where x is just a character while α denotes a (possibly empty) string.

- For an internal node v of T labeled by $x\alpha$, if there is another node s(v) labeled by α, then a pointer from v to s(v) is called a suffix link.

- NB: If node v is labeled by a single character (x), then $\alpha=\varepsilon$ and s(v) is the root

Example of Suffix Links

- In the graph below we have 2 suffix links
  - one from the node labeled xa to the one labeled a
  - one from the node labeled a to the root



- The root is not considered internal node so we do not have suffix links leaving from the root.

- What are suffix links good for?

Intuitive Motivation for Suffix Links

- Extension j (of phase i + 1) finds the end of the path S[j..i] in the tree (and extends it with char S[i + 1])
- Extension j+1 similarly finds the end of the path S[j+1..i]
- Let v be an internal node labeled by S[j]α on the path S[j..i]. Then we can avoid traversing path α when locating the end of path S[j+1..i], by starting from node s(v)
  - we know that if we have a suffix link from v to s(v), then s(v) is the node labeled by α
- Do suffix links always exist?
- Yes, and each suffix link (v,s(v)) is easy to set.

Computation of Suffix Links

- Observation: If an internal node v, labeled by $x\alpha$, is created during extension j (of phase i+1), then extension j+1 will identify the corresponding node s(v)
  - An internal node v can only be created by extension rule 2.
  - That is, v is inserted at the end of path S[j..i], which continued by some character c≠S[i+1].
  - In extension j+1, there is a path labeled α which certainly continues with character c, maybe also other characters.
    - If path α continues only with c, then extension rule 2 creates node s(v) at the end of path α.
    - Otherwise, i.e., the path α continues with at least 2 different characters , then node s(v) already exists in the tree

Computation of Suffix Links

- In Ukkonen's algorithm any newly created internal node will have a suffix link from it by the end of the next extension.
  - This is true for I_1, since I_1 does not have internal nodes
  - Suppose it is true through the end of phase i, and consider phase i+1
  - If a node v is created during extension j, the correct node s(v) ending the suffix link from v is either found or created during extension j+1 (see the previous Observation)
  - No new internal node is created in the last extension of a phase
  - Thus, for all internal nodes created during phase i+1, the suffix links starting from them are known by the end of this phase

Speeding up Path Traversals

- Consider the extensions of phase i+1:
  - Locate the suffixes S[j..i] with j from 1 to i+1
  - Extend them with character S[i+1]

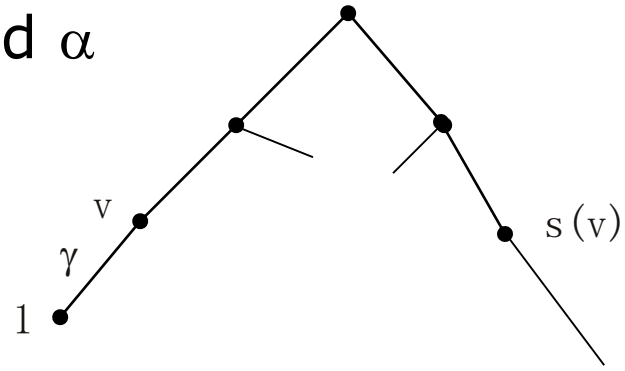
- Extension 1 extends path S[1..i] with char S[i+1]

- Path S[1..i] always ends at leaf 1, and is thus extended by Rule 1

- We can perform extension 1 in constant time, if we maintain a pointer to the node at the end of S[1..i]

Locating Subsequent Paths

- Let S[1..i]=$x\alpha$, where x is just a char, and α is a string
- Extension 2: find the end of S[2..i]=α
- Let (v,1) be the tree-edge entering leaf 1, labeled by γ
- If v is the root, then, to find the end of α, we just walk down the tree following the path labeled α

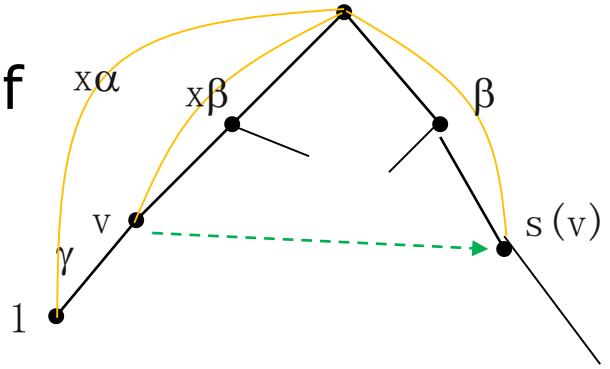


- Otherwise, v is an internal node,

and we know it has a suffix link to a node s(v) (see the earlier Observation)

Locating Subsequent Paths

- Since the label of node v is a prefix of $x\alpha$, the label of s(v) is a prefix of α
  - $\alpha=\beta\gamma$, $S[1..i]=x\alpha=x\beta\gamma$
- So, the end of the string α is somewhere in the subtree $s(v)$
  - To find the end of α, we do not need to walk the entire path from the root
- To find the end of α we need to:
  - Walk up from leaf 1 to node v
  - Follow the suffix link to node s(v)
  - Walk from the node s(v) down the path labeled by γ



- The green line depicts the suffix link from v to s(v)
- The orange lines only depict the string labeling of paths (they are neither tree-edges, nor suffix links)

Locating Subsequent Paths

- What about subsequent extensions j (for j=2;..;i)?
- Extension j-1 has located the end of the path S[j-1..i]
- Starting from there, walk up at most one node either
  - (a) to the root, or
  - (b) to an internal node v with a suffix link
- In case (a), traverse path S[j..i] explicitly down-wards from the root

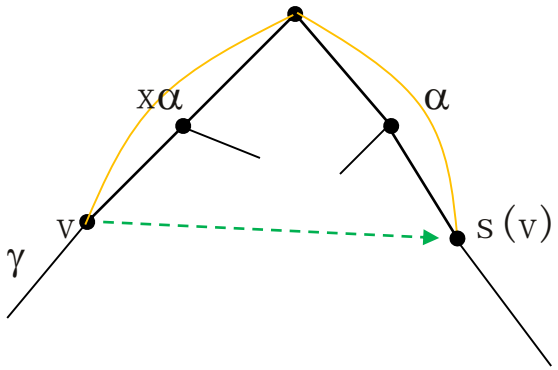
Short-cutting Traversals

- In case (b) let $x\alpha$ be the label of v and let γ be such that S[j-1..i]=$x\alpha\gamma$.

- Then follow the suffix link of v, and continue by matching γ down-wards from node s(v) (which is now labeled by α)

- Having found the end of path $\alpha\gamma$=S[j+1..i], apply extension rules to ensure that it extends with S[i+1]

- Finally, if a new internal node w was created in extension j, set its suffix link to point to the end node of path S[j+1..i]



Speeding up Explicit Traversals

- Using suffix links is definitely an improvement compared with walking-down from the root in each extension.

- Using only the suffix links is not enough to improve the time bound.

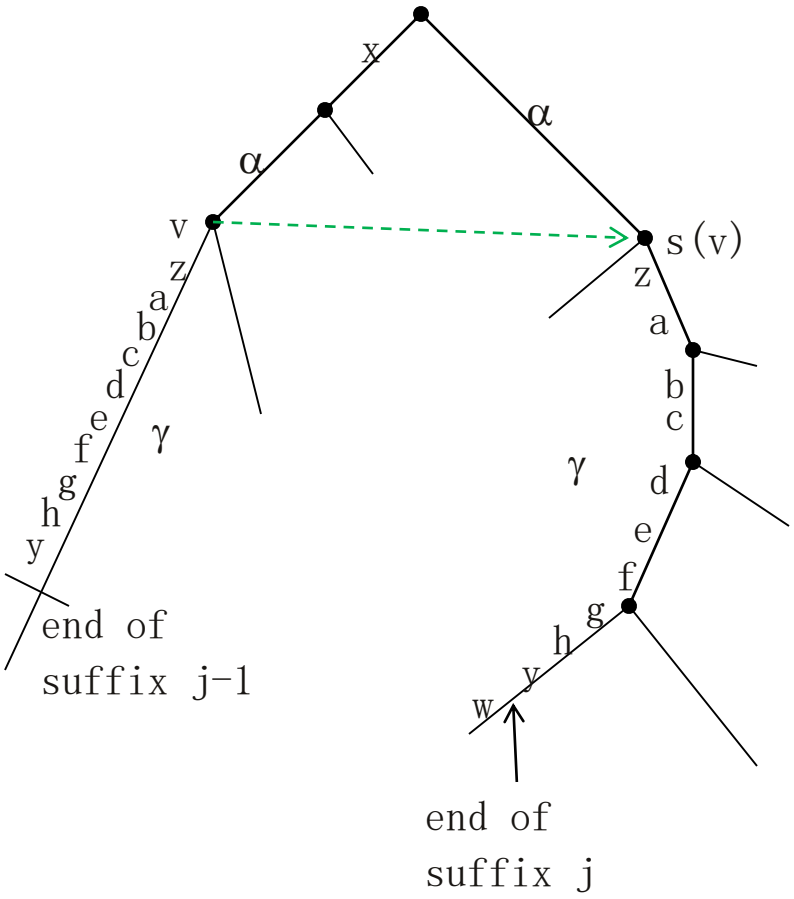
- We introduce next a trick that will reduce the worst-case complexity to $O(m^2)$.

Skip/count Trick

- In extension j:
  - Starting from the end of the string S[j-1..i] go up at most one edge, labeled by γ, to a node v
  - If v is internal node , then it has a suffix link to a node s(v)
  - Follow the suffix link from v to s(v)
  - Walk down from s(v) along a path labeled by γ
- Let $k=|\gamma|$.
- All edges leaving from s(v) start with distinct characters.
- So, the first character of γ occurs as the first character on exactly one edge out of s(v); let k' denote the length of its label.

Skip/count Trick

- k=|γ| and k'=the length of the edge leaving from s(v) and starting with the first letter of γ
  - Example: k=10, k'=2
- If k'<k, the algorithm simply skips to the node at the end of that edge, and continues in a similar way to find the next edge.
- If k'≥k, then the algorithm skips to the character k on the edge and stops, since the path labeled by γ ends inside that edge exactly k characters down its label.



Bounding the Time of Tree Traversals

- Thus, the time needed to traverse a path is proportional to the *node-length* of the path (instead of its string-length).

- Denote by depth(v) the node-depth of v, i.e., the number of nodes on the path from the root to v.

- Following a suffix link leads at most one level closer to the root.
  - For any node v with a suffix link to s(v), we have that depth(s(v))≥depth(v)-1

Linear Bound for any Single Phase

- Theorem: Using suffix links and the skip/count trick, a single phase takes time O(m).

Proof:

- There are $i+1 \leq m$ extensions in phase $i+1$

- In a single extension the algorithm does the following:
  - first moves at most one level up.
  - then a suffix link may be followed,
  - then it walks down a number of nodes,
  - applies an extension rule
  - maybe adds a suffix link

- In any extension, all other operations except tree-traversals take constant time.

Linear Bound for any Single Phase

- So, we only need to analyze the time for the down-walks.

- We do this by investigating how the current node-depth changes through the phase.

- In an extension, the possible up movement and suffix link traversal decrement current node depth at most twice. So, the current node depth is decremented at most 2m times during the entire phase.

- On the other hand, the current node depth cannot exceed m. So, the node-depth is incremented (by following downward edges) at most 3m times over the entire phase.

- Using the skip/count trick, the time per down-edge traversal is constant. Thus, the total time of a phase is O(m).

Algorithm complexity

- The total time of a phase is O(m).
- Since there are m phases, the total time of the algorithm is $O(m^2)$.
- A few more tricks are needed to get total time linear