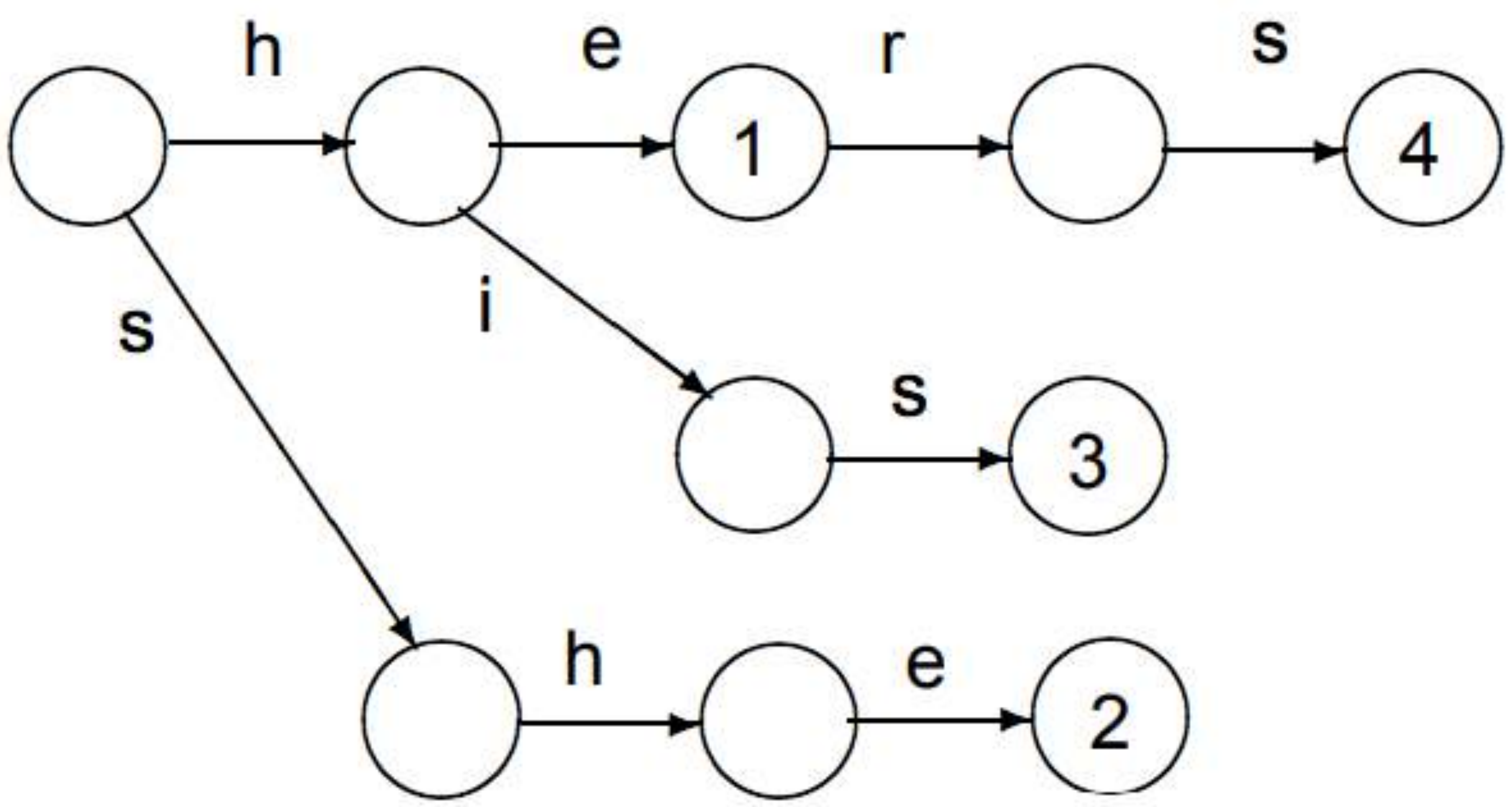
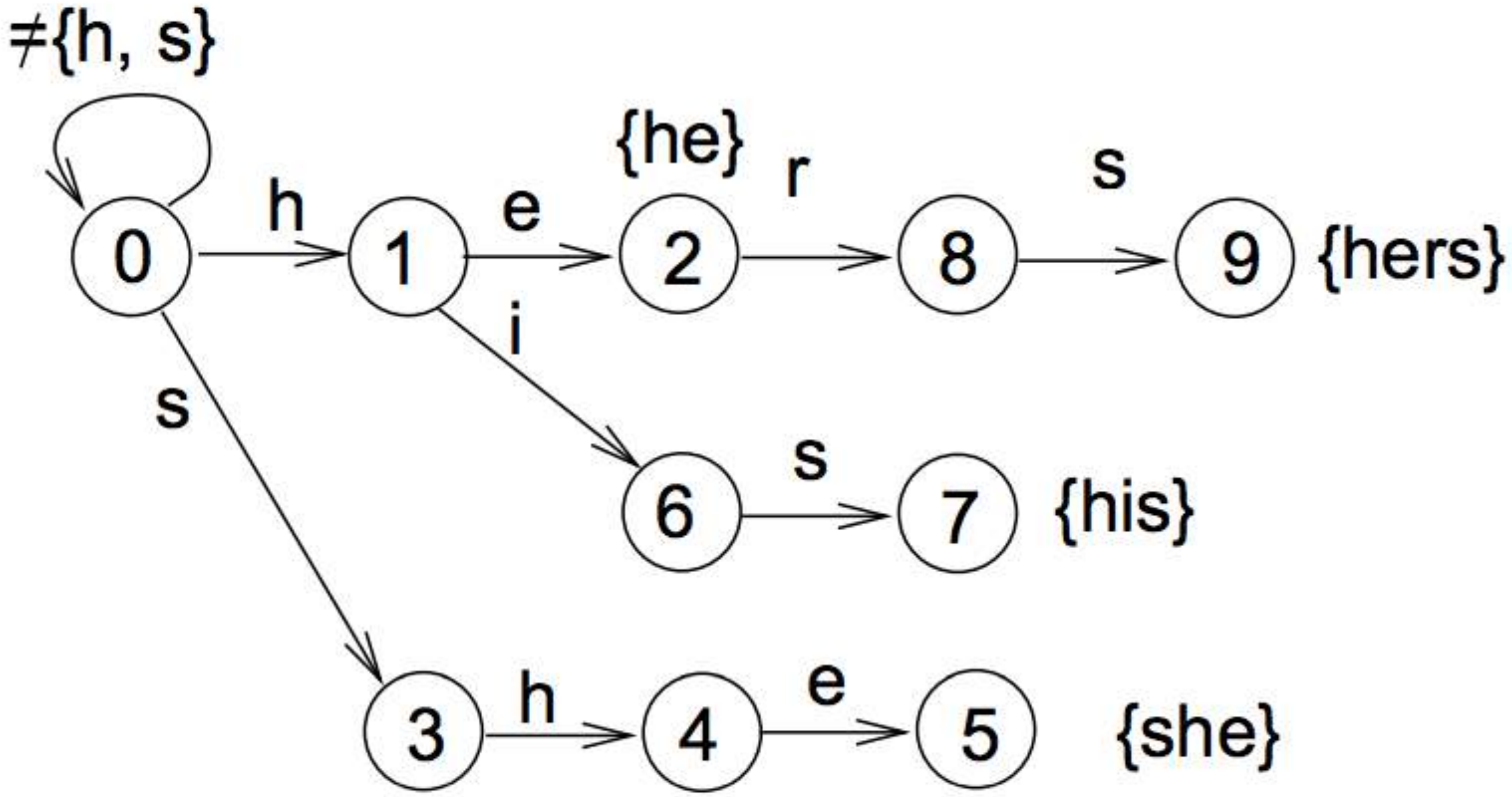
Aho-Corasick algorithm

A keyword tree for $\mathcal{P} = \{\text{he}, \text{she}, \text{his}, \text{hers}\}$:



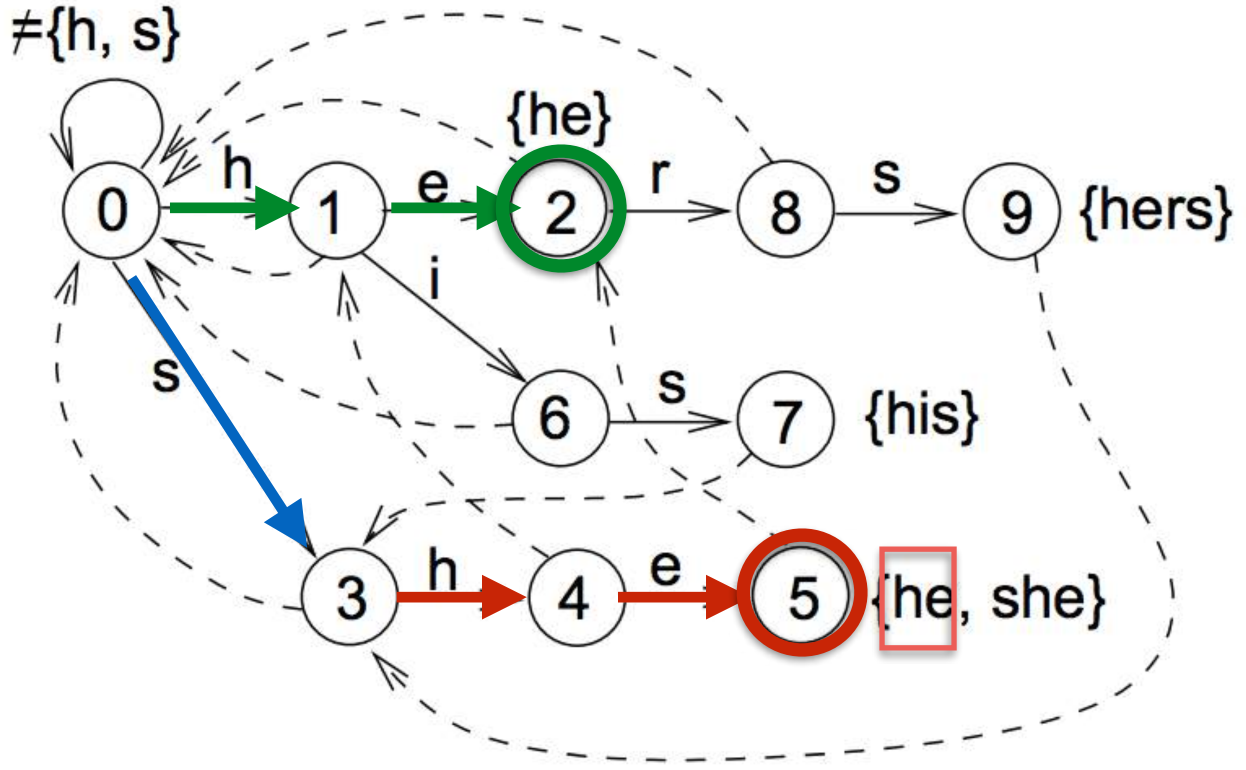
Aho-Corasick algorithm

Add pattern labels



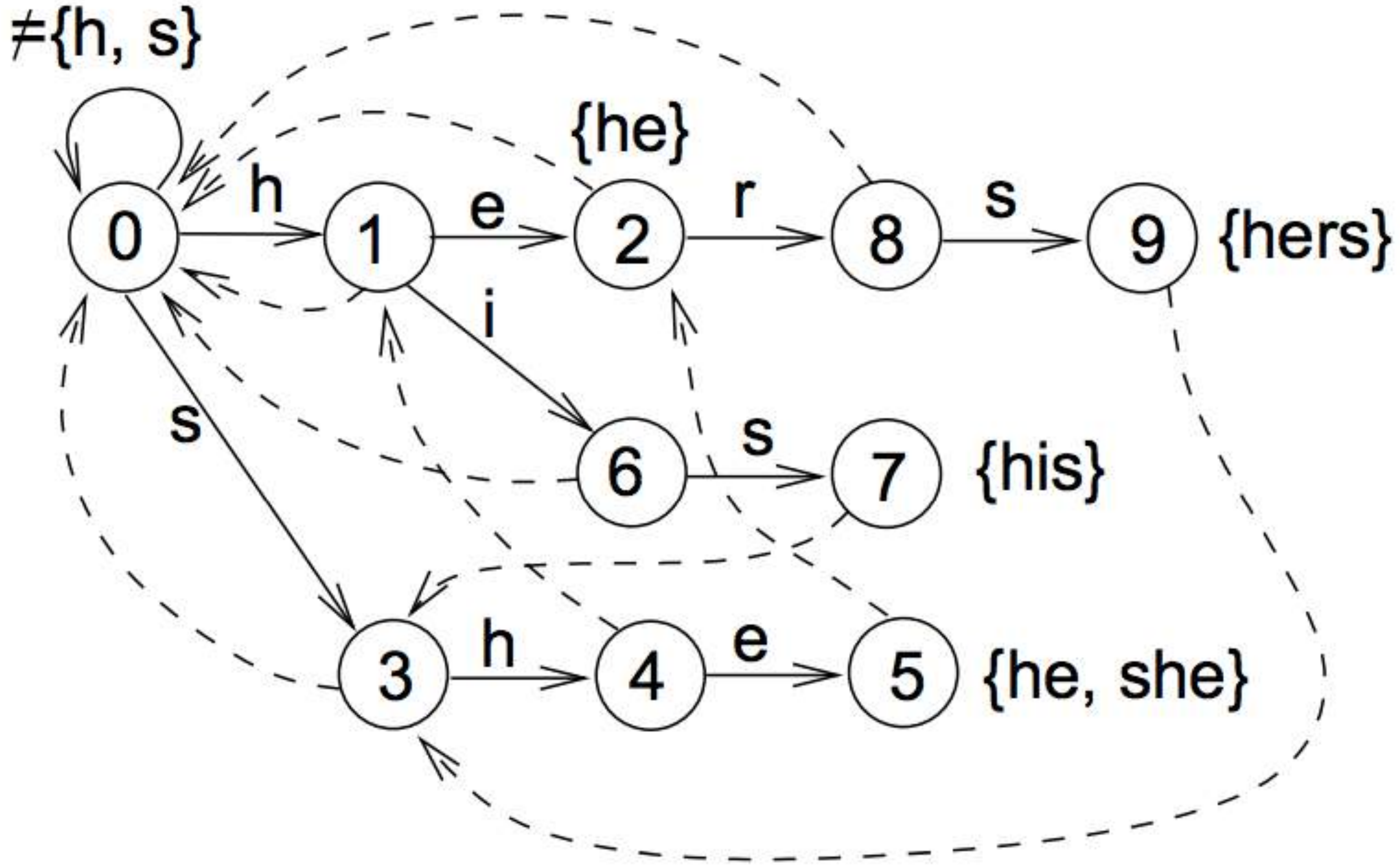
Adding failing edges

- If currently at node q representing word L(q), find the longest proper suffix of L(q) that is a prefix of some pattern, and go to the node representing that prefix. Insert the labels of the pointed node (if there is any) to node q's set of labels.
- Example: node q = 5, L(q) = she; longest proper suffix that is a prefix of some pattern: "he". Dashed edge to node q'=2



Aho-Corasick Algorithm

Add Failing Edges and Labels



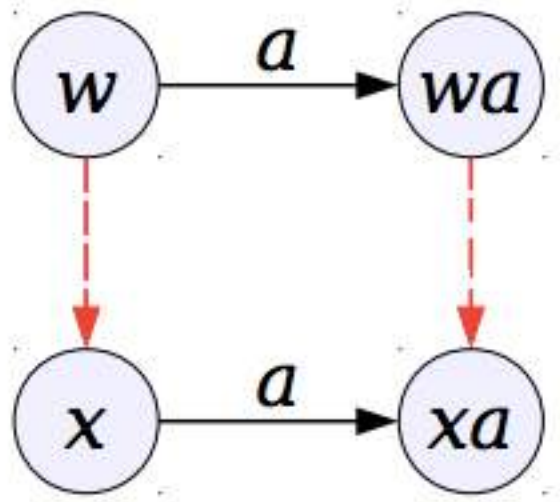
Aho-Corasick Algorithm: Construction

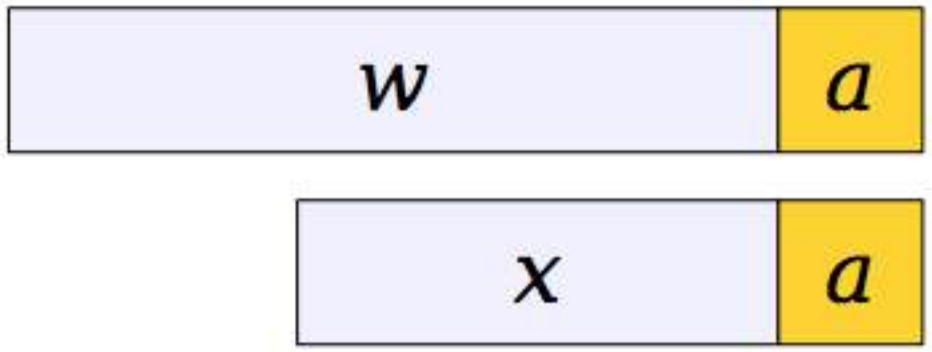
What about a naive algorithm?

A better algorithm: intuition

Suppose we already know the failing edge from a node **w** to **x**. If we follow a solid edge with label **a**, there are two possibilities:

- ***Case 1:*** *xa* exists.

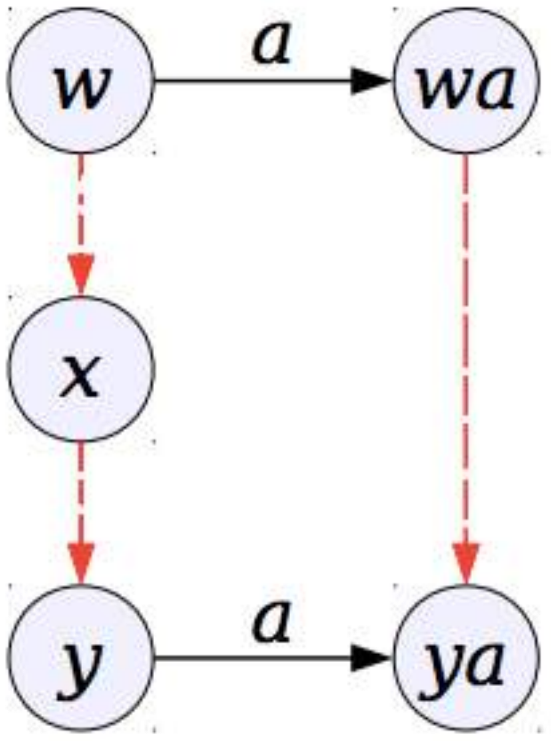


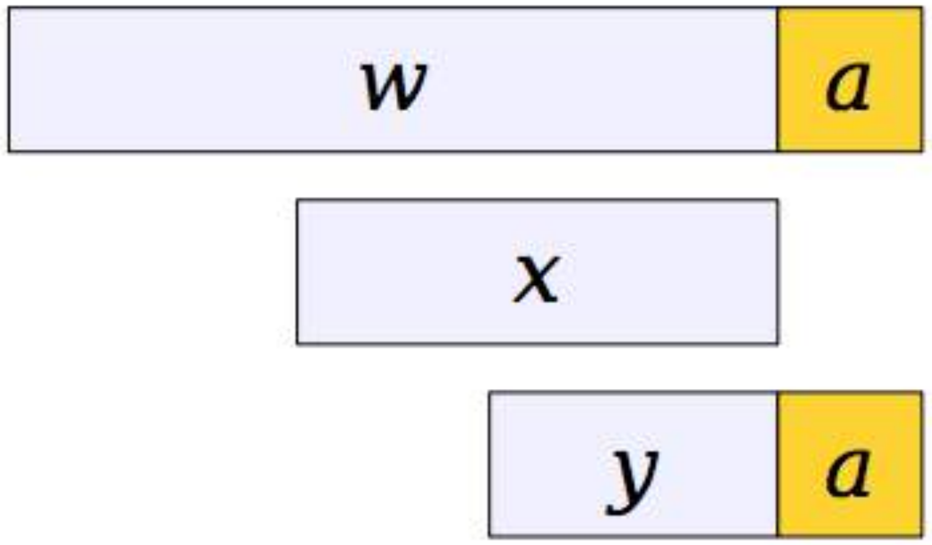


A better algorithm: intuition

Suppose we already know the failing edge from a node **w** to **x**. If we follow a solid edge with label **a**, there are two possibilities:

- ***Case 2:*** *xa* does not exist.

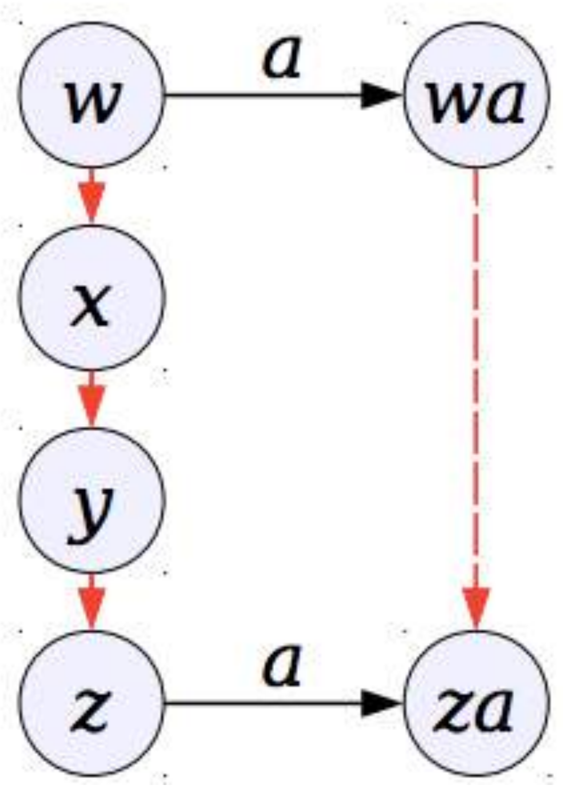


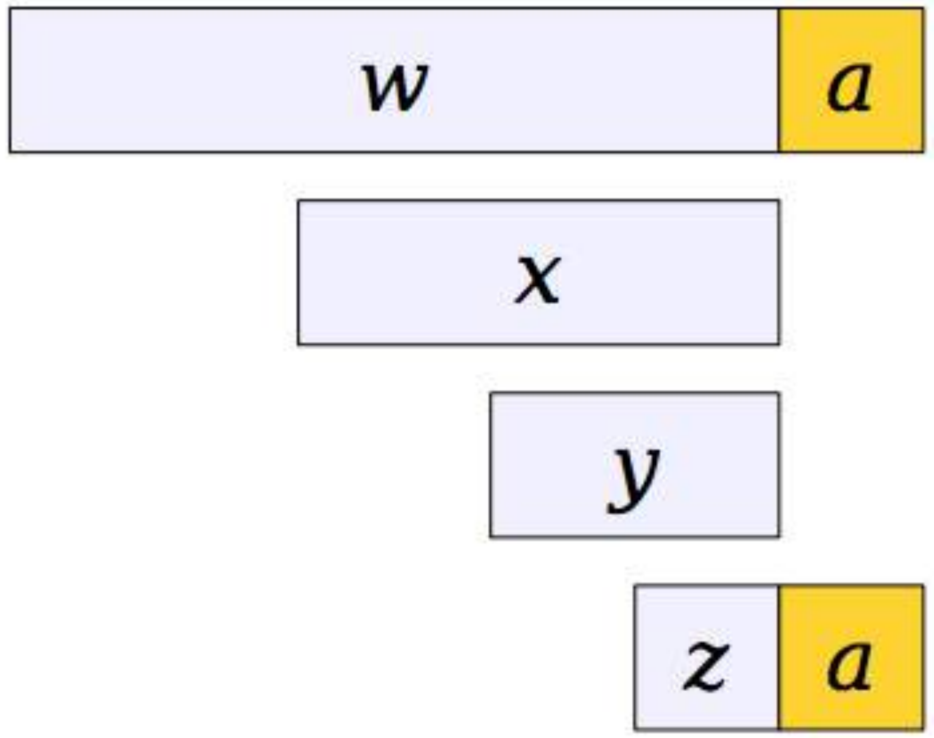


A better algorithm: intuition

Suppose we already know the failing edge from a node **w** to **x**. If we follow a solid edge with label **a**, there are two possibilities:

- ***Case 2:*** *xa* does not exist.





Constructing failing edge for a node

- To construct the failing edge for a node **wa**:
  - Follow **w**'s failing edge to node **x**.
  - If node **xa** exists, **wa** has a failing edge to **xa**.
  - Otherwise, follow **x**'s failing edge and repeat.
  - If you need to follow all the way back to the root, then **wa**’s failing edge points to the root.

- *Observation* 1: Failing edges point from longer strings to shorter strings.
- *Observation* 2: If we precompute failing edges for nodes in ascending order of string length, all of the information needed for the above approach will be available at the time we need it.

Complexity

- Focus on the time to fill in the failing edges for a single pattern of length **n**.
  - The failing edges moves one-step backward because it always points to a shorter string.
  - The solid edges moves one-step forward.
  - We cannot take more steps backward than forward. Therefore, across the entire construction, we can take at most **n** steps backward for this pattern.

- Total time required to construct failing edges for a pattern of length n: O(n).
- Total time required to construct failing edges for all k patterns: O(kn).

A different approach: suffix tree

- Build a tree from the text
- Used if the text is expected to be the same during several pattern queries
- Tree building is O(m) where m is the size of the text. This is preprocessing.
- Given any pattern of length n, we can answer if it occurs in text in O(n) time
- Suffix tree = “modified” keyword tree of all suffixes of text

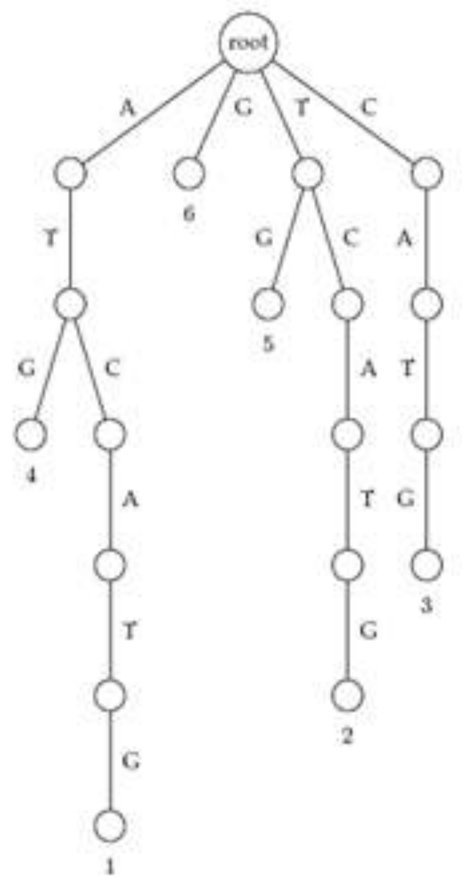
Construct a suffix tree

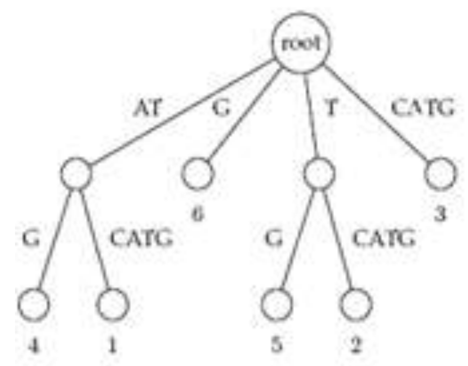
`Text: ATCATG`

suffixes

ATCATG
TCATG
CATG
ATG
TG
G

→ Keyword Tree → Suffix Tree



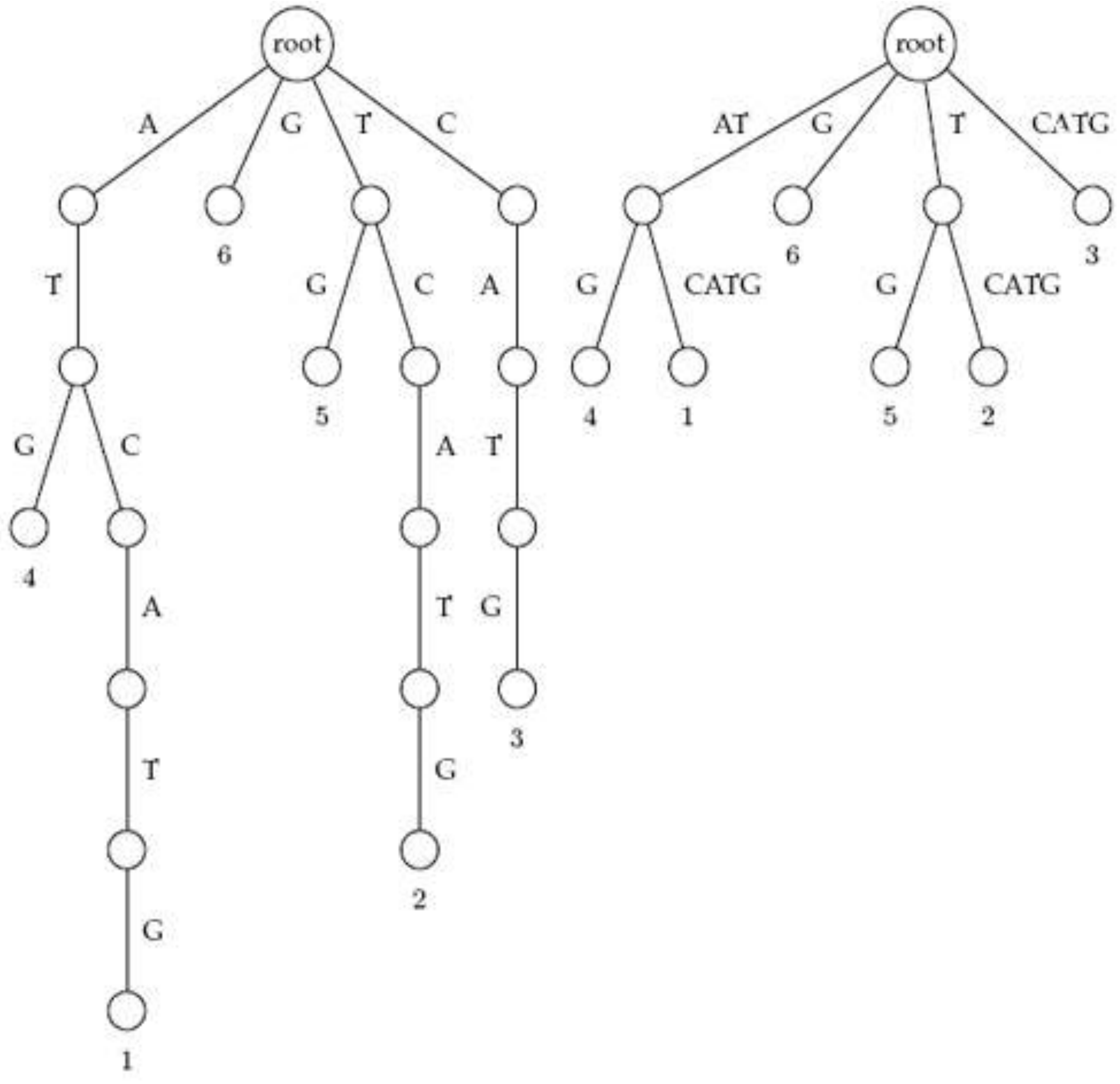


Suffix tree = Collapsed Keyword Tree on Suffixes

Similar to keyword trees, except edges that form paths are collapsed

- Each edge is labeled with a *substring* of a text for less space
- All internal edges have at least two outgoing edges
- Leaves labeled by the location of the suffix on the text.

Text: ATCATG



(a) Keyword tree (b) Suffix tree

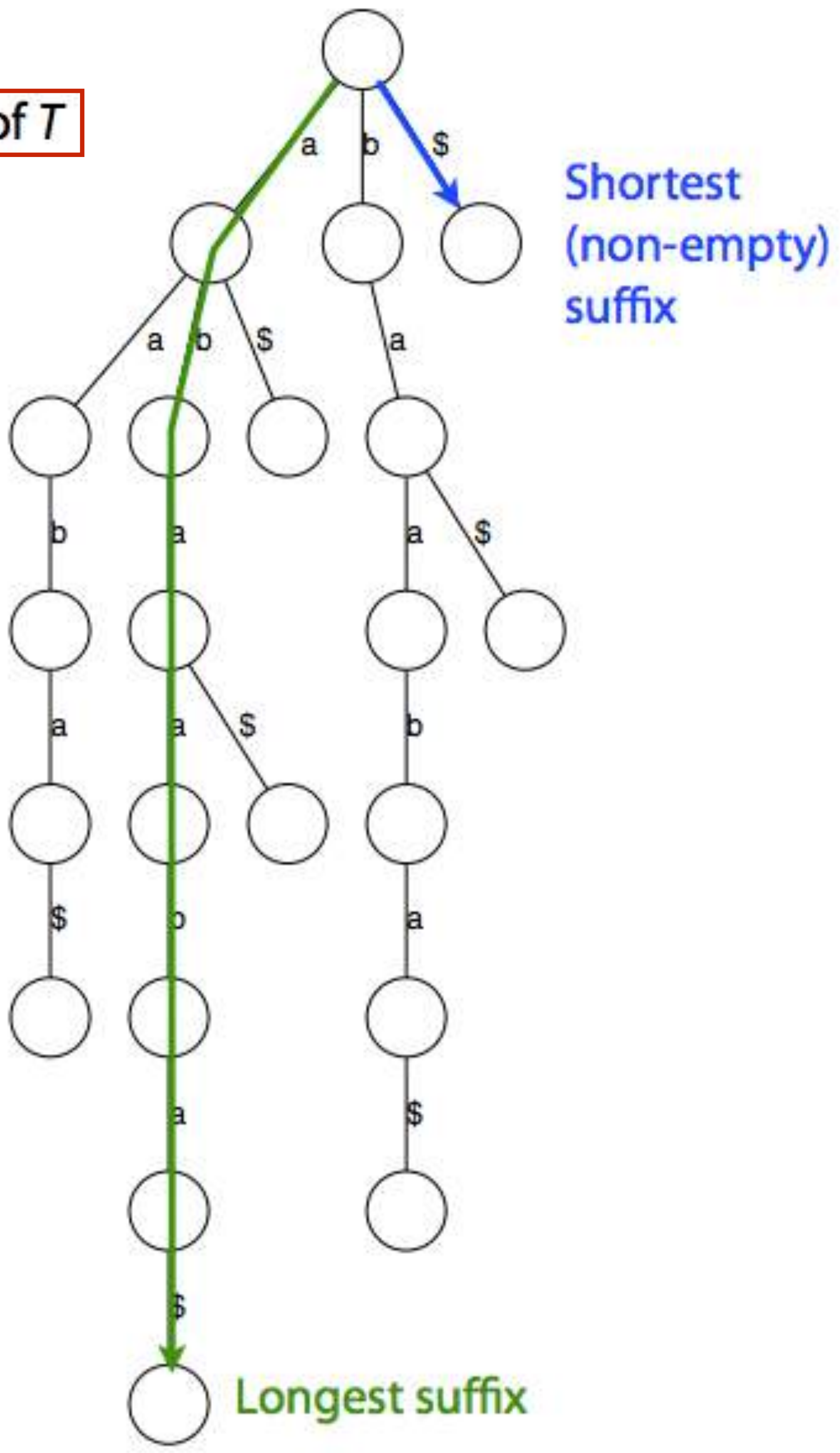
Example: suffix keyword tree

add special *terminal character* **$** to the end of *T*

T: abaaba   *T***$**: abaaba**$**

Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf

Would this still be the case if we hadn't added **$**?

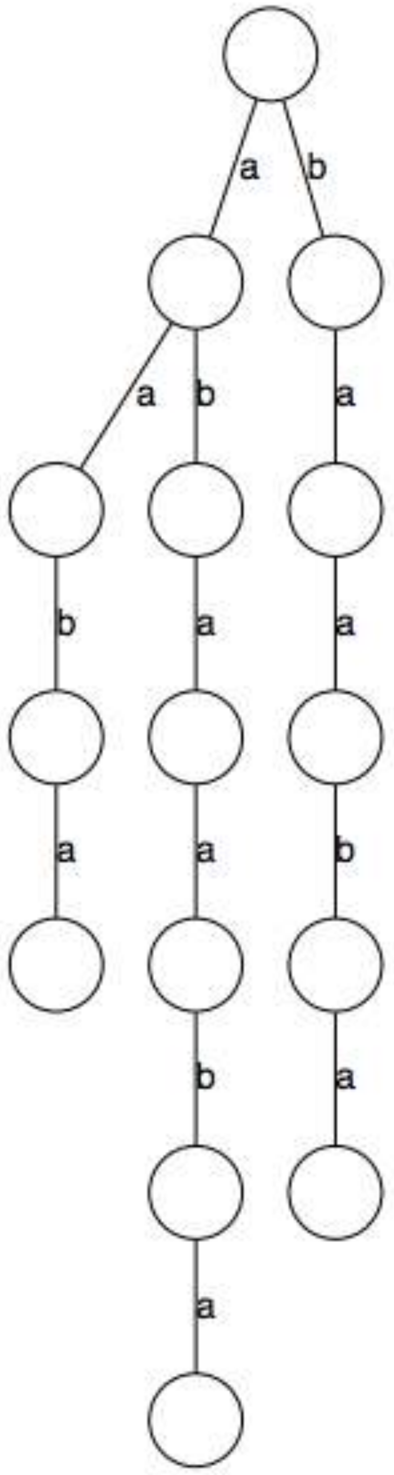


Example: suffix keyword tree

T: abaaba

Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf

Would this still be the case if we hadn't added **$**? **No**



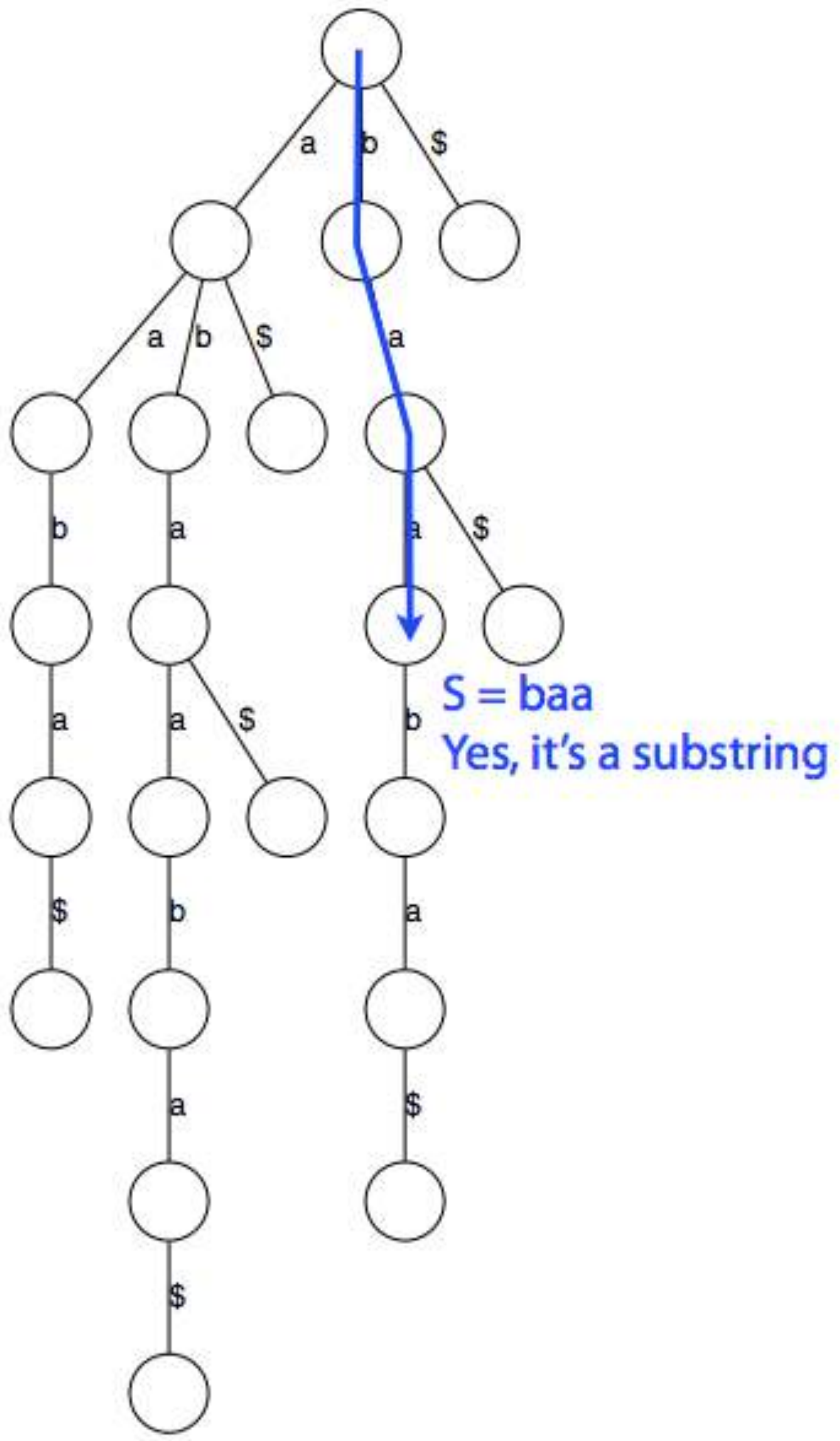
Example: suffix keyword tree

How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.

Start at the root and follow the edges labeled with the characters of *S*

- If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*
- If we exhaust *S* without falling off, *S* is a substring of *T*



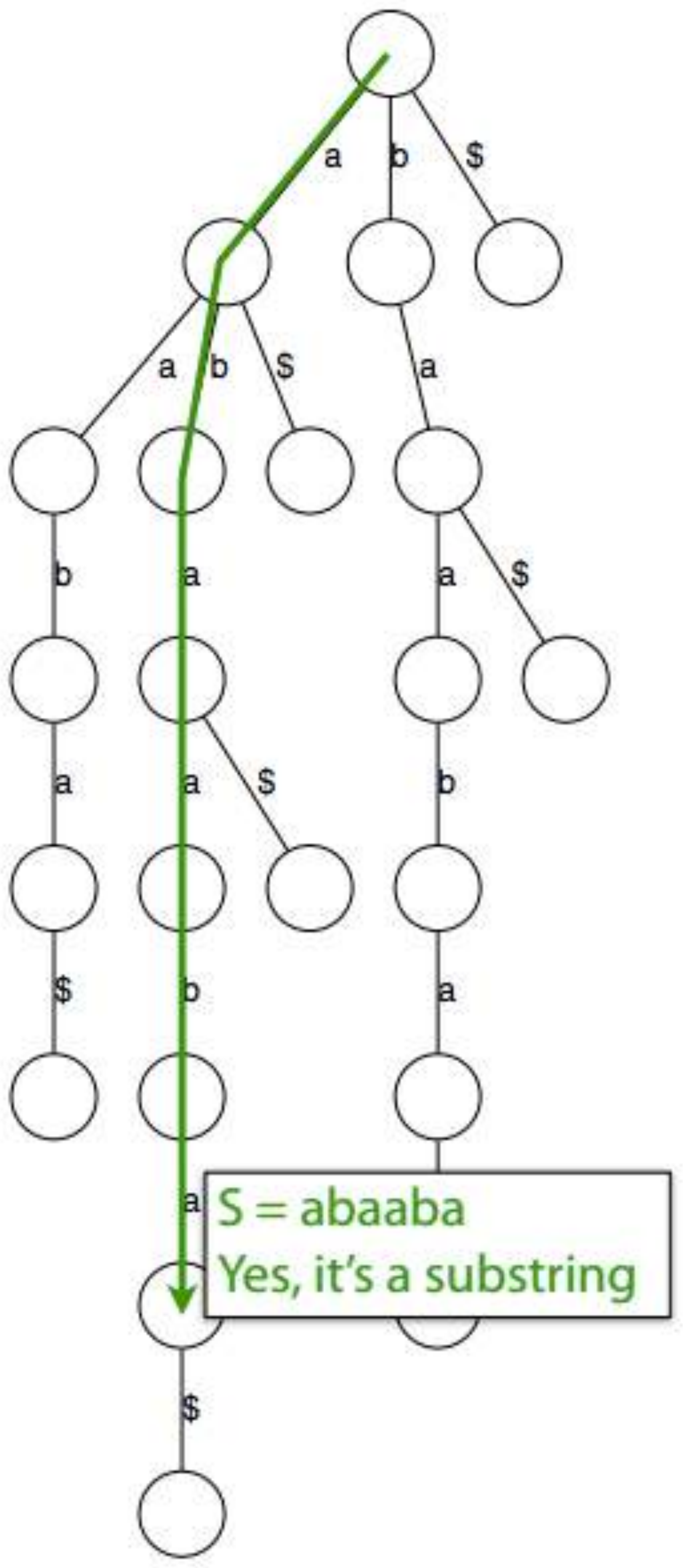
Example: suffix keyword tree

How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.

Start at the root and follow the edges labeled with the characters of *S*

- If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*
- If we exhaust *S* without falling off, *S* is a substring of *T*



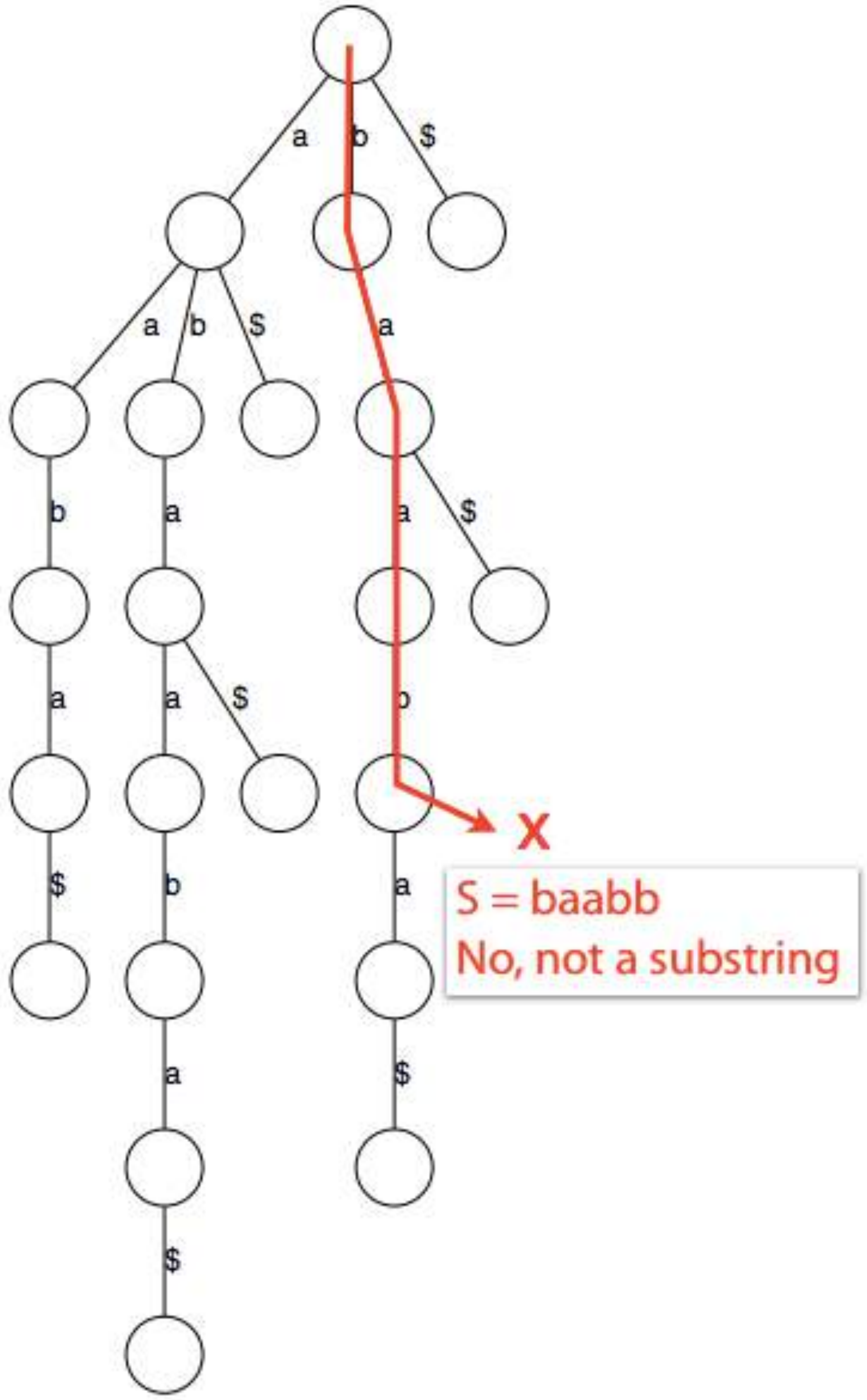
Example: suffix keyword tree

How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.

Start at the root and follow the edges labeled with the characters of *S*

- If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*
- If we exhaust *S* without falling off, *S* is a substring of *T*



Summary

- Keyword and suffix trees are used to find patterns in a text

- Keyword trees:
  - Build keyword tree of patterns, and thread text through it
  - Usage: checking a set of patterns within various texts

- Suffix trees:
  - Build suffix tree of text, and thread patterns through it
  - Usage: checking various patterns in the same text