Suffix Trees and Arrays

Chin Lung Lu

Computational Biology

Analyses and Applications of Sequences

Suffixes of a String

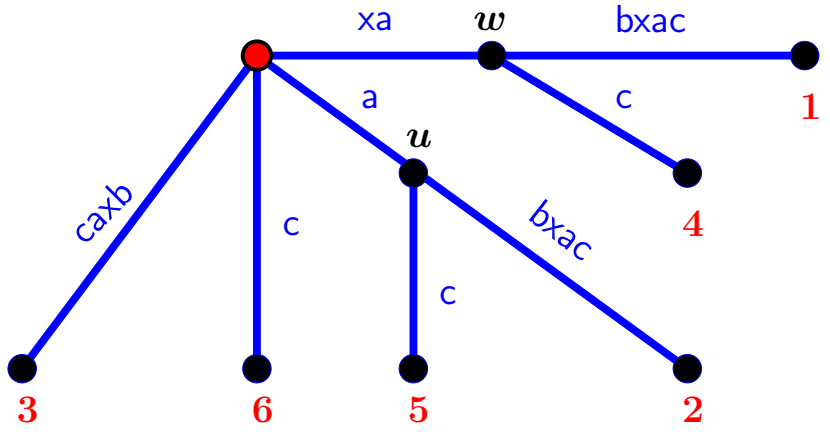
- For a string $S = S[1]S[2]\cdots S[m]$ of length m:
  - Each $S[1,i]$ is a prefix of S for $1 \le i \le m$.
  - Each $S[i,m]$ is a suffix of S for $1 \le i \le m$.
- **Example**: Let $S =$ xabxac.

$S[1,6]$	xabxac
$S[2,6]$	abxac
$S[3,6]$	bxac
$S[4,6]$	xac
$S[5,6]$	ac
$S[6,6]$	c

Suffix Tree $\mathcal{T}$ for S of Length m

1. A rooted tree with m leaves numbered 1 to m.
2. Each internal node, excluding the root, of $\mathcal{T}$ has at least 2 children.
3. Each edge of $\mathcal{T}$ is labeled with a nonempty substring of S.
4. No two edges out of a node can have edge-labels starting with the same character.
5. Suffix $S[i,m]$ corresponds to the concatenation of the edge-labels on the path from the root to leaf i.

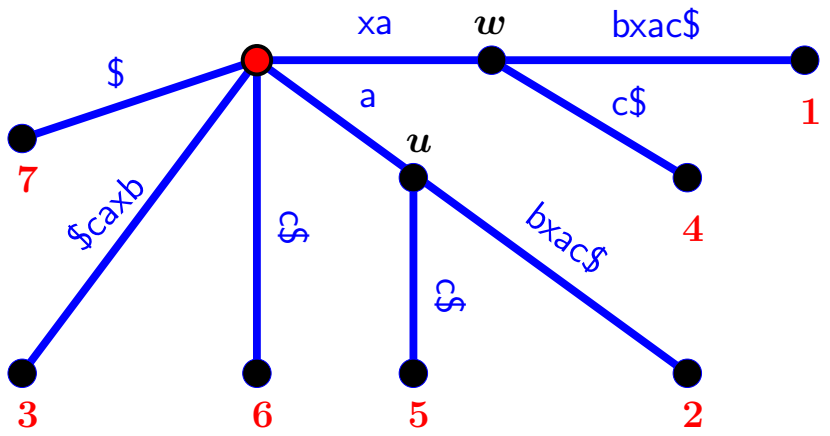
Example: Suffix Tree of xabxac



Existence of a Suffix Tree of S

- If one suffix S_j of S matches a prefix of another suffix S_i of S, then the path for S_j would not end at a leaf.
- For example, $S = xabxa$
  $S_1 = xabxa$ and $S_4 = xa$
- How to avoid this problem?
  - Assume that the last character of S appears nowhere else in S.
  - Add a new character \$ not in the alphabet to the end of S.

Example: Suffix Tree of xabxac\$



Application: Exact String Matching

- Given two strings T and P, where $|T| = m$ and $|P| = n$, find all the occurrences of P in T?

1. Build a suffix tree $\mathcal{T}$ for T in $\mathcal{O}(m)$ time
2. Match the characters of P along the unique path in $\mathcal{T}$ starting from the root until
   - P is exhausted: every leaf in the subtree below the point of the last match corresponds to a occurrence of P in T
   - No more matches are possible: P does not appear anywhere in T

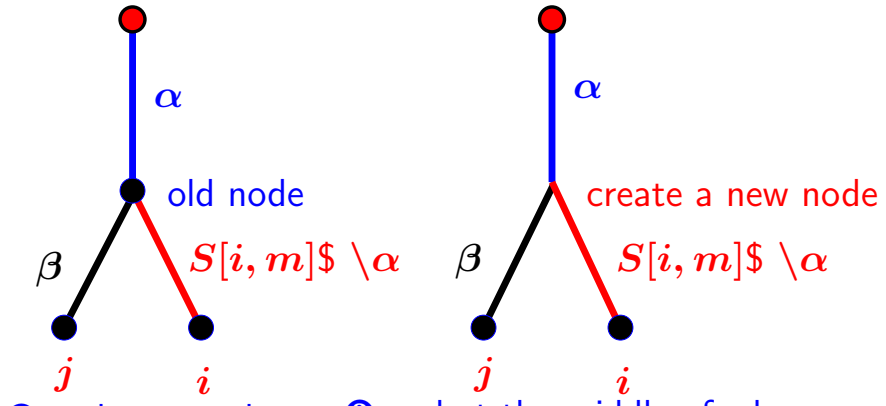
Naive Method: Suffix Tree of S\$

1. Create a single edge $\mathcal{T}_1$ with edge labeled S\$ and leaf labeled 1
2. **for** $i = 2$ to $m + 1$ **do** (where $m = |S|$)
   Construct $\mathcal{T}_i$ by adding $S[i, m]$\$ into $\mathcal{T}_{i-1}$
   end for
3. $\mathcal{T}_{m+1}$ is the suffix tree of S\$

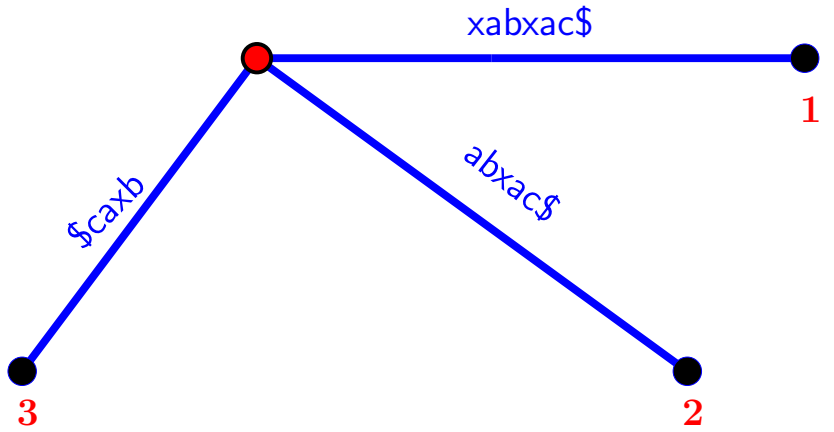
- **Time complexity**: $\mathcal{O}(m^2)$

How to Add $S[i,m]\$$ into $\mathcal{T}_{i-1}$?

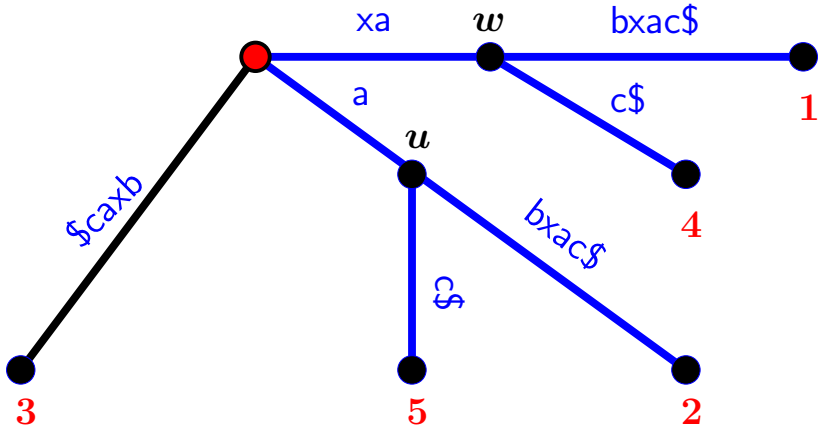
- Find the longest path from the root of $\mathcal{T}_{i-1}$ whose label matches a prefix of $S[i,m]\$$



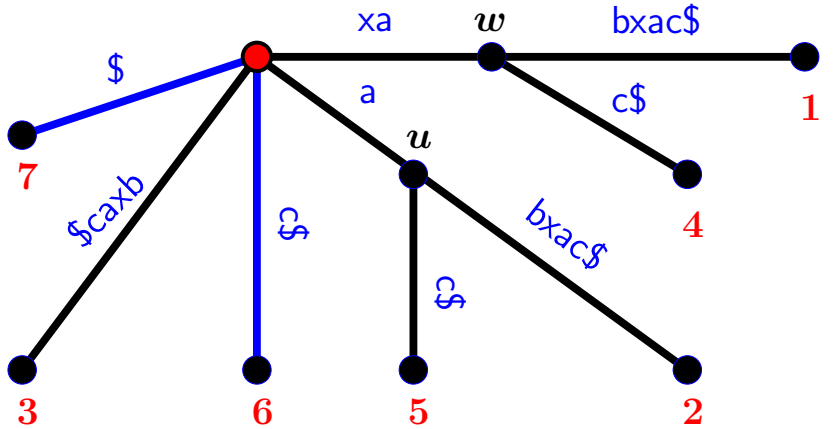
Steps 1–3: Suffix Tree of xabxac$



Steps 4–5: Suffix Tree of xabxac$



Steps 6–7: Suffix Tree of xabxac$



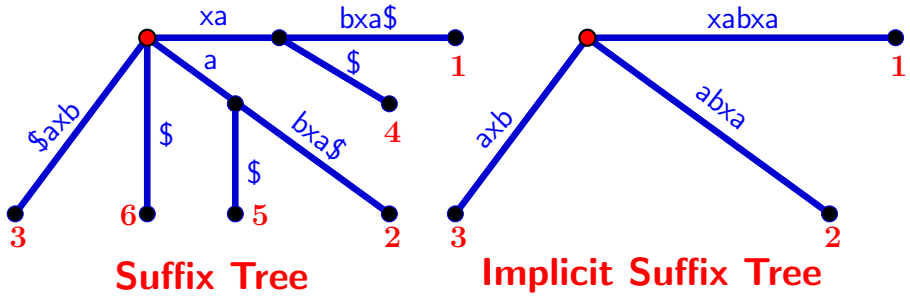
History of Linear-Time Suffix Tree

- Weiner's algorithm [Wei73] (FOCS, 1973)
  - "The algorithm of 1973" called by Knuth
  - First algorithm of linear time, but much space
- McCreight's algorithm [McC76] (JACM, 1976)
  - Linear time and quadratic space
  - More readable
- Ukkonen's algorithm [Ukk95] (Algorithmica, 1995)
  - On-line algorithm of linear time and less space

Implicit Suffix Tree

A tree obtained from the suffix tree $\mathcal{T}$ of $S\$$ by

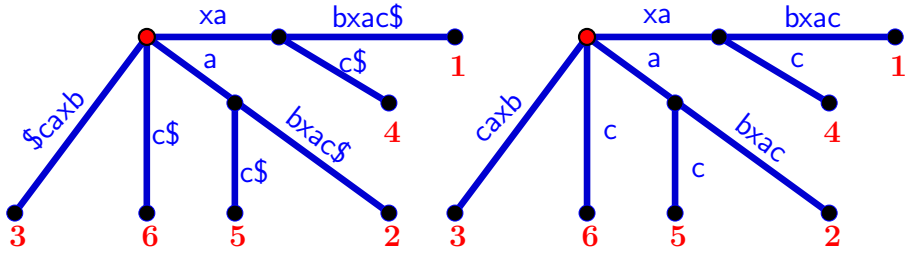
1. removing \$ from the edge labels of $\mathcal{T}$,
2. removing any edge without label, and
3. removing any node with degree 2



- Each suffix is in the tree, but may not end at a leaf.

Implicit Suffix Tree

- If S ends with a character that appears nowhere else in S then the implicit suffix tree of S is a true suffix tree of S.



Suffix Tree = Implicit Suffix Tree

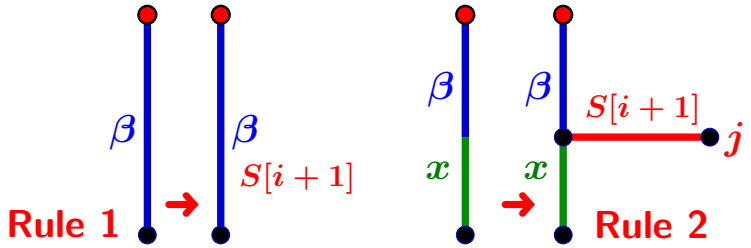
Ukkonen's Algorithm

- $\mathcal{I}_i$: the implicit suffix tree of the string $S[1,i]$

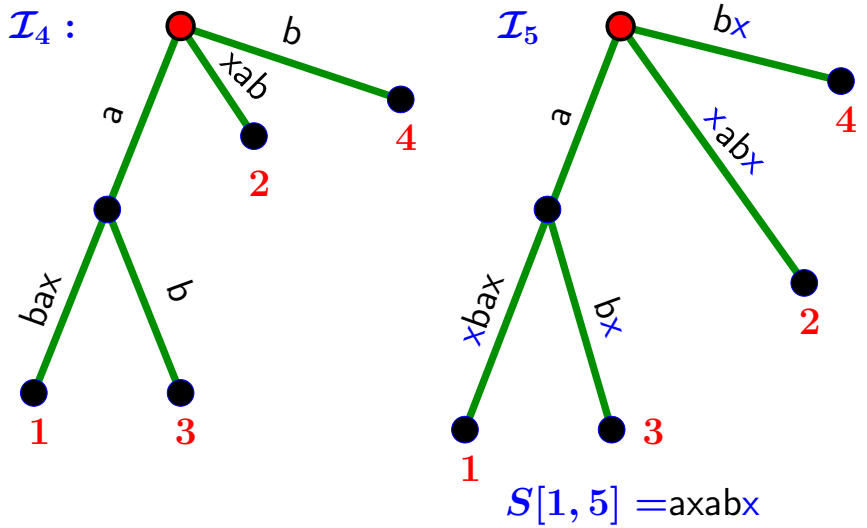
1. Construct $\mathcal{I}_1$;
2. **for** $i = 1$ to $m-1$ **do** /* Phase $i+1$ */
   for $j = 1$ to $i+1$ **do** /* Extension j */
   /* Construct $\mathcal{I}_{i+1}$ from $\mathcal{I}_i$ */
   Find the end of the path P from the root whose label is $S[j,i]$ in $\mathcal{I}_i$ and extend P with $S[i+1]$ by suffix entension ruls;
   end for
   end for
3. Convert $\mathcal{I}_m$ into a suffix tree $\mathcal{S}$;

Suffix Entension Rules

- **Goal**: to extend each $S[j,i]$ into $S[j,i+1]$
- **Rule 1**: Path β (i.e., $S[j,i]$) ends at a leaf
- **Rule 2**: Path β does not end at a leaf and the continued character $x \neq S[i+1]$
- **Rule 3**: Some path from the end of β starts with $S[i+1]$ (do nothing since $S[j,i+1]$ is in $\mathcal{I}_i$)

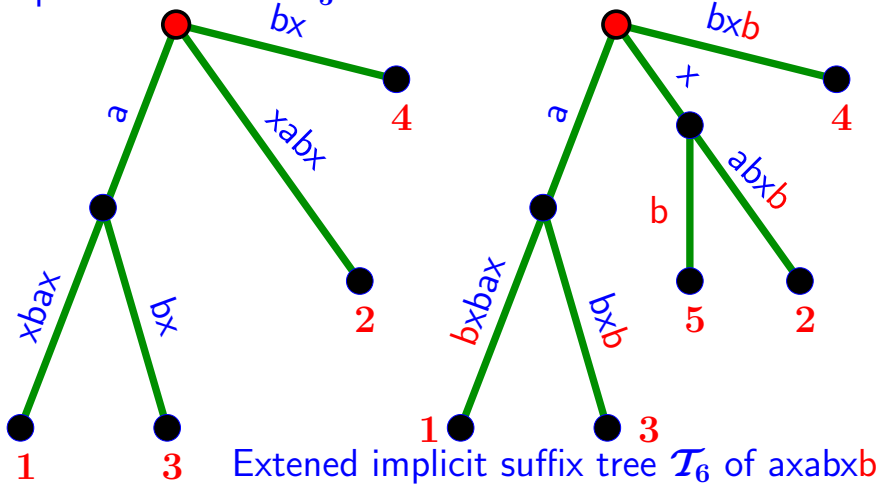


Example 1: Rules 1 & 3



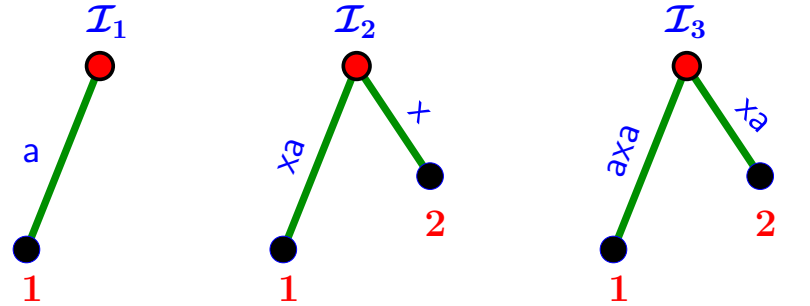
Example 2: Rules 1 & 2

Implicit suffix tree $\mathcal{T}_5$ of axabx

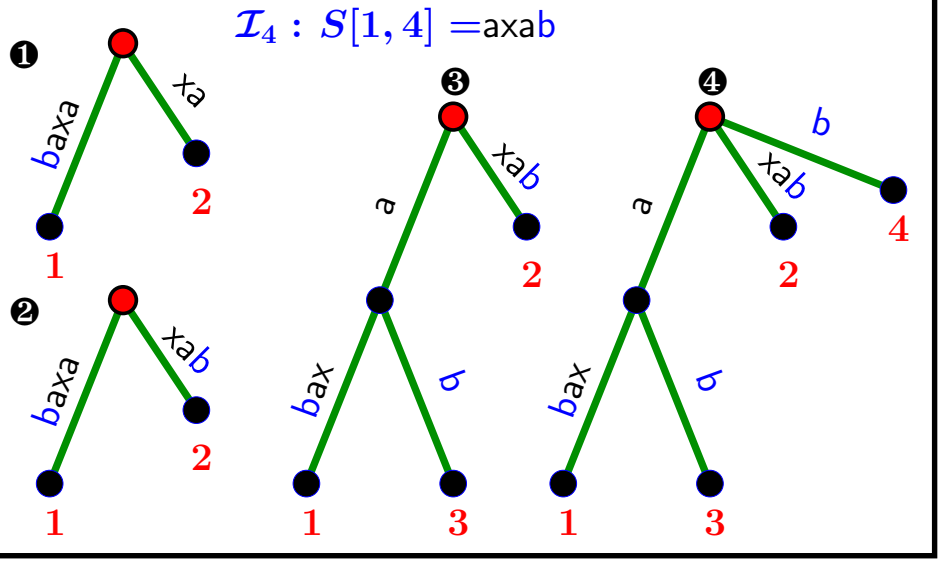


Extened implicit suffix tree $\mathcal{T}_6$ of axabxb

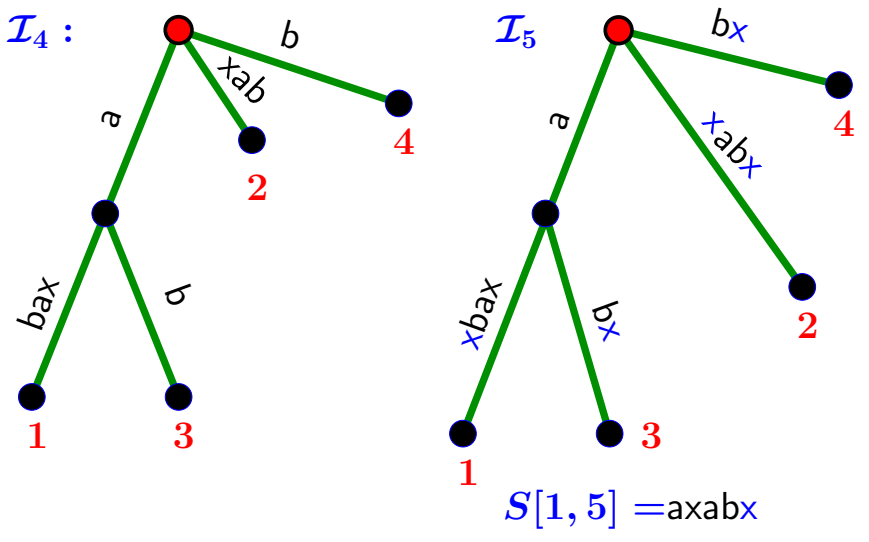
Ukkonen's Algorithm for axabxc ❶



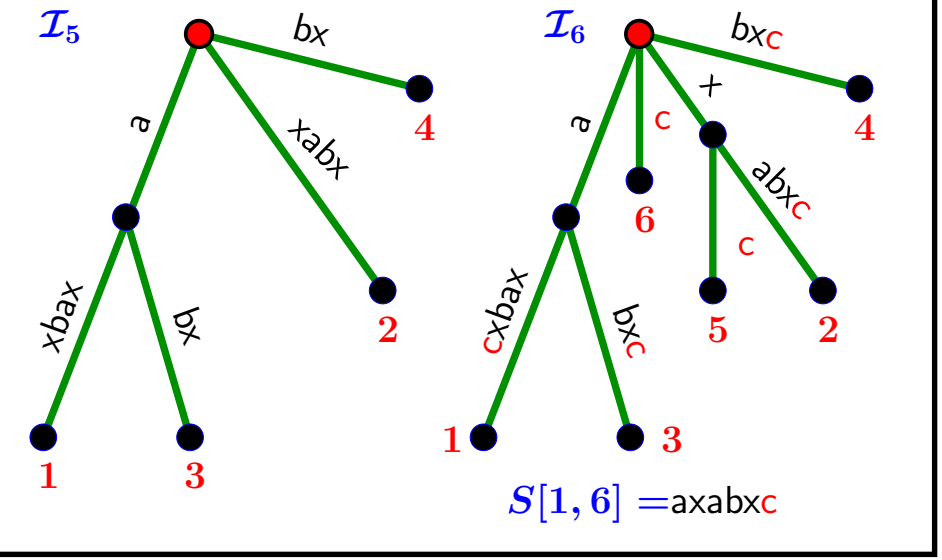
Ukkonen's Algorithm for axabxc ❷



Ukkonen's Algorithm for axabxc ❸



Ukkonen's Algorithm for axabxc ❹

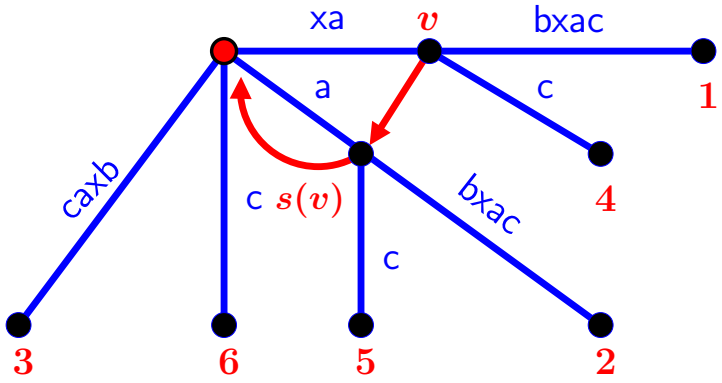


Key Issue to Ukkonen's Algorithm

- Once the end of a suffix β of $S[1,i]$ has been found in the current implicit suffix tree, only constant time is needed to execute the extension rules.
- The key issue in implementing Ukknonen's algorithm is how to locate the ends of all the $i+1$ suffixes of $S[1,i]$.
- Naive: Constructing $\mathcal{I}_{i+1}$ from $\mathcal{I}_i$ costs $\mathcal{O}(i^2)$
  $\therefore$ After constructing $\mathcal{I}_m$, total cost is $\mathcal{O}(m^3)$.

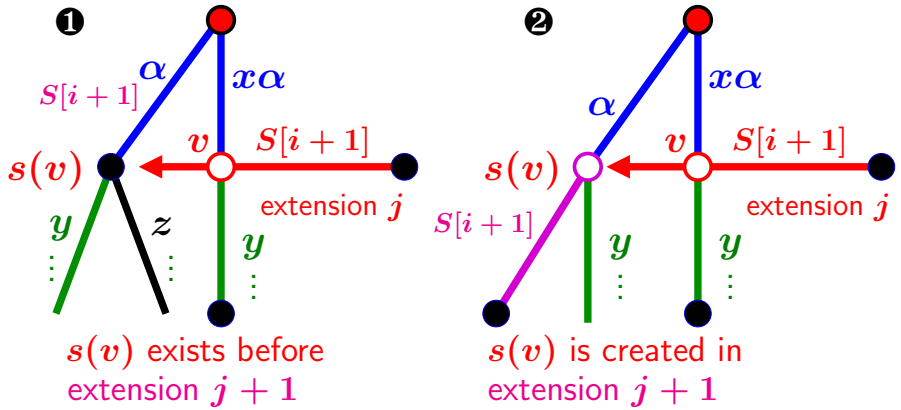
Suffix Link

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



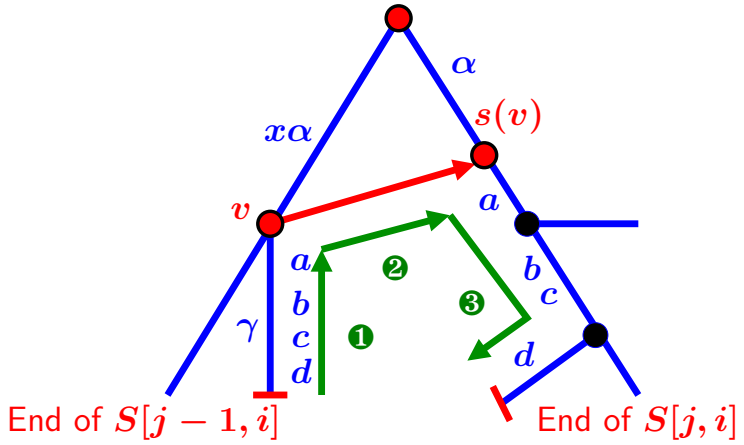
Suffix Link

- Every internal node of an implicit suffix tree has a suffix link from it. (By induction on $\mathcal{I}_i \longrightarrow \mathcal{I}_{i+1}$)



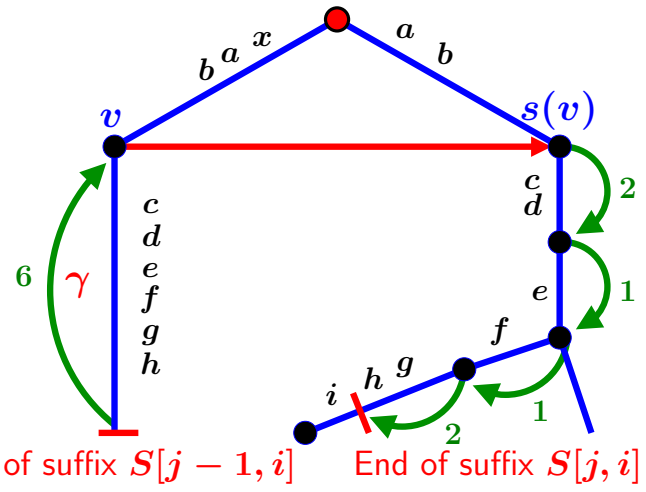
Build $\mathcal{I}_{i+1}$ via Suffix Links

- Extend $S[j,i]$ to $S[j,i+1]$: locate the end of $S[j,i]$ from the end of $S[j-1,i]$ via suffix link



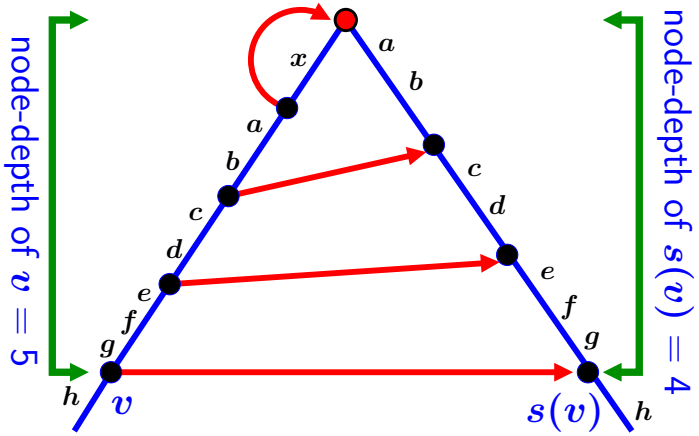
Trick 1: Skip/Count ❶

- Reduce the time of traversing the γ path from $s(v)$



Trick 1: Skip/Count ❷

- The node-depth of v is at most one greater than the node-depth of $s(v)$.



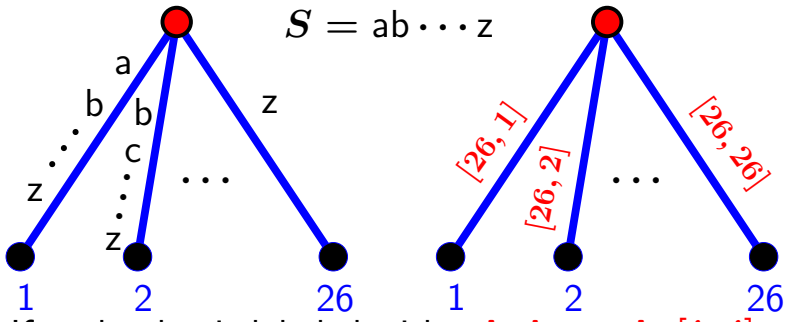
Trick 1: Skip/Count ❸

Using the skip and count trick, any phase of Ukkonen's algorithm takes $\mathcal{O}(m)$ time.

- Phase $i+1$ for $\mathcal{I}_{i+1}$ has $i+1 \leq m$ extensions.
- Totally, the current node-depth is decreased by $\leq 2m$, since for each extension j,
  - Up-walk: decreases this depth by ≤ 1
  - Suffix link traversal: decreases this depth by ≤ 1
- The total possible increment to the current node-depth is $\leq 3m$ over the entire phase.
  - No node has depth $> m$ ($\because$ path root$\rightarrow$leaf corresponds to a suffix, whose length $\leq m$).

Trick 2: Edge-Label Compression

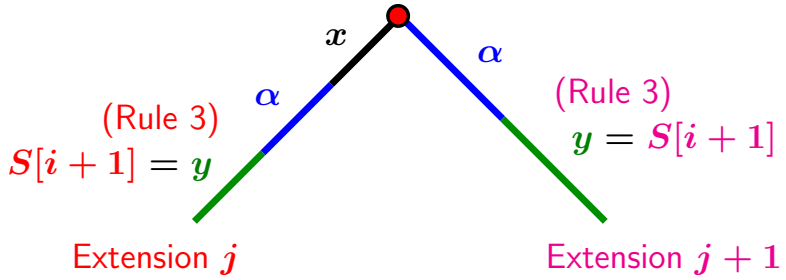
- If the edges are lebeled with substrings, the suffix tree may require $\Theta(m^2)$ space.



- If each edge is labeled with a **index pair** $[i, j]$, denoting substring $S[i, j]$, the suffix tree requires only $\mathcal{O}(m)$ space ($\because$ # of edges $\leq 2m - 2$).

Trick 3: Rule 3 Is a Stopper

- In any phase, if suffix extension rule 3 applies in extension j, it will also apply in all extensions k, where $k > j$, until the end of the phase.



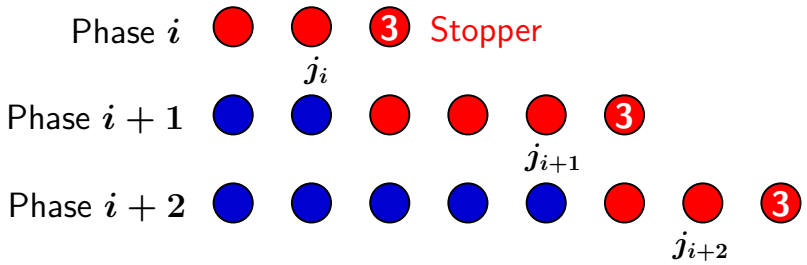
- Hence, we can end any phase when the first extension rule 3 applies.

Trick 4:Once a Leaf, Always a Leaf

- In any phase i, there is an initial sequence of consecutive extensions (starting with rule 1) in which only rule 1 or 2 applies.
  - j_i: the last extension in this sequence
- **Observation**: $j_i \leq j_{i+1}$
- In phase $i+1$, two extensions are considered:
  1. Implicit extensions: extensions 1 to j_i (write $[\cdot, e]$ on the leaf edge for these extensions, where e is set to $i+1$ once at the beginning)
  2. Explicit extensions: extension $j_i + 1$ to the first extension where rule 3 applies

Complexity: Ukkonen's Algorithm

- Since all implicit extensions in any phase are done by increasing e that costs a constant time, their total cost in all phases is $\mathcal{O}(m)$.
- Totally, $\leq 2m$ explicit extensions are executed.



(●: explicit, ●: implicit)

Complexity: Ukkonen's Algorithm

- The cost of each explicit extension is a constant (possible up-walk and suffix-link traversal) plus some time proportional to the number of node skips it does during the down-walk in that extension.
- Since no node has depth $> m$ and $\leq 2m$ explicit extensions are executed, the total number of node skips done during all the down-walks is bound by $5m = \mathcal{O}(m)$.
- Time-complexity of Ukkonen's algorithm: $\mathcal{O}(m)$

Application: Finding Substrings

Input: a database $\mathcal{D}$ (a set of strings) and a string P

Output: find all the strings in $\mathcal{D}$ containing P as a substring

- Usage: identity of the remains of a person
- KMP and Boyer-Moore methods: work inefficiently
- Generalize suffix tree (a suffix tree representing all the suffixes of a set of strings):
  - $\mathcal{D}$ is stored in $\mathcal{O}(|\mathcal{D}|)$ space.
  - Each lookup of P costs $\mathcal{O}(|P|)$ time.

Application: LCS

Longest Common Substring Prolbem

- **Input**: two strings S_1 and S_2
- **Output**: find the longest substring S common to S_1 and S_2
- **Example**:

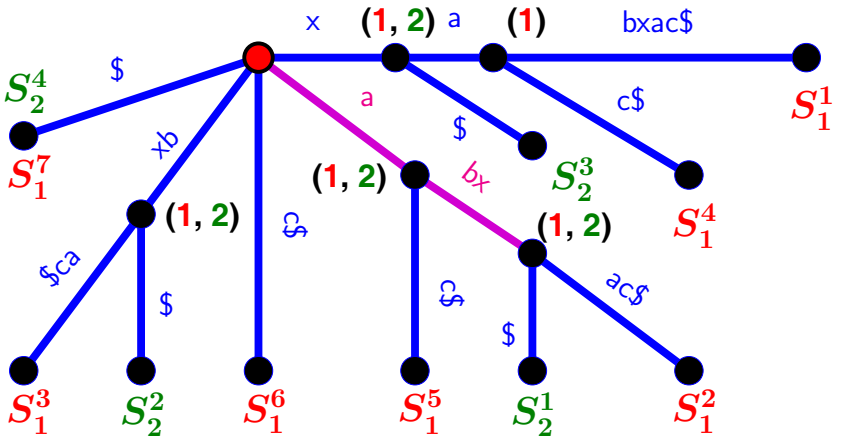
$$\begin{aligned} S_1 &= \text{common-substring} \\ S_2 &= \text{common-subsequence} \\ S &= \text{common-subs} \end{aligned}$$

Longest Common Substring

1. Build a generalized suffix tree for S_1 and S_2.
2. Each leaf represents either a suffix from one of S_1 and S_2, or a suffix from both S_1 and S_2.
3. Mark each internal node v with a 1 (resp. 2) if there is a leaf in the subtree of v representing a suffix from S_1 (resp. S_2).
4. The path-label of any internal node marked both 1 and 2 is a substring common to both S_1 and S_2, and the longest such string is the longest common substring.

Longest Common Substring

- $S_1\$ = \text{xabxac\$}$, $S_2\$ = \text{abx\$}$, $S = \text{abx}$



Application: DNA contamination

- **Input**: a string S_1 (the newly isolated and sequenced string of DNA) and a known string S_2 (the combined sources of possible contamination)
- **Output**: find all substring of S_2 that occur in S_1 and that are longer than some given length l
- To know whether the DNA of interest has been contaminated
- Possible contaminats: cloning vectors, PRC primers, the complete genomic sequence of the host organism etc.

Suffix Array

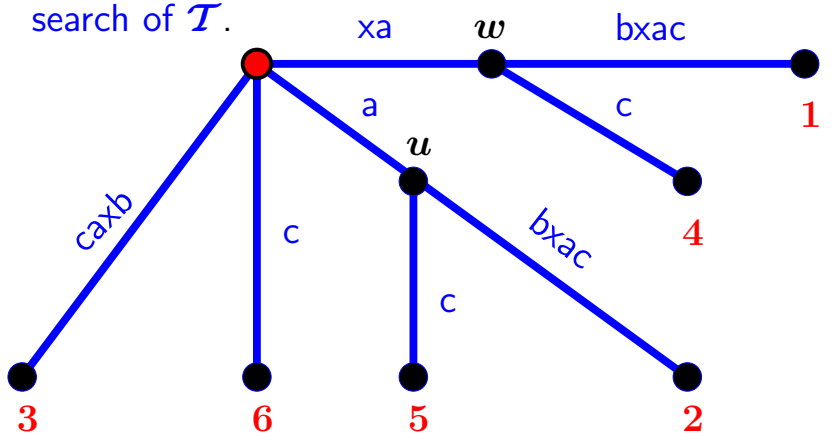
- A suffix array of an m-character string S, is an array of integers in the range 1 to m, specifying the lexicographic order of the m suffixes of S.
- **Example**: let S = xabxac

S_2	abxac
S_5	ac
S_3	bxac
S_6	c
S_1	xabxac
S_4	xac

index	1	2	3	4	5	6
Suffix array	2	5	3	6	1	4

How to Build Suffix Array?

- A suffix array of S can be obtained from the suffix tree $\mathcal{T}$ of S by performing a lexical depth-first search of $\mathcal{T}$.



Suffix Array:Exact String Matching

- Given two strings T and P, where $|T| = m$ and $|P| = n$, find all the occurrences of P in T?
- Using the binary search on the suffix array of T, all the occurrences of P in T can be found in $\mathcal{O}(n \log m)$ time.
- **Example**: let T = xabxac and P = ac

index	1	2	3	4	5	6
Suffix array	2	5	3	6	1	4

References

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

[Ukk95] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, 1995.

[Wei73] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory (old name of FOCS)*, pages 1–11, Washington, DC, 1973.