Suffix Trees

02-713
Slides by Carl Kingsford

Preprocessing Strings

- Typical setting: A long, known, and fixed text string (like a genome) and many unknown, changing query strings.

  - Allowed to preprocess the text string once in anticipation of the future unknown queries.

- Preprocessing string data into data structures that make many questions (like searching) easy to answer.

Suffix Tries

- A <u>trie</u>, pronounced "try", is a tree that exploits some structure in the keys

  - e.g. if the keys are strings, a binary search tree would compare the entire strings, but a trie would look at their individual characters

- Suffix trie are a space-inefficient data structure to store a string that allows many kinds of queries to be answered quickly.

- Suffix trees are hugely important for searching large sequences like genomes. Eg. the basis for a tool called "MUMMer".

Suffix Tries

s = abaaba$
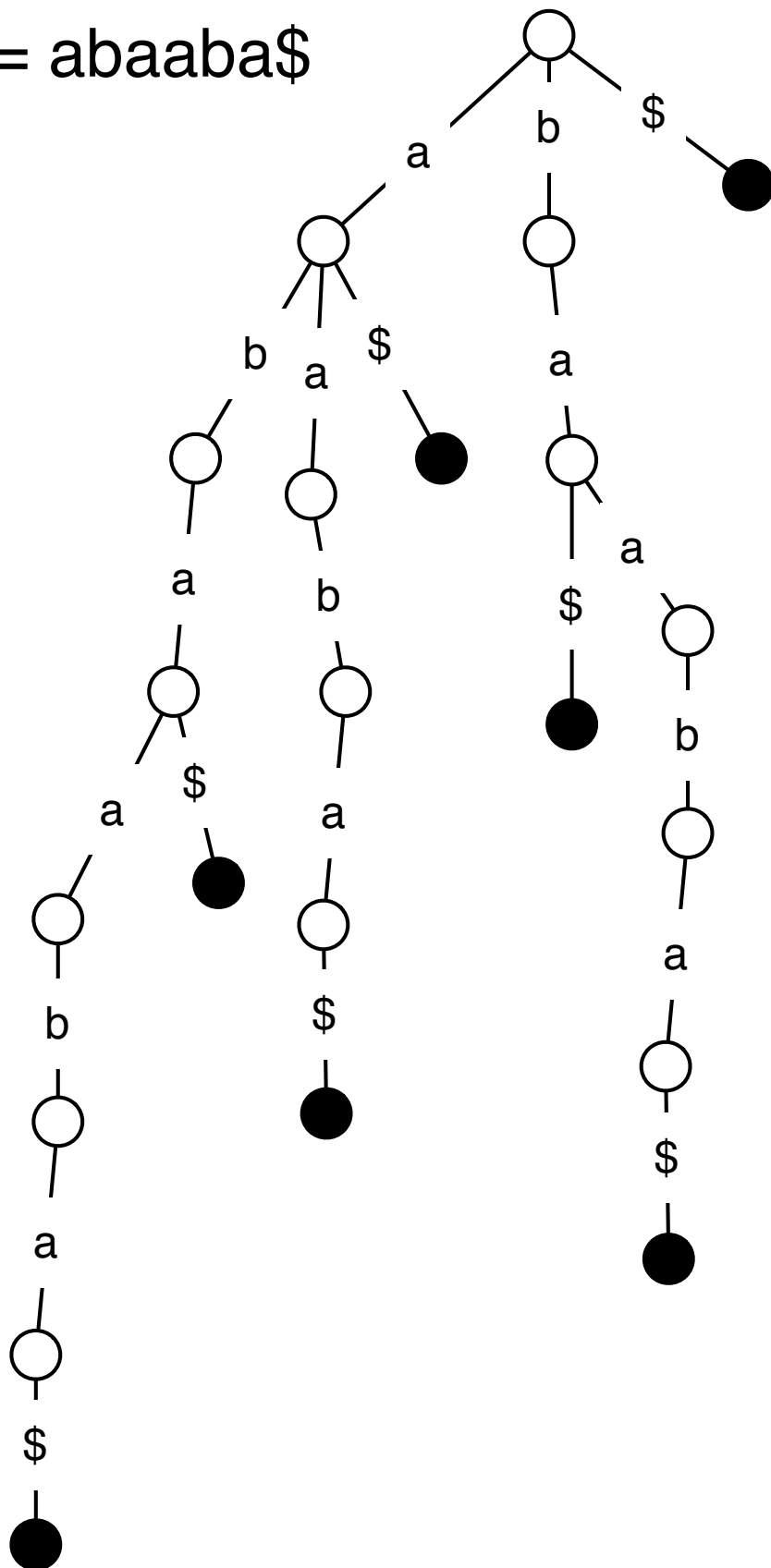
SufTrie(s) = suffix trie representing string s.

Edges of the suffix trie are labeled with letters from the alphabet ∑ (say {A,C,G,T}).

Every path from the root to a solid node represents a suffix of s.

Every suffix of s is represented by some path from the root to a solid node.

Why are all the solid nodes leaves?
How many leaves will there be?
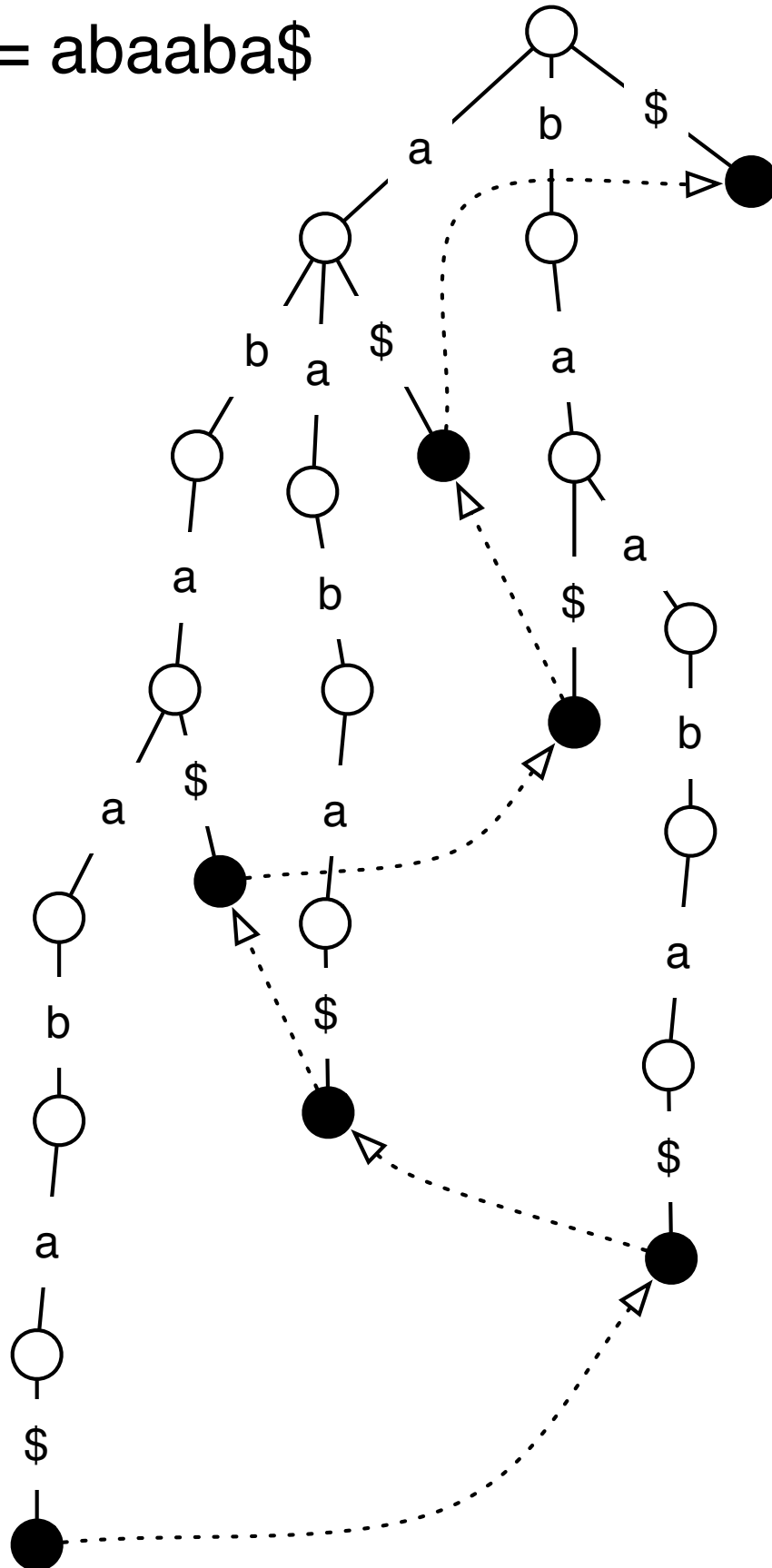
Processing Strings Using Suffix Tries

Given a suffix trie T, and a string q, how can we:

- determine whether q is a substring of T?

- check whether q is a suffix of T?

- count how many times q appears in T?

- find the longest repeat in T?

- find the longest common substring of T and q?

Main idea:
    every substring of s is a prefix of some suffix of s.

Searching Suffix Tries

s = abaaba$

Is "baa" a substring of s?

Follow the path given by the query string.

After we've built the suffix trees, queries can be answered in time:
  O(|query|)
regardless of the text size.

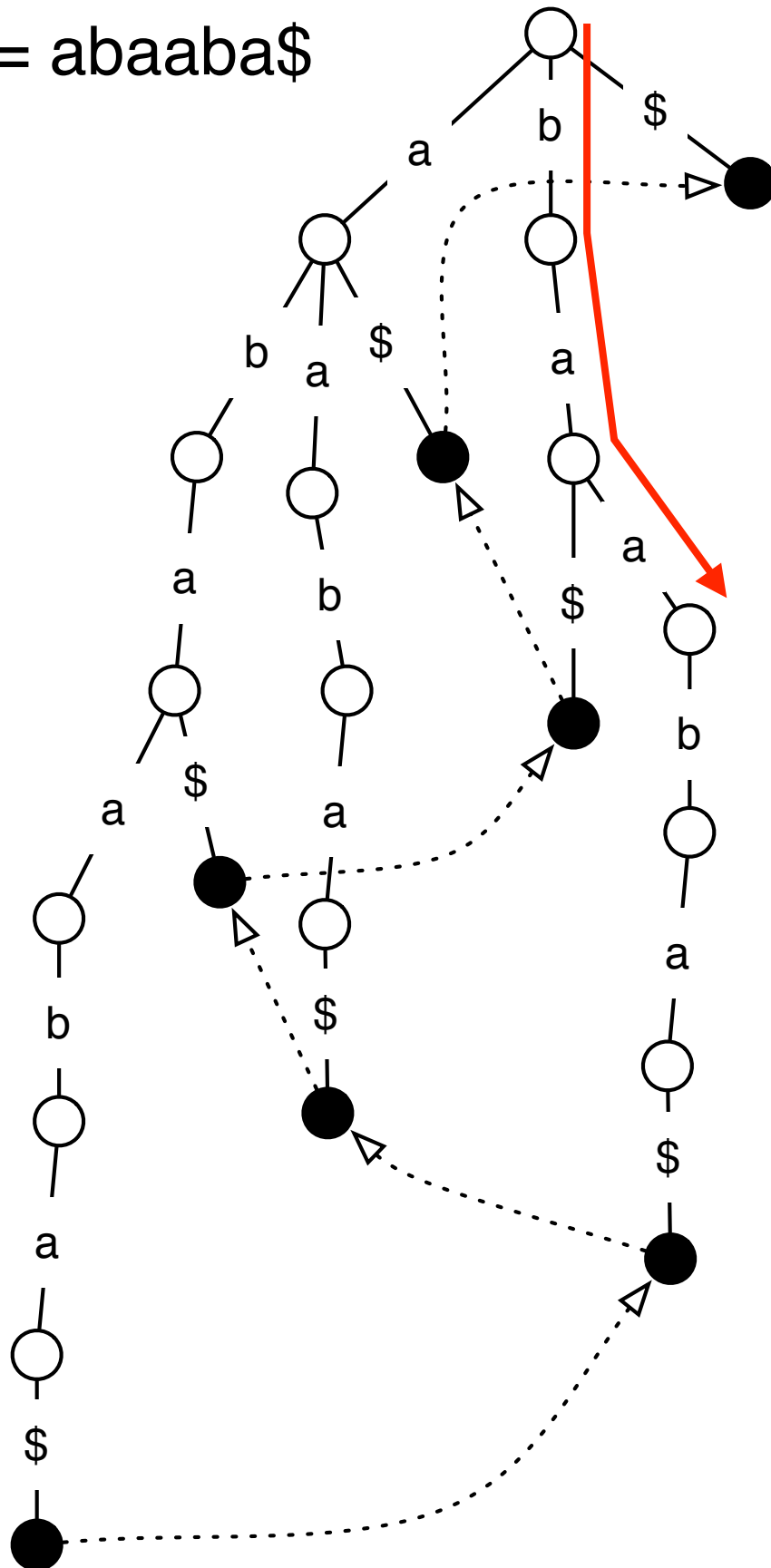# Searching Suffix Tries

s = abaaba$

Is "baa" a substring of s?

Follow the path given by the query string.

After we've built the suffix trees, queries can be answered in time:
  O(|query|)
regardless of the text size.

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Check whether q is a **suffix** of T:

Count # of occurrences of q in T:

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Count # of occurrences of q in T:

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Follow the path for q starting from the root.
The number of leaves under the node you end up in is the
number of occurrences of q.

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Follow the path for q starting from the root.
The number of leaves under the node you end up in is the
number of occurrences of q.

Find the longest repeat in T:

Find the deepest node that has at least 2 leaves under it.

Find the lexicographically (alphabetically) first suffix:

Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Follow the path for q starting from the root.
The number of leaves under the node you end up in is the
number of occurrences of q.

Find the longest repeat in T:

Find the deepest node that has at least 2 leaves under it.

Find the lexicographically (alphabetically) first suffix:

Start at the root, and follow the edge labeled with the
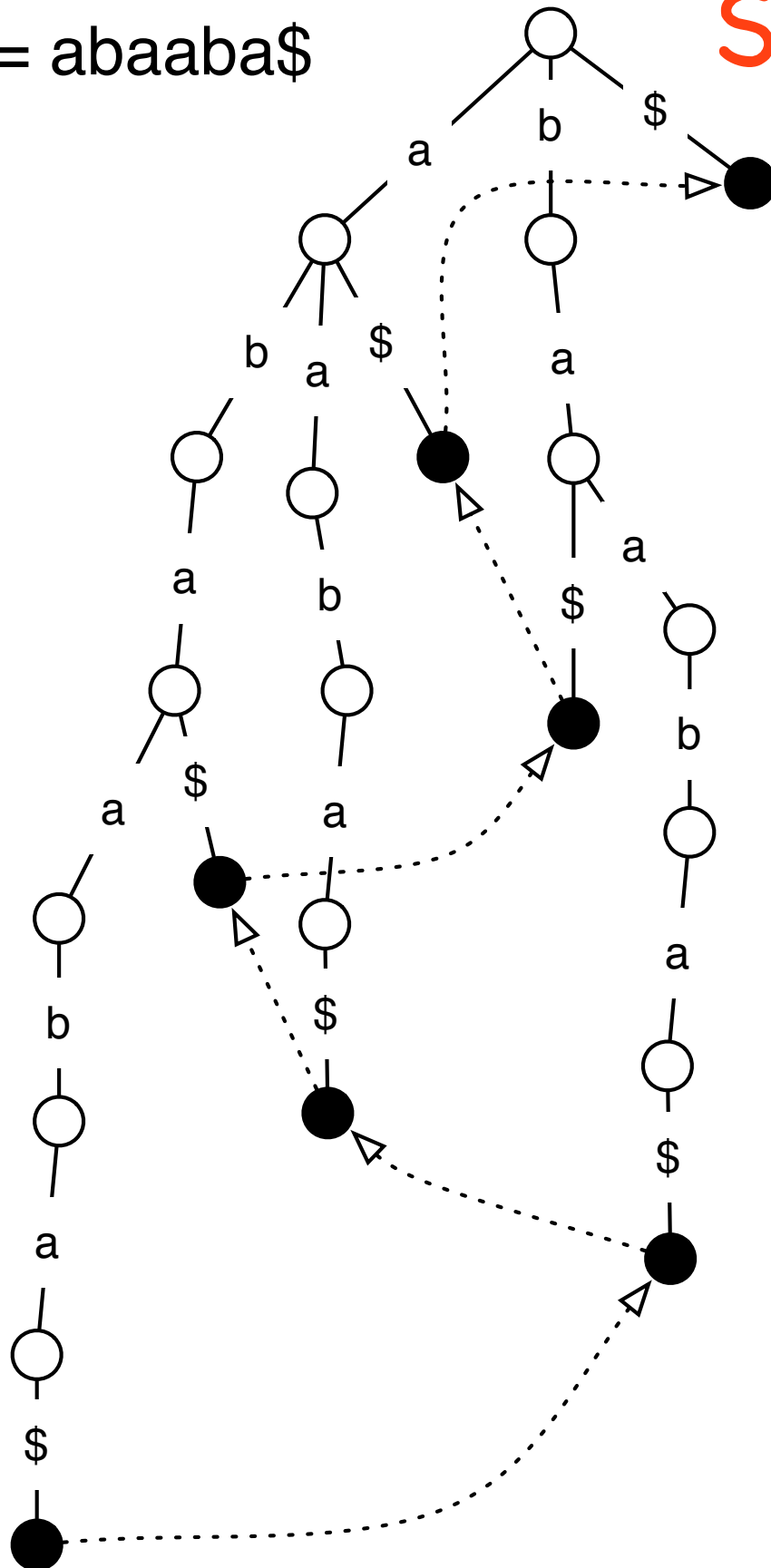lexicographically (alphabetically) smallest letter.

Suffix Links

s = abaaba$
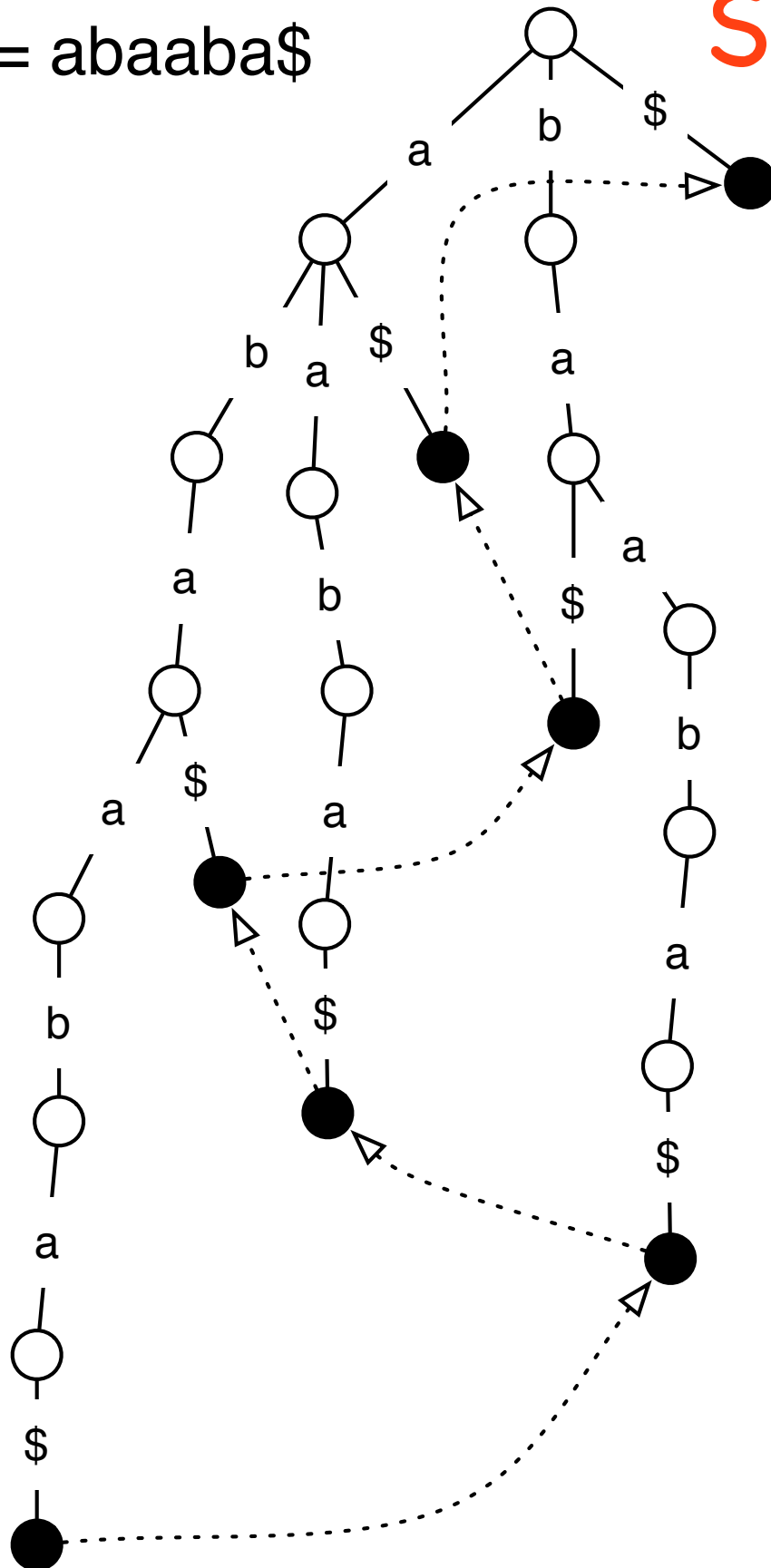
- Suffix links connect node representing "xα" to a node representing "α".

- Most important suffix links are the ones connecting suffixes of the full string (shown at right).

- But every node has a suffix link.
  - Why?
  - How do we know a node representing α exists for every node representing xα?

Suffix Tries

s = abaaba$

A node represents the prefix of some suffix:

$$ab\boxed{aab}a\$$$

The node's suffix link should link to the <u>prefix</u> of the suffix s that is 1 character shorter.

Since the suffix trie contains all suffixes, it contains a path representing s, and therefore contains a node representing every prefix of s.

s = abaaba$

A node represents the prefix of some suffix:

$$ab\boxed{\underline{aab}a\$}$$

$$\underbrace{}_{s}$$

The node's suffix link should link to the prefix of the suffix s that is 1 character shorter.

Since the suffix trie contains all suffixes, it contains a path representing s, and therefore contains a node representing every prefix of s.

9

Applications of Suffix Tries (II)

Find the longest common substring of T and q:

abaaba$

T = abaaba$
q = bbaa
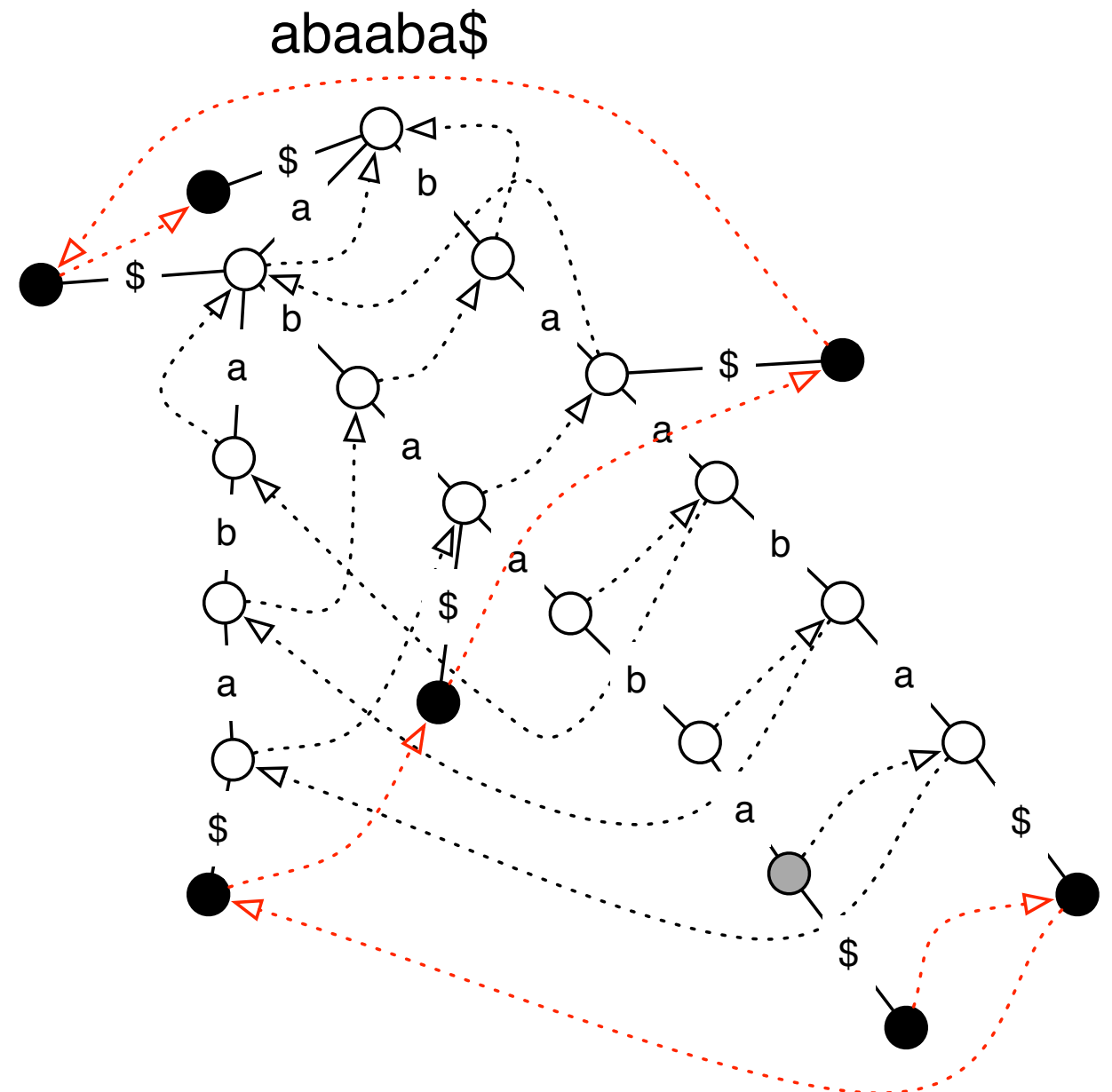
Applications of Suffix Tries (II)

Find the longest common substring of T and q:

Walk down the tree following q.
If you hit a dead end, save the current depth, and follow the suffix link from the current node.
When you exhaust q, return the longest substring found.

T = abaaba$
q = bbaa
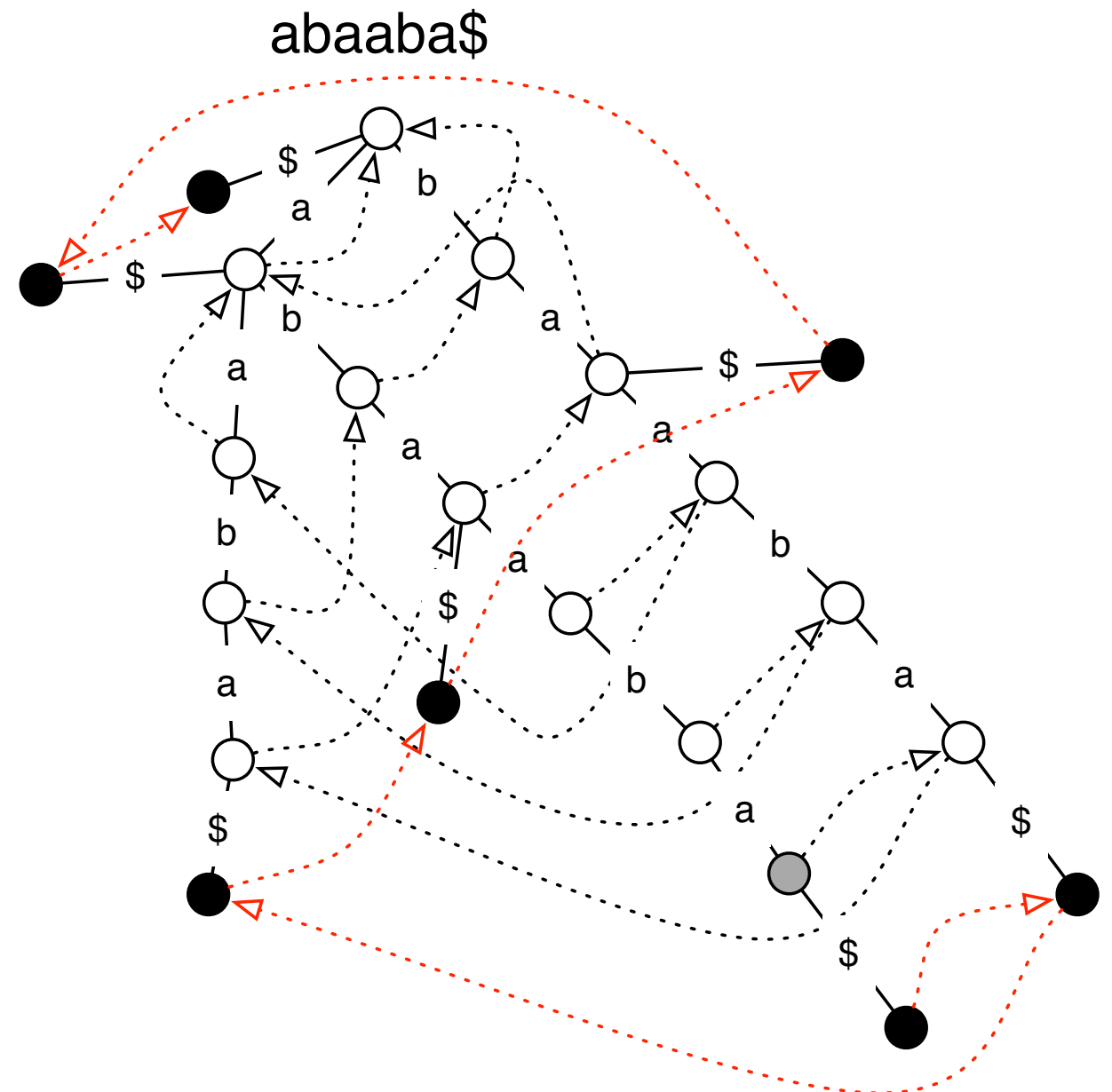
abaaba$

Constructing Suffix Tries

Suppose we want to build suffix trie for string:

s = abbacabaa

We will walk down the string from left to right:

abbacabaa
→

building suffix tries for s[0], s[0..1], s[0..2], ..., s[0..n]

To build suffix trie for s[0..i], we
will use the suffix trie for s[0..i-1]
built in previous step

To convert SufTrie(S[0..i-1]) → SufTrie(s[0..i]), add character s[i] to all the suffixes:
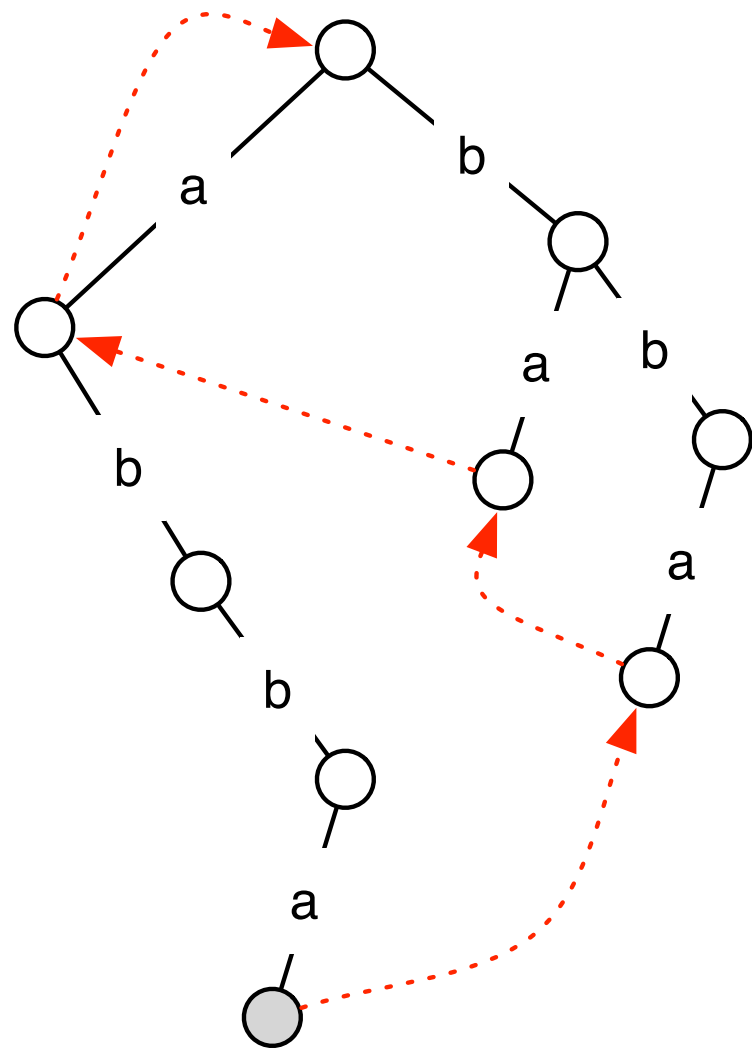
abbacabaa
i=4

Need to add nodes for
the suffixes:

abbac
bbac
bac
ac
c

Purple are suffixes that
will exist in
SufTrie(s[0..i-1]) Why?

How can we find these
suffixes quickly?

Suppose we want to build suffix trie for string:

s = abbacabaa

We will walk down the string from left to right:

abba**c**abaa
→

building suffix tries for s[0], s[0..1], s[0..2], ..., s[0..n]

To build suffix trie for s[0..i], we
will use the suffix trie for s[0..i-1]
built in previous step

To convert SufTrie(S[0..i-1]) → SufTrie(s[0..i]), add character s[i] to all the suffixes:

abba**c**abaa
i=4

Need to add nodes for
the suffixes:

abba**c**
bba**c**
ba**c**
a**c**
c

Purple are suffixes that
will exist in
SufTrie(s[0..i-1]) Why?

How can we find these
suffixes quickly?

abba**c**abaa

i=4

Need to add nodes for the suffixes:

abba**c**
bba**c**
ba**c**
a**c**
c

Purple are suffixes that will exist in SufTrie(s[0..i-1]) Why?

How can we find these suffixes quickly?

SufTrie(abba)

SufTrie(abbac)
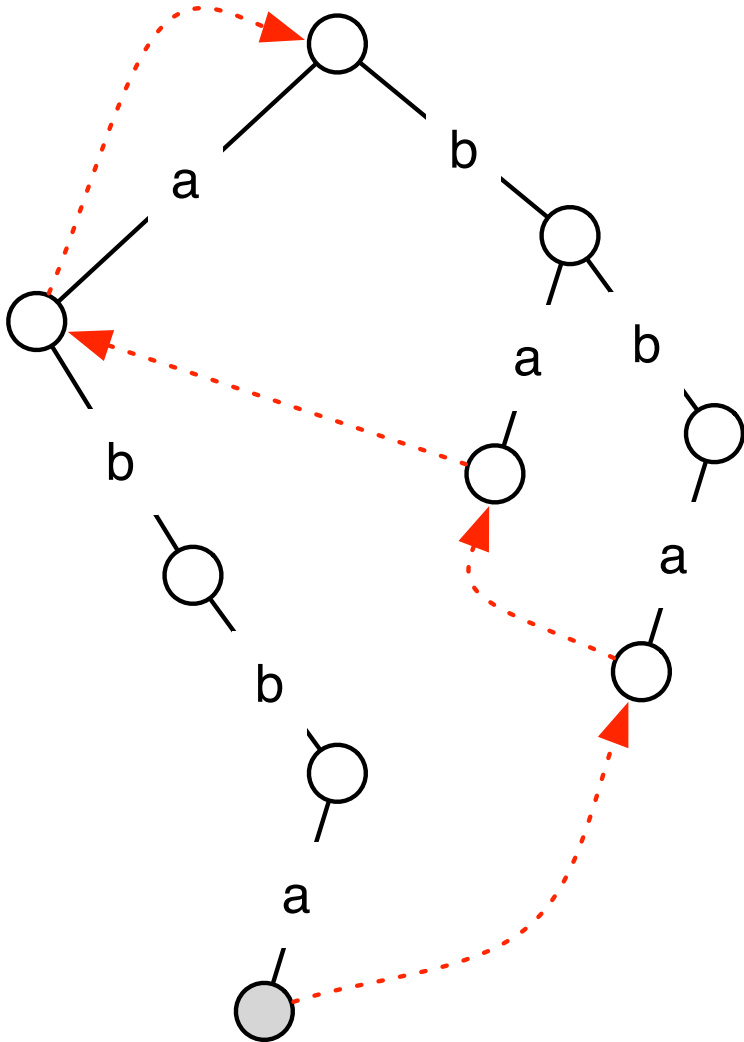
Where is the new deepest node? (aka longest suffix)

How do we add the suffix links for the new nodes?

13

abbacabaa
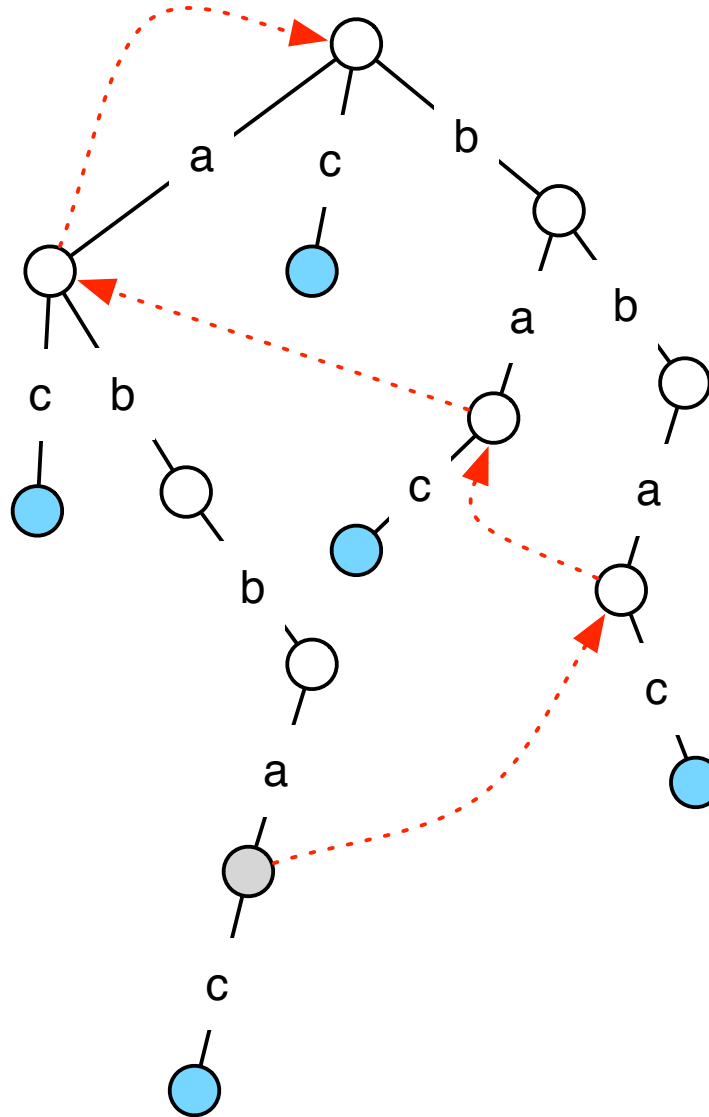i=4

Need to add nodes for
the suffixes:

abbac
bbac
bac
ac
c

Purple are suffixes that
will exist in
SufTrie(s[0..i-1]) Why?

How can we find these
suffixes quickly?

SufTrie(abba)

SufTrie(abbac)

Where is the new
deepest node? (aka
longest suffix)

How do we add the
suffix links for the
new nodes?

To build SufTrie(s[0..i]) from SufTrie(s[0..i-1]):

CurrentSuffix = longest (aka deepest suffix)

Repeat:
   Add child labeled s[i] to CurrentSuffix.
   Follow suffix link to set CurrentSuffix to next shortest suffix.

Add suffix links connecting nodes you just added in the order in which you added them.

until you reach the root or the current node already has an edge labeled s[i] leaving it.

Because if you already have a node for suffix αs[i] then you have a node for every smaller suffix.

In practice, you add these links as you go along, rather than at the end.

Python Code to Build a Suffix Trie

```python
class SuffixNode:
    def __init__(self, suffix_link = None):
        self.children = {}
        if suffix_link is not None:
            self.suffix_link = suffix_link
        else:
            self.suffix_link = self

    def add_link(self, c, v):
        """link this node to node v via string c"""
        self.children[c] = v
```

```python
def build_suffix_trie(s):
    """Construct a suffix trie."""
    assert len(s) > 0

    # explicitly build the two-node suffix tree
    Root = SuffixNode()         # the root node
    Longest = SuffixNode(suffix_link = Root)
    Root.add_link(s[0], Longest)

    # for every character left in the string
    for c in s[1:]:
        Current = Longest; Previous = None
        while c not in Current.children:

            # create new node r1 with transition Current -c->r1
            r1 = SuffixNode()
            Current.add_link(c, r1)

            # if we came from some previous node, make that
            # node's suffix link point here
            if Previous is not None:
                Previous.suffix_link = r1

            # walk down the suffix links
            Previous = r1
            Current = Current.suffix_link

        # make the last suffix link
        if Current is Root:
            Previous.suffix_link = Root
        else:
            Previous.suffix_link = Current.children[c]

        # move to the newly added child of the longest path
        # (which is the new longest path)
        Longest = Longest.children[c]
    return Root
```
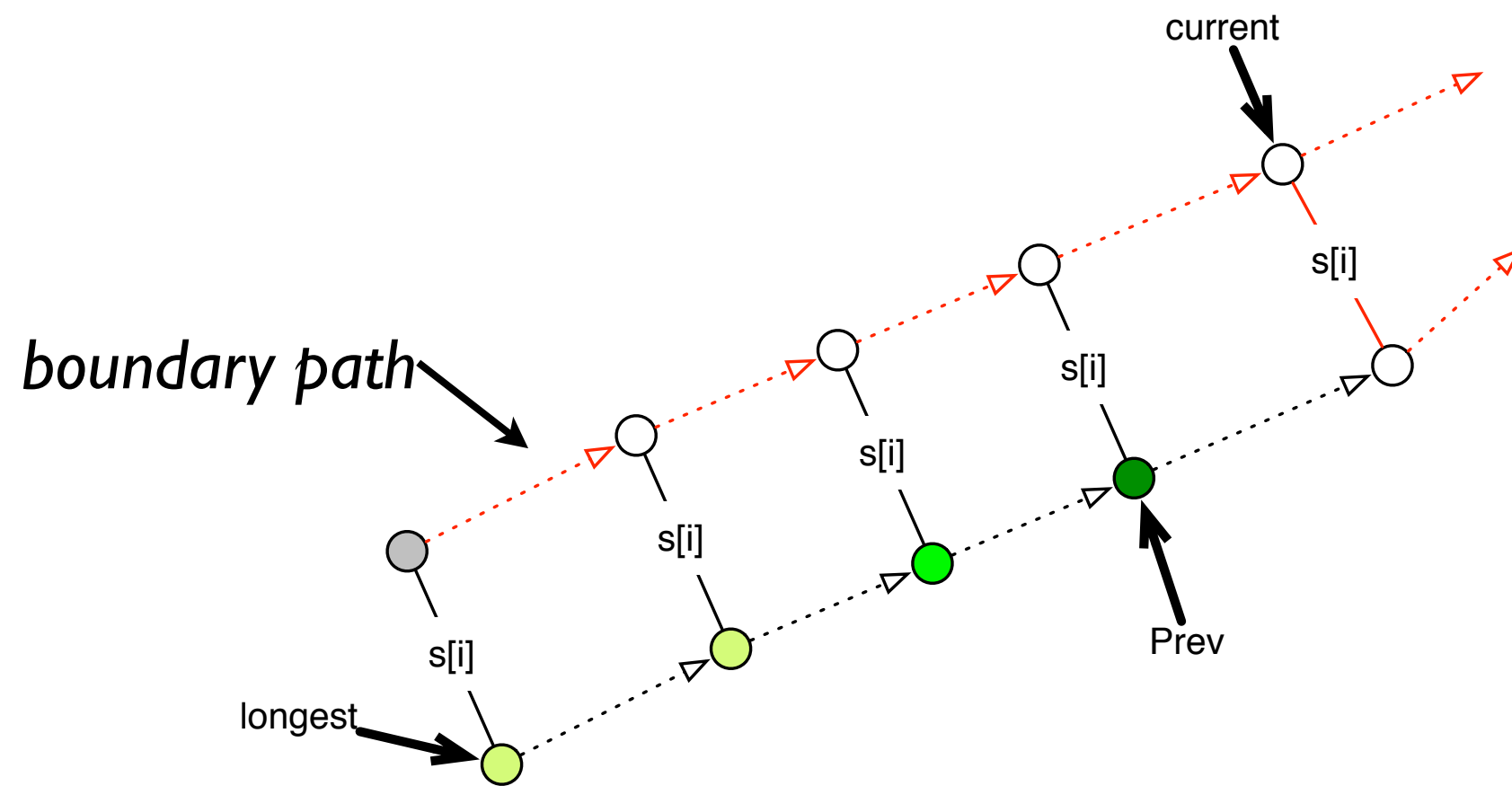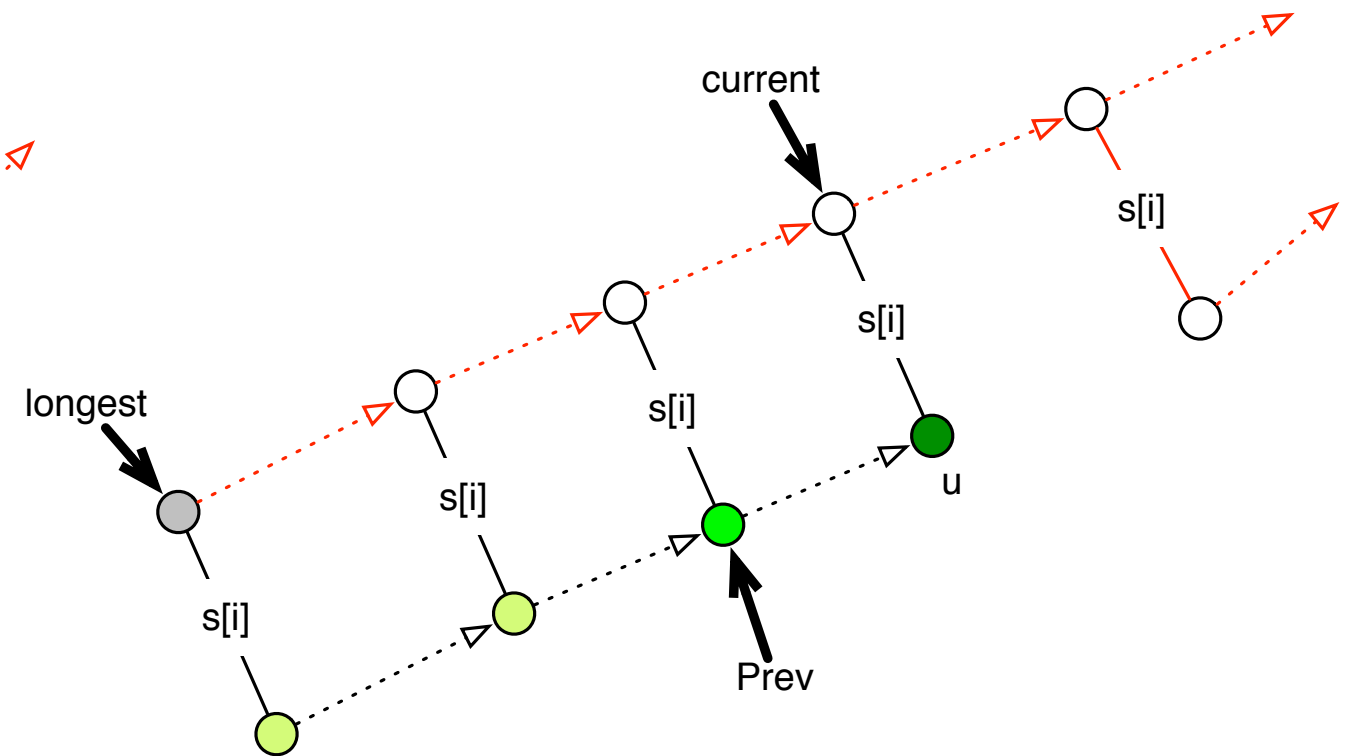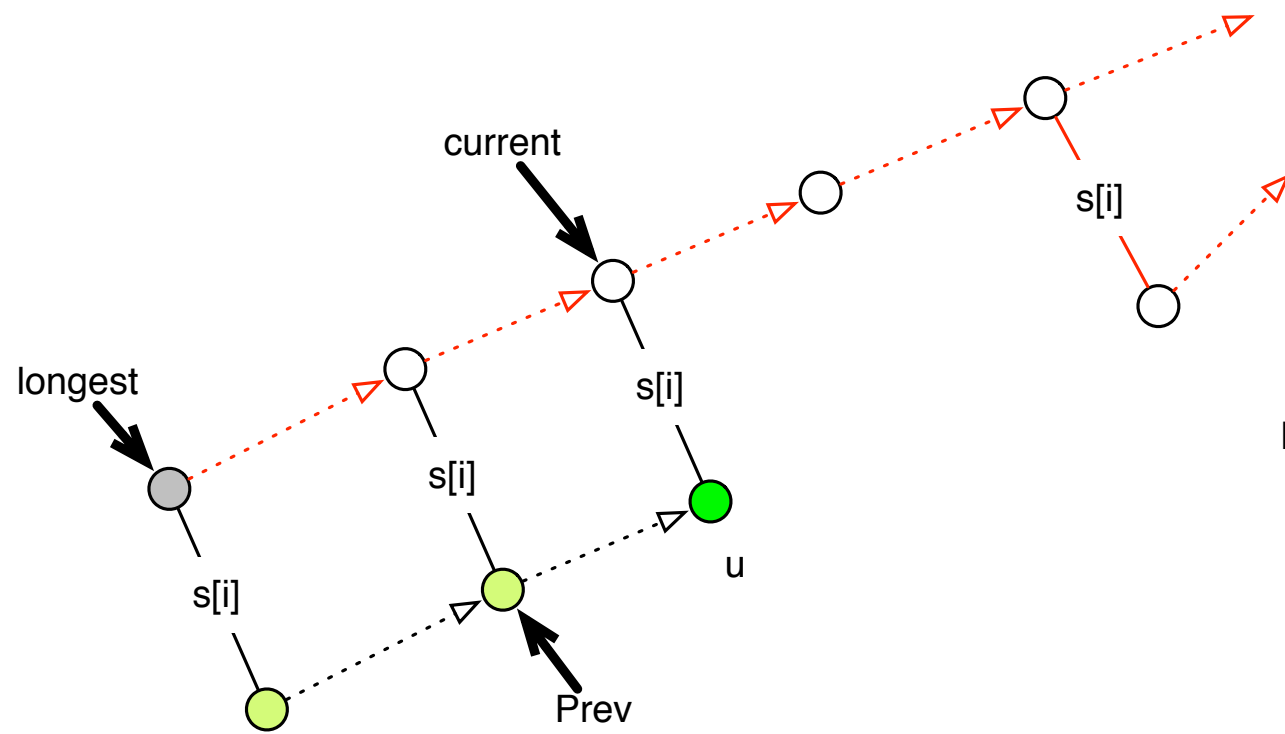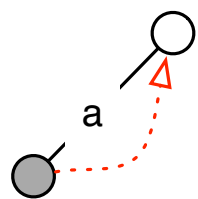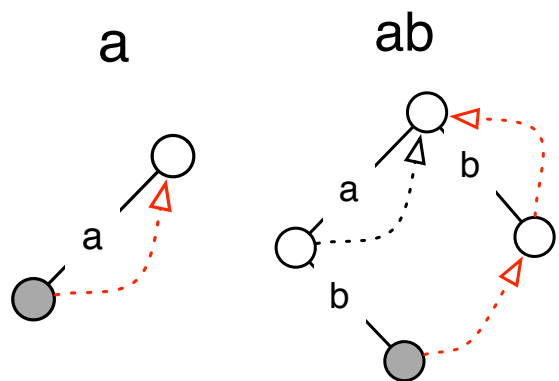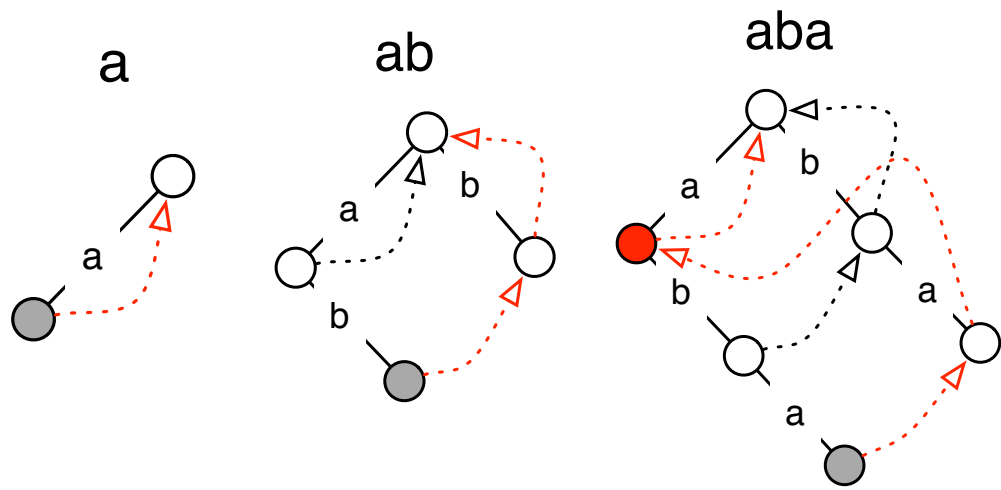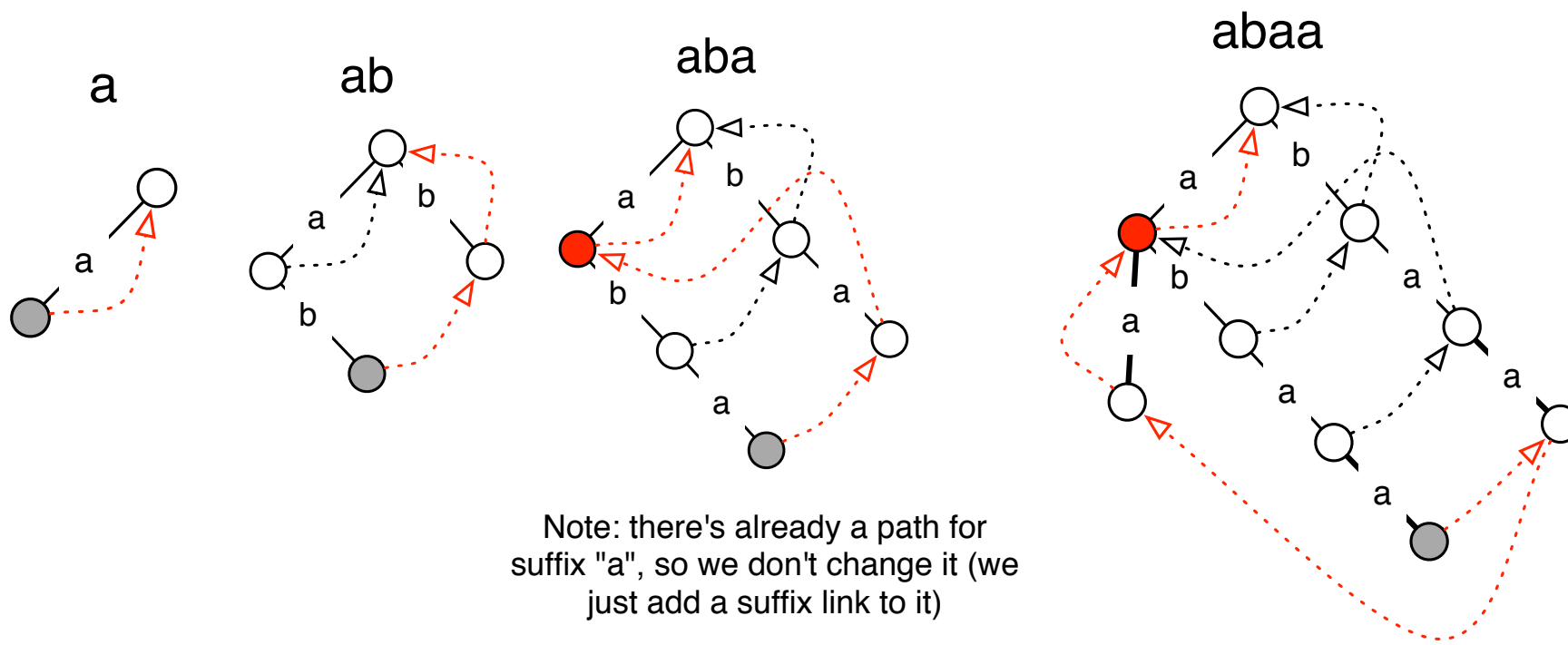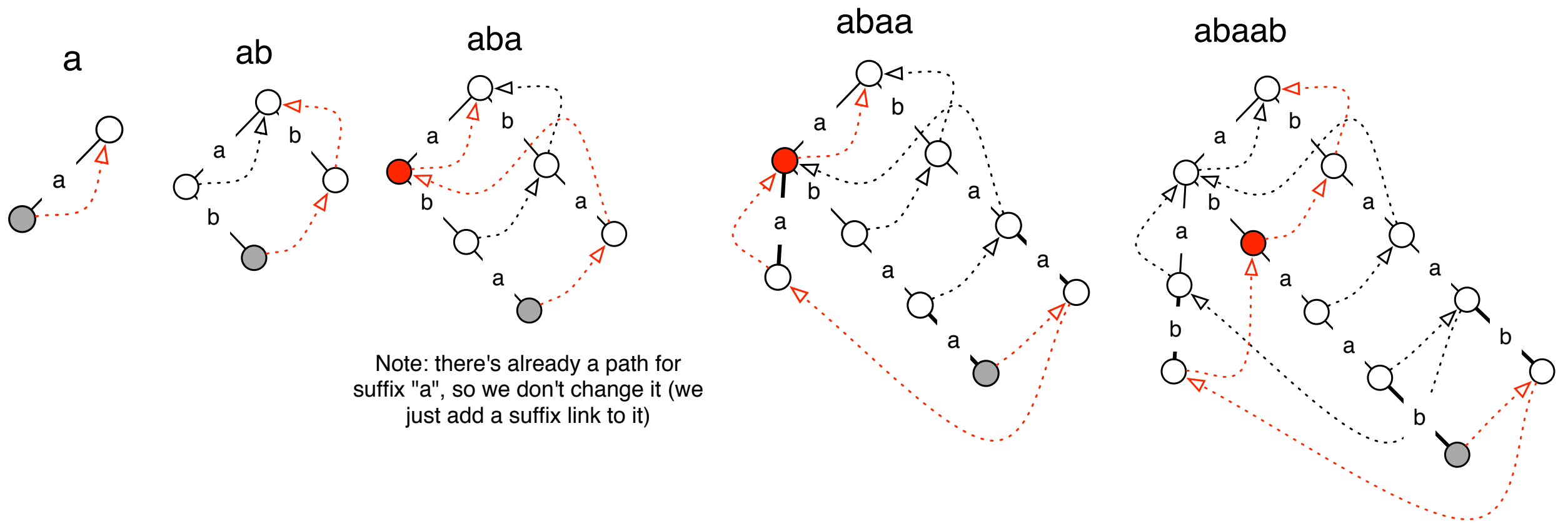
s[0]

current

longest

s[i]

s[i]

s[i]

s[i]

u

Prev

current

longest

s[i]

s[i]

s[i]

s[i]

u

Prev

boundary path

current

s[i]

s[i]

s[i]

s[i]

s[i]

Prev
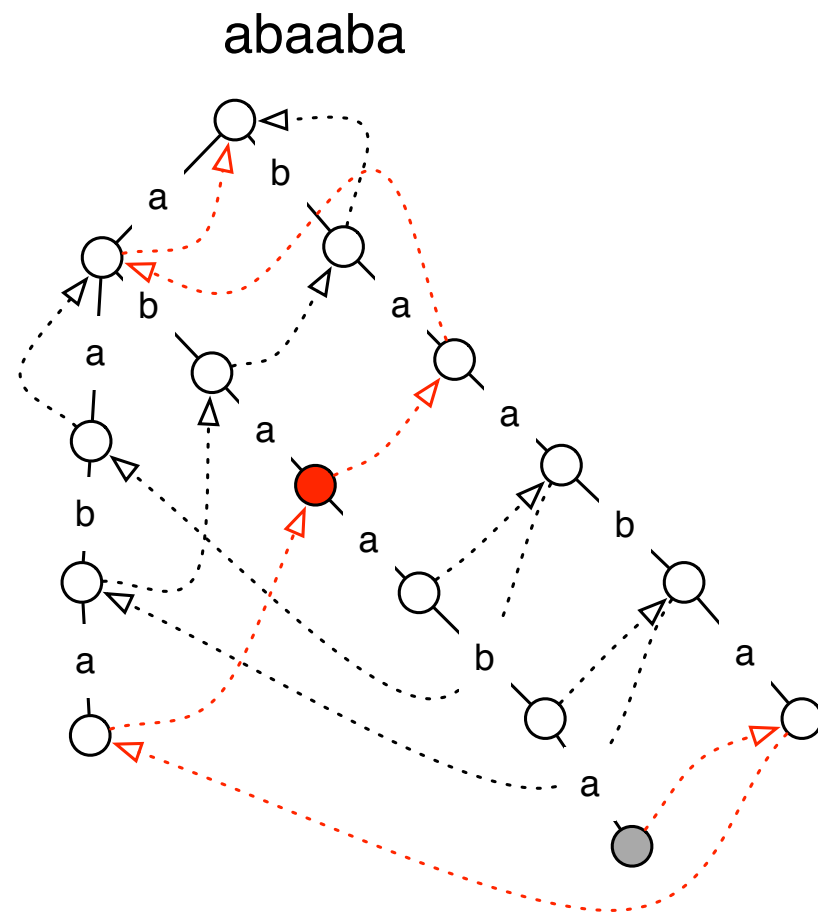
longest

a

a   ab

a

ab

aba

Note: there's already a path for
suffix "a", so we don't change it (we
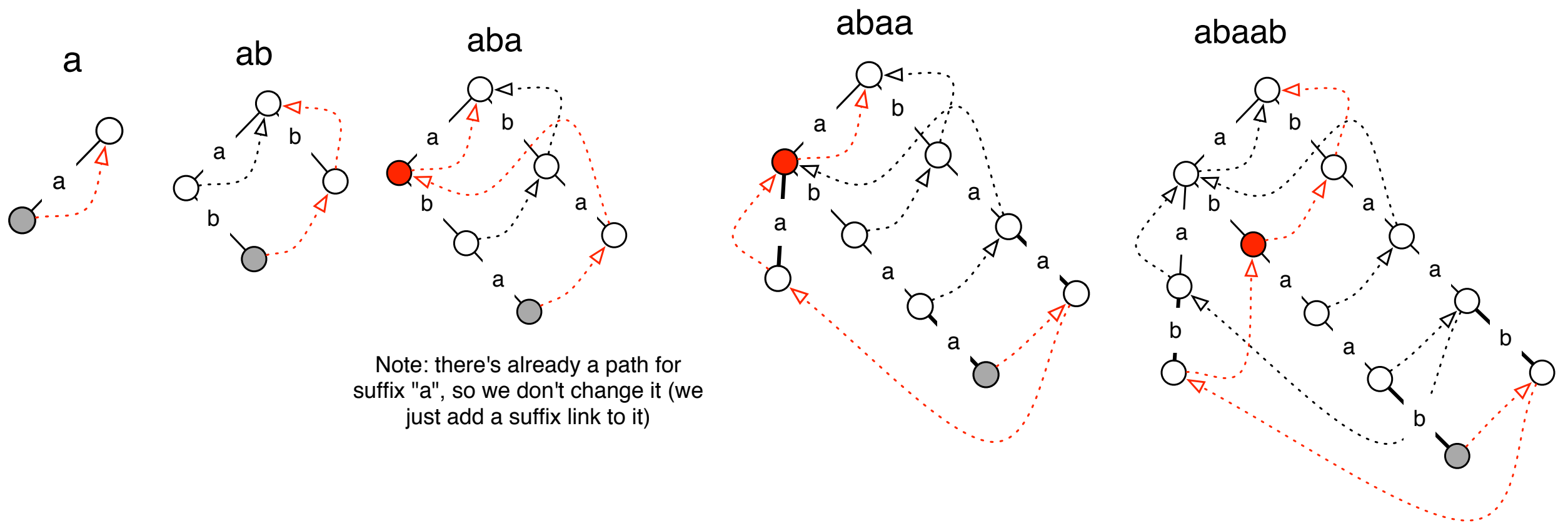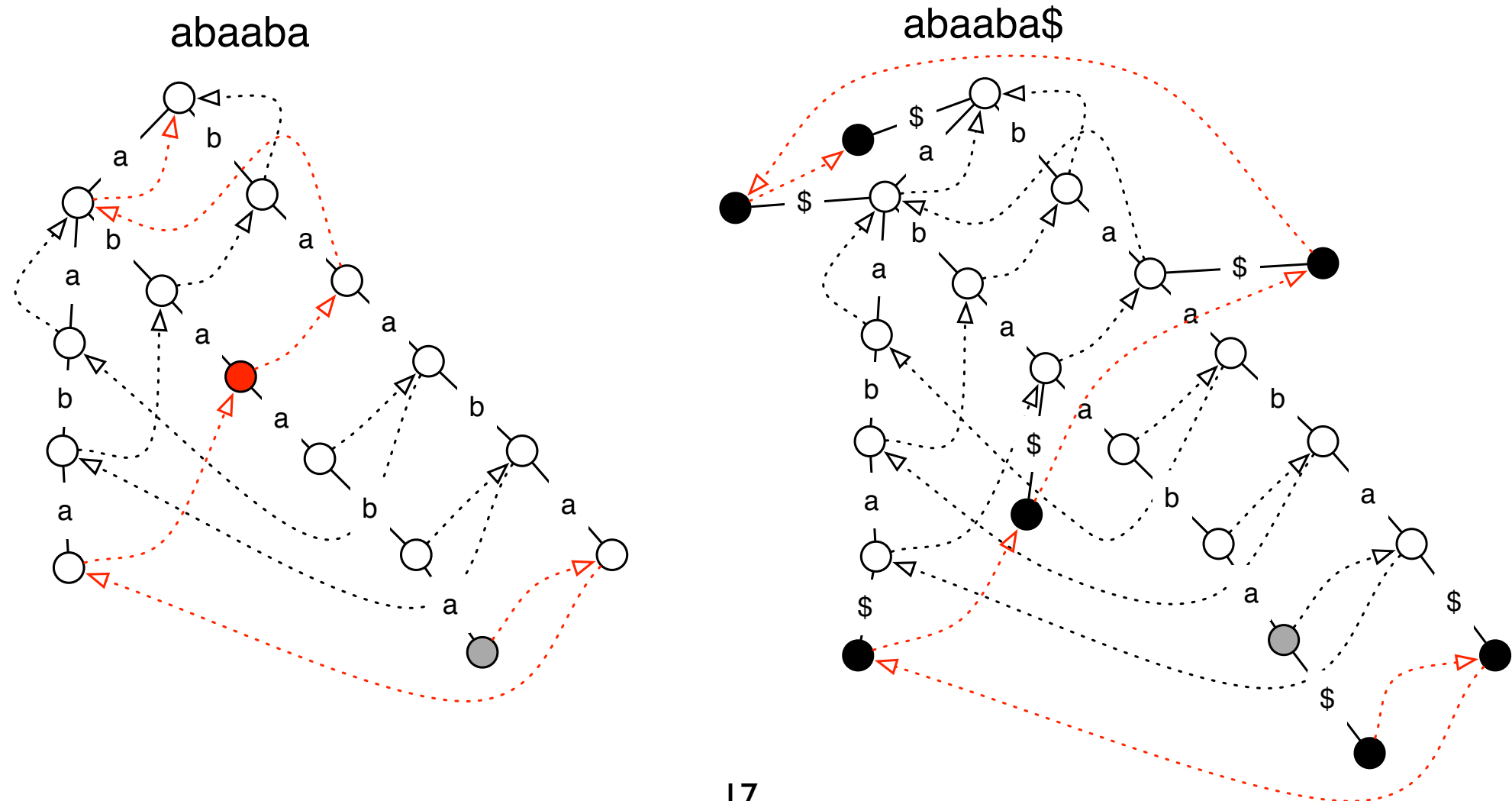just add a suffix link to it)

a

ab

aba

abaa

Note: there's already a path for
suffix "a", so we don't change it (we
just add a suffix link to it)

a   ab   aba   abaa   abaab

Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)

17

a

ab

aba

Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)

abaa

abaab

abaaba

a   ab   aba

Note: there's already a path for
suffix "a", so we don't change it (we
just add a suffix link to it)

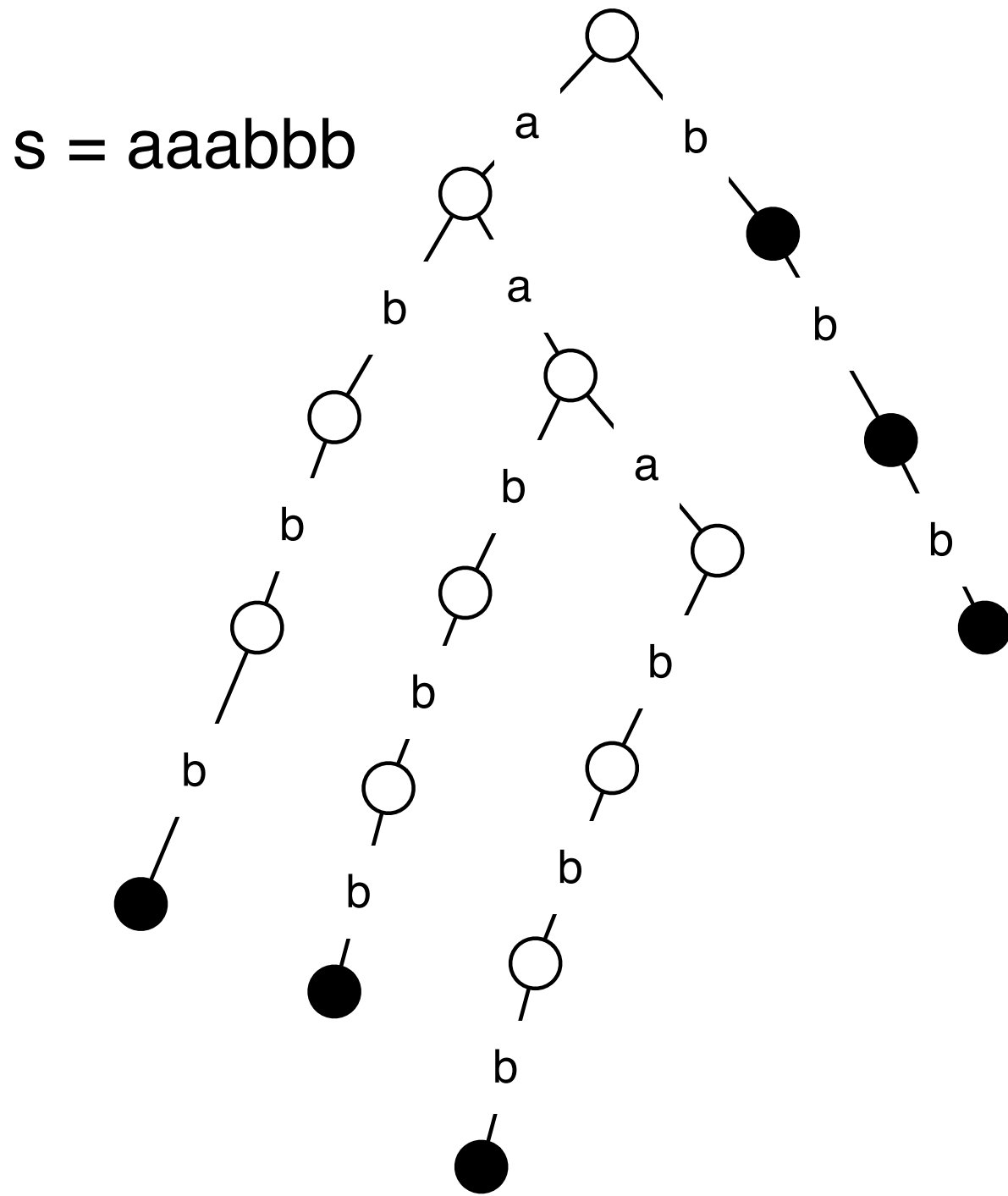abaa   abaab

abaaba   abaaba$
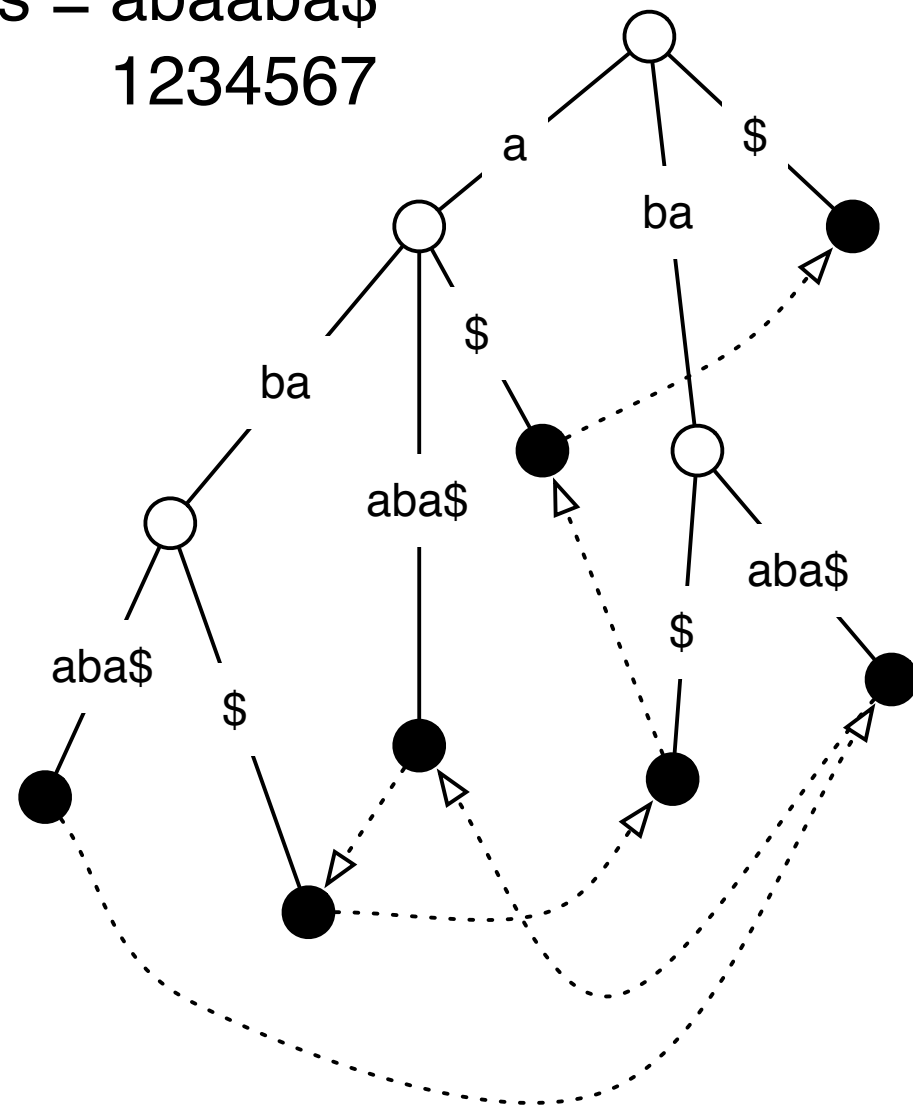
17

How many nodes can a suffix trie have?

s = aaabbb

- s = $a^n b^n$ will have
  - 1 root node
  - n nodes in a path of "b"s
  - n paths of n+1 "b" nodes

- Total = n(n+1)+n+1 = $O(n^2)$ nodes.

- This is not very efficient.

- How could you make it smaller?

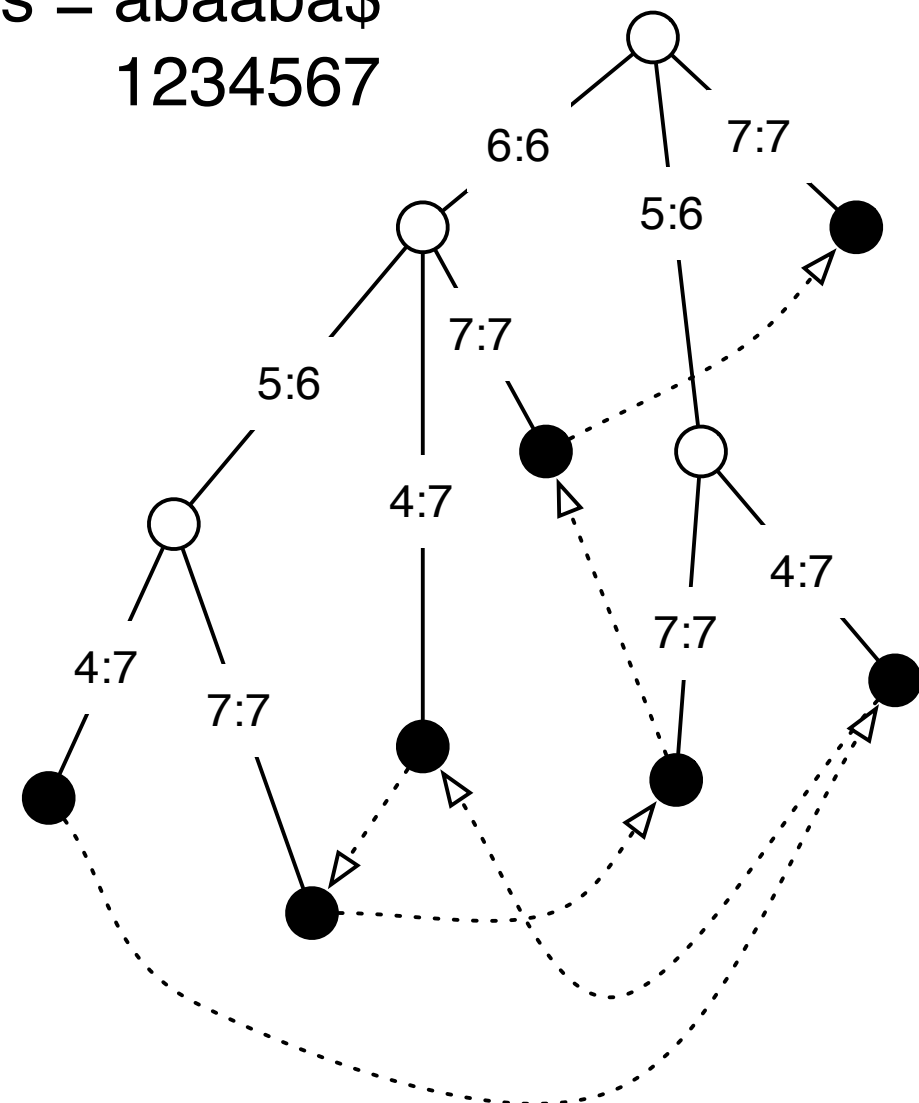So... we have to "trie" again...

Space-Efficient Suffix Trees

A More Compact Representation

s = abaaba$
1234567

s = abaaba$
1234567

- Compress paths where there are no choices.

- Represent sequence along the path using a range [i,j] that refers to the input string s.

Space usage:

- In the compressed representation:

    - # leaves = O(n) [one leaf for each position in the string]

    - Every internal node is at least a binary split.

    - Each edge uses O(1) space.

- Therefore, # number of internal nodes is about equal to the number of leaves.

- And # of edges ≈ number of leaves, and space per edge is O(1).

- Hence, linear space.

Constructing Suffix Trees – Ukkonen's Algorithm

- The same idea as with the suffix trie algorithm.

- Main difference: not every trie node is explicitly represented in the tree.

- Solution: represent trie nodes as pairs (u, α), where u is a real node in the tree and α is some string leaving it.

- Some additional tricks to get to O(n) time. (We'll talk about these later.)

s = abab

abab

bab

u

v
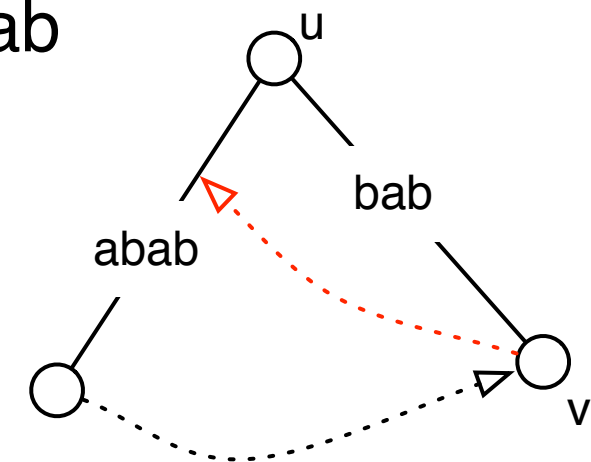
suffix_link[v] = (u, ab)

Storing more than one string with
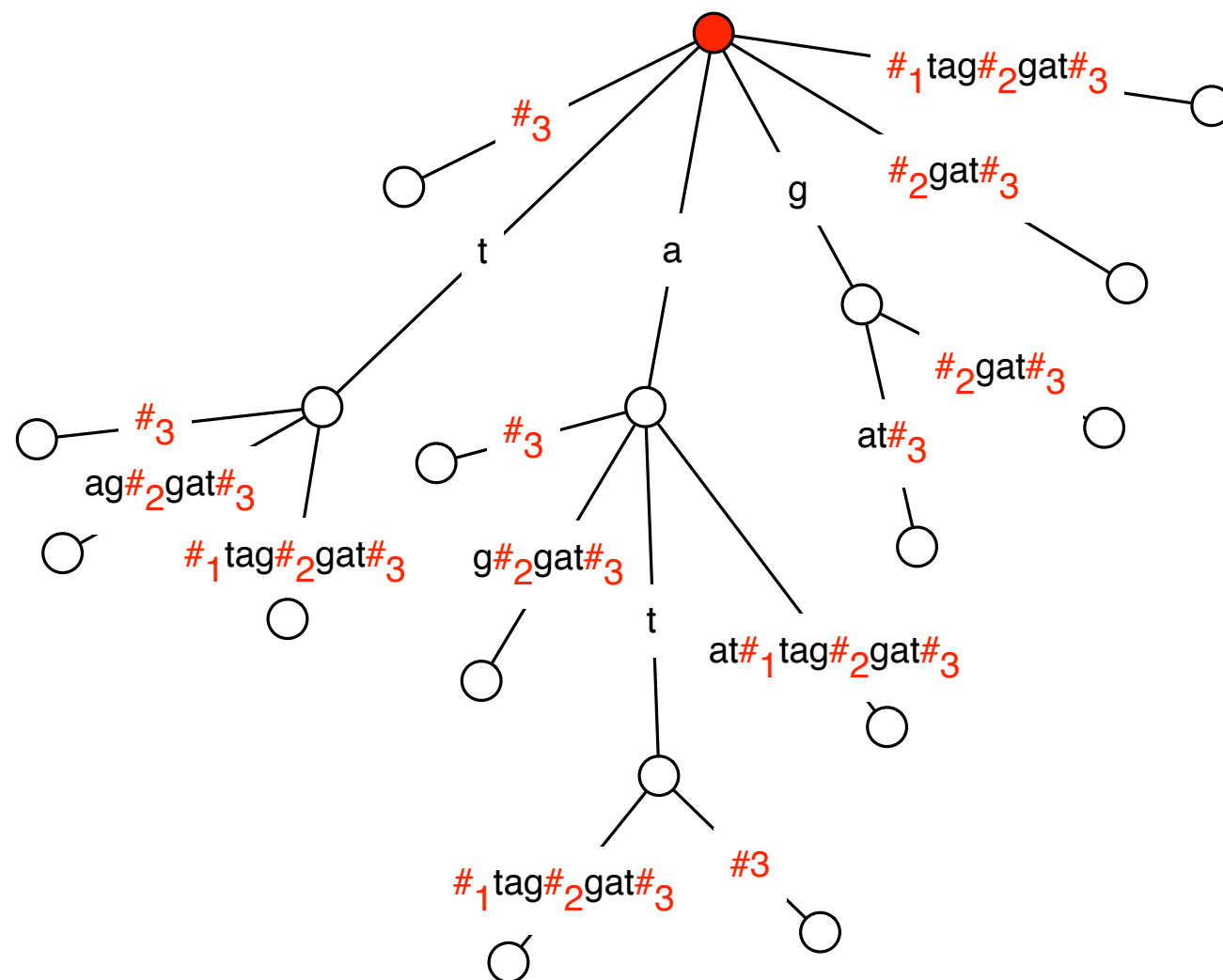## Generalized Suffix Trees

Constructing Generalized Suffix Trees

Goal. Represent a set of strings P = {s_1, s_2, s_3, ..., s_m}.

Example. att, tag, gat

Simple solution:
   (1) build suffix tree for string aat#$_1$tag#$_2$gat#$_3$
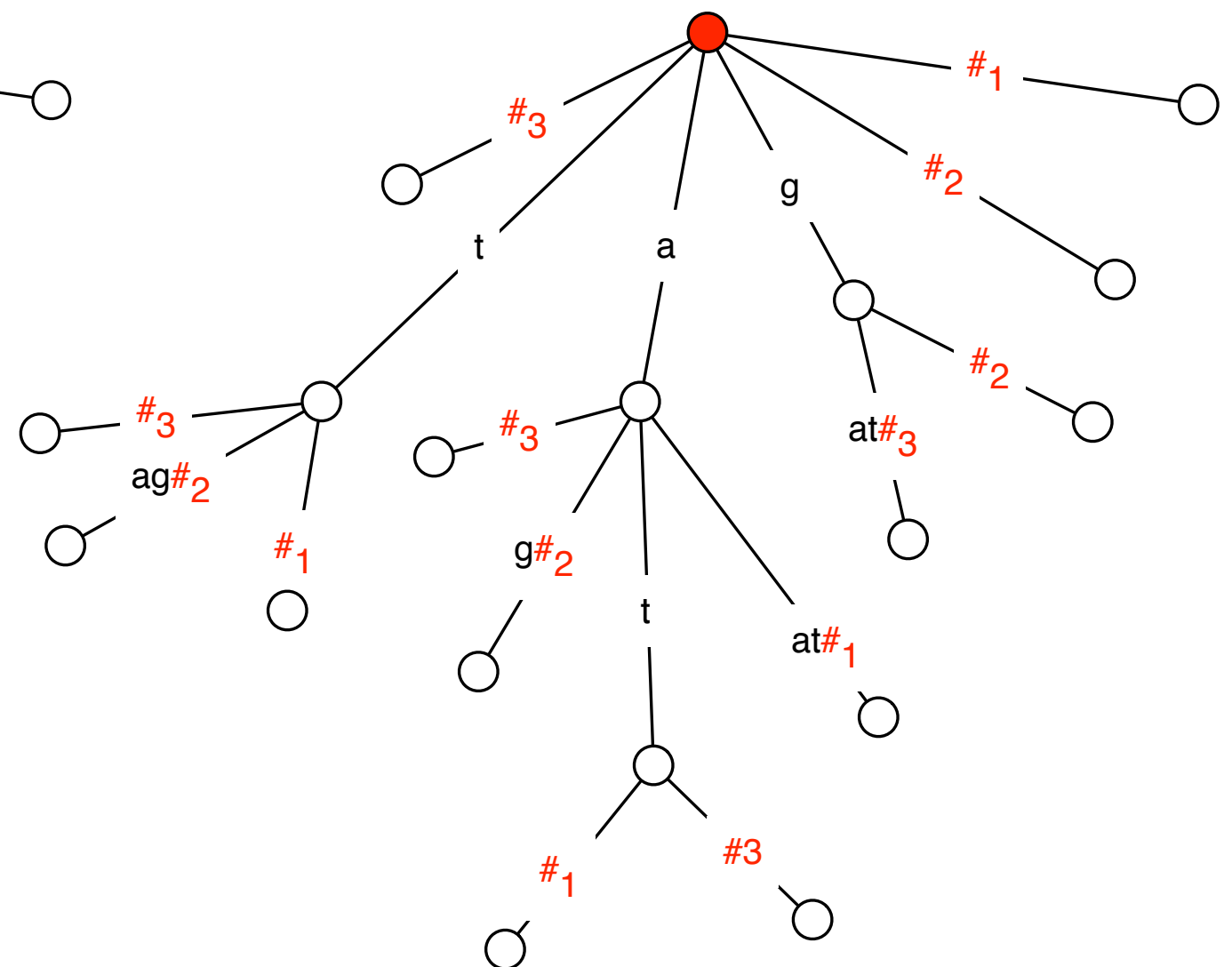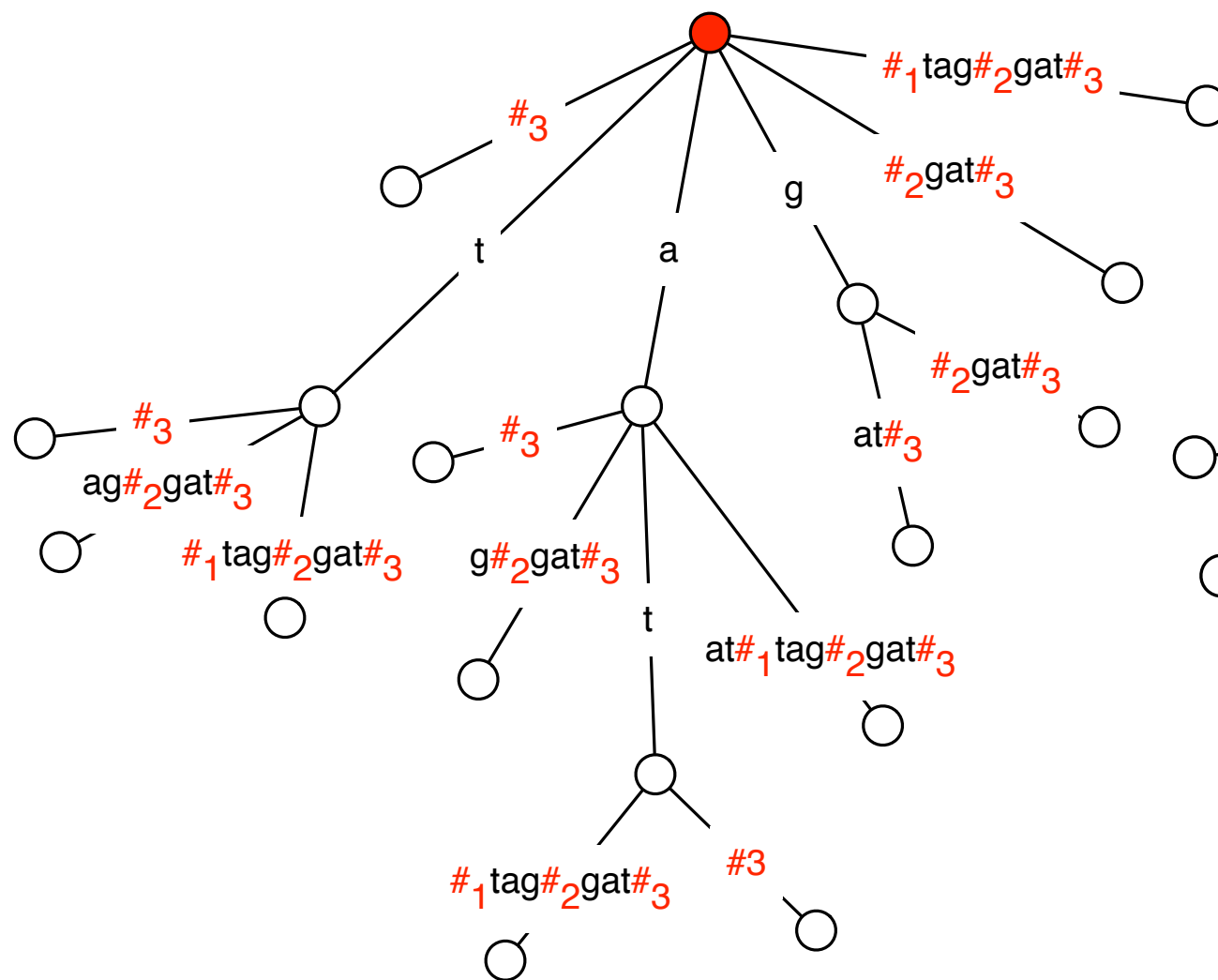
Constructing Generalized Suffix Trees

Goal. Represent a set of strings P = {s₁, s₂, s₃, ..., sₘ}.

Example. att, tag, gat

Simple solution:
   (1) build suffix tree for string aat#₁tag#₂gat#₃

(2) For every leaf node, remove any text after the first # symbol.

24

Applications of Generalized Suffix Trees

Longest common substring of S and T:

Determine the strings in a database $\{S_1, S_2, S_3, ..., S_m\}$ that contain query string q:

Applications of Generalized Suffix Trees

Longest common substring of S and T:

Build generalized suffix tree for {S, T}
Find the deepest node that has has descendants from both strings (containing both $\#_1$ and $\#_2$)

Determine the strings in a database $\{S_1, S_2, S_3, ..., S_m\}$ that contain query string q:

Applications of Generalized Suffix Trees

Longest common substring of S and T:

Build generalized suffix tree for {S, T}
Find the deepest node that has has descendants from both strings (containing both $\#_1$ and $\#_2$)

Determine the strings in a database {S_1, S_2, S_3, ..., S_m} that contain query string q:

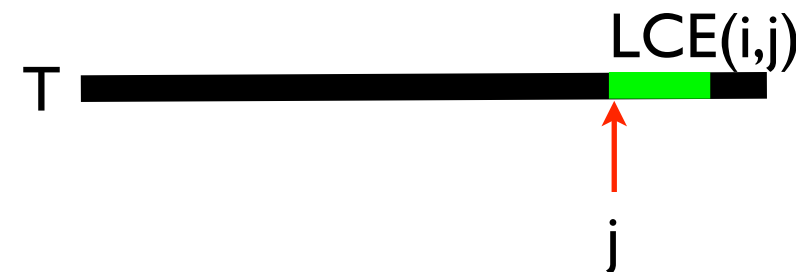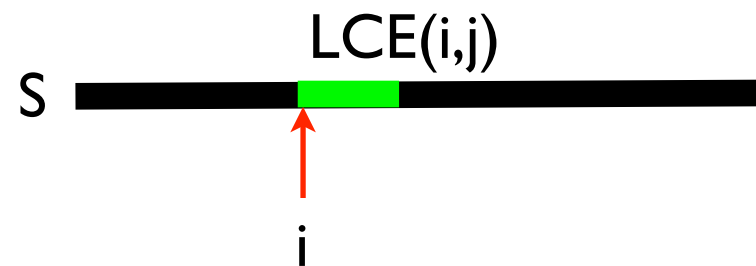Build generalized suffix tree for {S_1, S_2, S_3, ..., S_m}
Follow the path for q in the suffix tree.
Suppose you end at node u: traverse the tree below u, and output i if you find a string containing $\#_i$.

Longest Common Extension

Longest common extension: We are given strings S and T. In the future, many pairs (i,j) will be provided as queries, and we want to quickly find:

the longest substring of S starting at i that matches a substring of T starting at j.
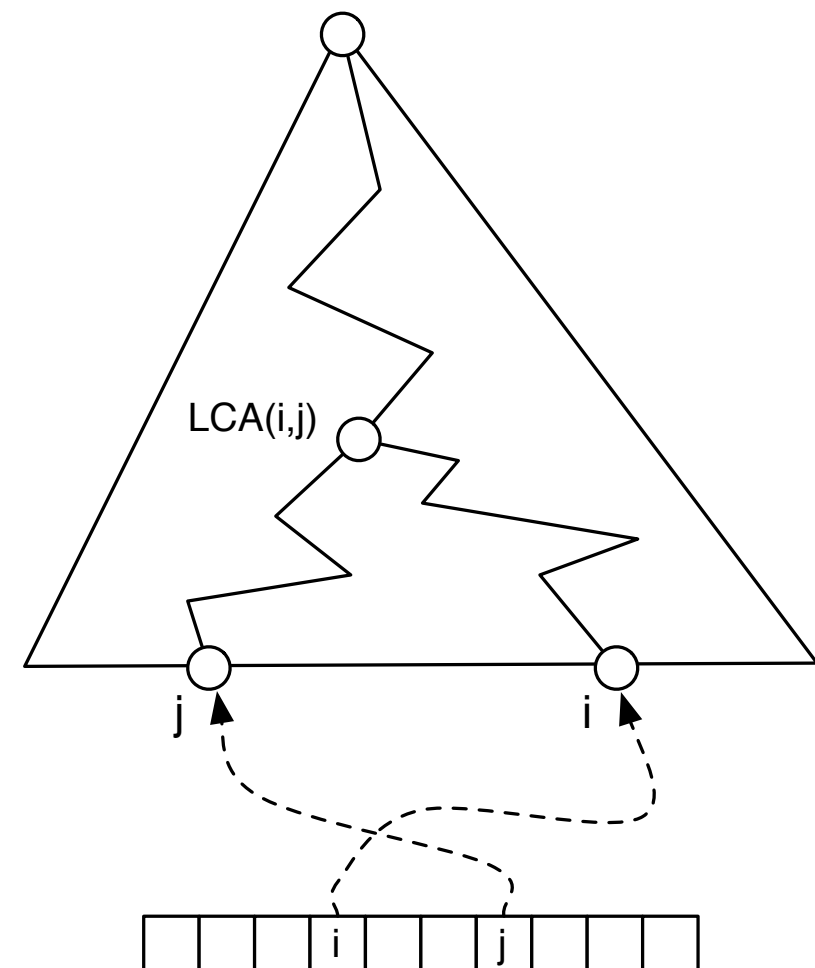
Build generalized suffix tree for S and T.

Preprocess tree so that lowest common ancestors (LCA) can be found in constant time.

Create an array mapping suffix numbers to leaf nodes.
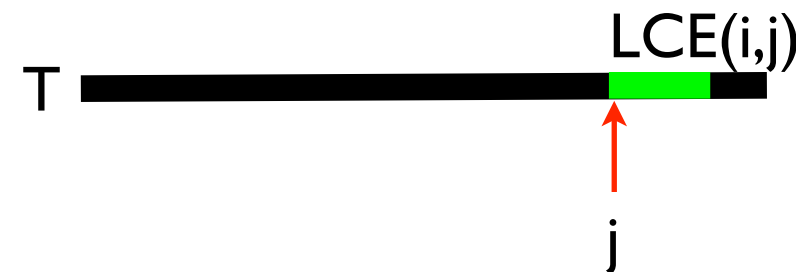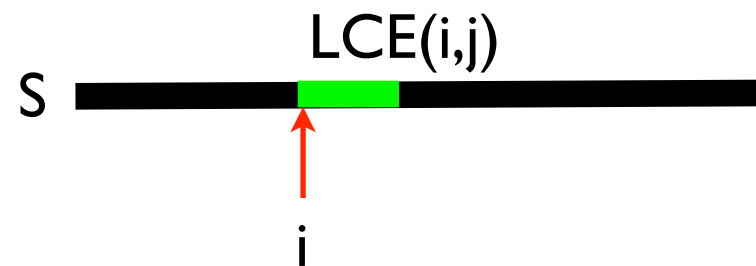
Given query (i,j):
    Find the leaf nodes for i and j
    Return string of LCA for i and j

26

Longest Common Extension

Longest common extension: We are given strings S and T. In the future, many pairs (i,j) will be provided as queries, and we want to quickly find:

the longest substring of S starting at i that matches a substring of T starting at j.

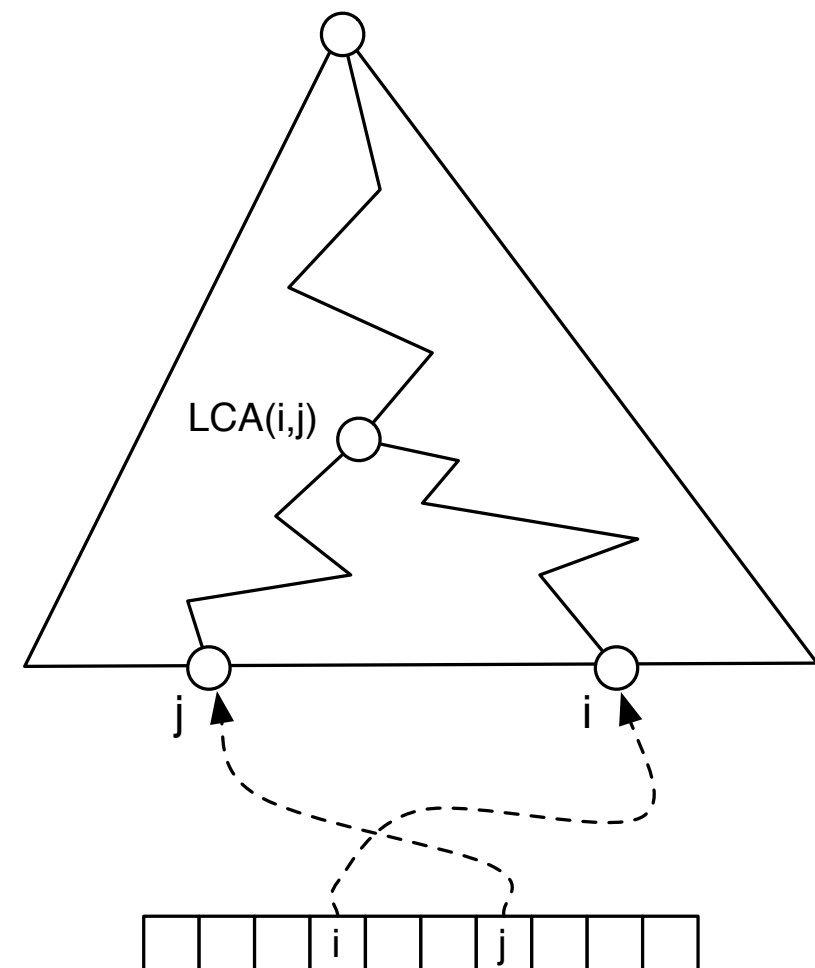Build generalized suffix tree for S and T.   O(|S| + |T|)

Preprocess tree so that lowest common ancestors (LCA) can be found in constant time.   O(|S| + |T|)

Create an array mapping suffix numbers to leaf nodes.   O(|S| + |T|)

Given query (i,j):
    Find the leaf nodes for i and j   O(1)
    Return string of LCA for i and j   O(1)

26

Using LCE to Find Palindromes

Maximal even palindrome at position i: the longest string to the left and right so that the left half is equal to the reverse of the right half.

$x \neq y$

S

x   y

i

= the reverse of

Goal: find all maximal palindromes in S.

Using LCE to Find Palindromes

Maximal even palindrome at position i: the longest string to the left and right so that the left half is equal to the reverse of the right half.

Goal: find all maximal palindromes in S.

Construct S^r, the reverse of S.

Preprocess S and S^r so that LCE queries can be solved in constant time (previous slide).

LCE(i, n-i) is the length of the longest palindrome centered at i.

For every position i:
    Compute LCE(i, n-i)

27

Using LCE to Find Palindromes

Maximal even palindrome at position i: the longest string to the left and right so that the left half is equal to the reverse of the right half.

Goal: find all maximal palindromes in S.

Construct S^r, the reverse of S. $O(|S|)$

Preprocess S and S^r so that LCE queries can be solved in constant time (previous slide). $O(|S|)$

LCE(i, n-i) is the length of the longest palindrome centered at i.

For every position i: $O(|S|)$
    Compute LCE(i, n-i) $O(1)$

Total time = $O(|S|)$

Recap

- Suffix tries natural way to store a string -- search, count occurrences, and many other queries answerable easily.

- But they are not space efficient: $O(n^2)$ space.

- Suffix trees are space optimal: $O(n)$, but require a little more subtle algorithm to construct.

- Suffix trees can be constructed in $O(n)$ time using Ukkonen's algorithm.

- Similar ideas can be used to store sets of strings.