Algorithms and data structures for read mapping

Based on
Lecture notes from Ron Shamir (Tel aviv Univ.)
And
Carl Kingsford (CMU)

Read mapping problem

- Input:
  - Reference genome R (human: $3*10^9$ bp)
  - Set of reads $S_1, S_2, \ldots, S_m$ ($m = 10^9$, $|S_i| = 100$)
- Output
  - For each read S_i, the position in R that matches S_i (possibly allowing for a small number of mismatches (SNPs, errors))

Solutions

- Naïve :
  - For each S_i
    - For each position $p=l,\ldots,|R|$
      - Try matching S_i to the substring $R[p-l+1,\ldots,p]$

- Complexity:
  $O(lm|R|)$ exact or inexact matching

Solutions (2)

- Less Naïve:
  - For each S_i
    - Match S_i to R using KMP [Knuth-Morris-Pratt]
- Complexity:

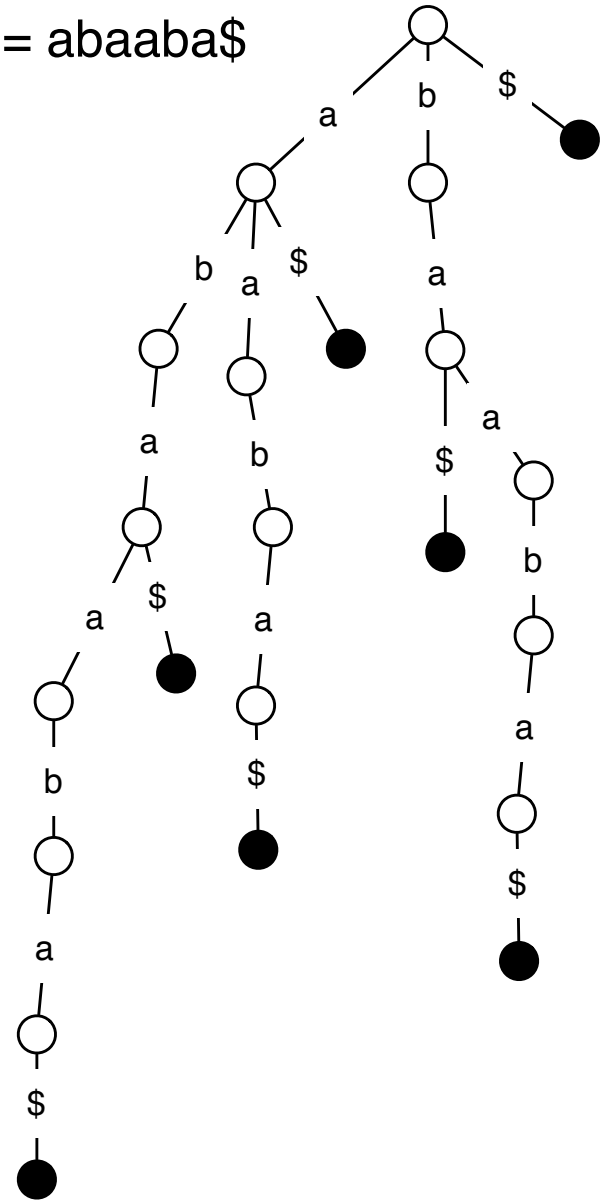
  $O(m(l+|R|))= O(ml+m|R|)$ exact matching

Solutions (3)

- Suffix tree approach:
  - Build suffix tree for R
  - For each S_i
    - Find matches of S_i to R by tree traversal from the root
- Time complexity: $O(lm+|R|)$ exact matching
- Space Complexity: $O(|R|\log|R|)$ vs $|R|\log|\Sigma|$ for the text
- Can store Human Genome text in 750M bytes (6G bits) but, need ~64G bytes for the tree
  - large constants, hard to implement

Suffix Tries

s = abaaba$



SufTrie(s) = suffix trie representing string s.

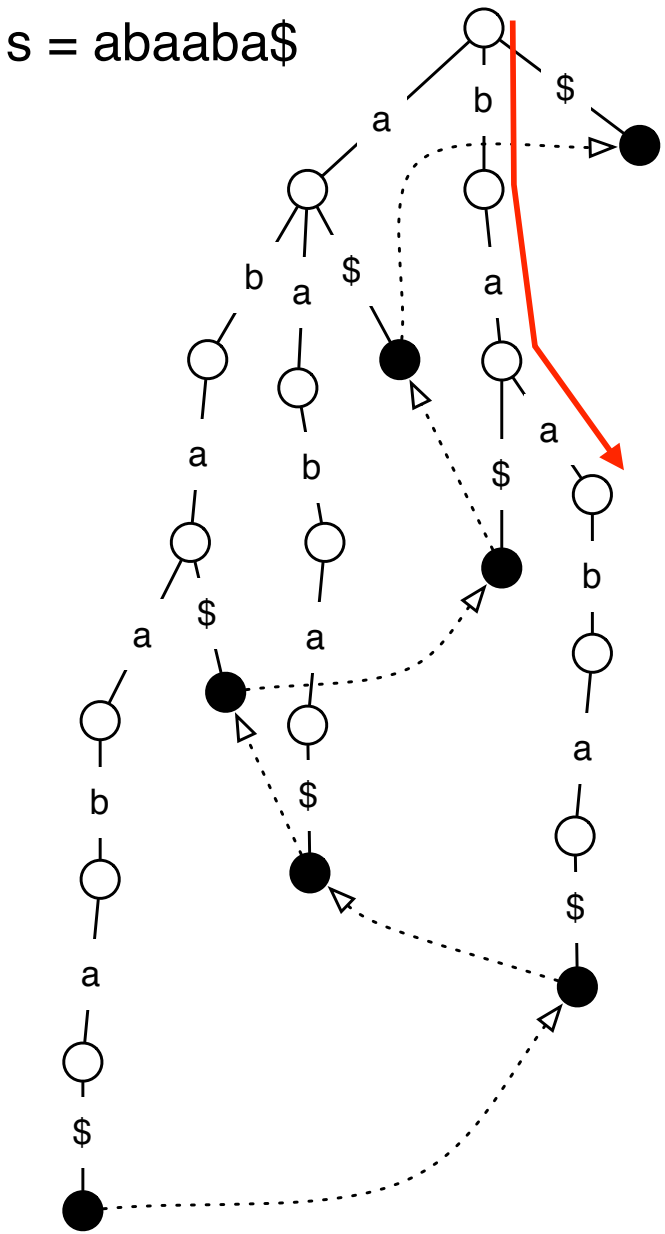
Edges of the suffix trie are labeled with letters from the alphabet ∑ (say {A,C,G,T}).

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.

How many leaves will there be?

Searching Suffix Tries



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:

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.

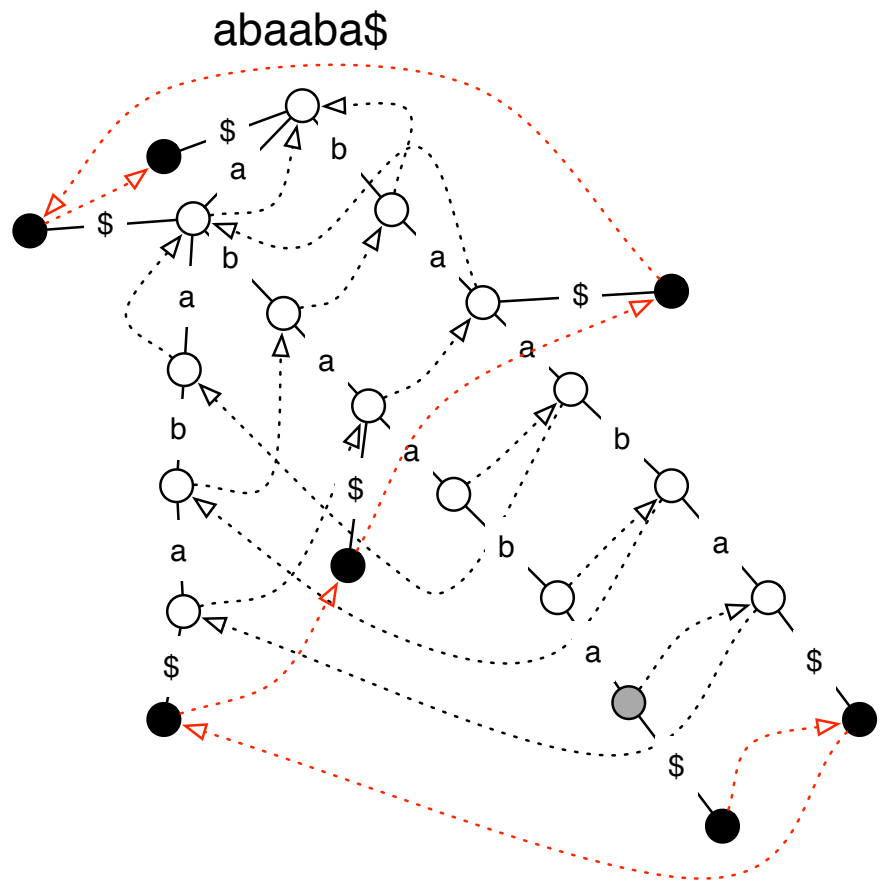
Applications of Suffix Tries (II)

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

q = abbaaa



Building a suffix trie

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 $\underbrace{s[0], s[0..1], s[0..2], ..., s[0..n]}$

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?

abbacabaa

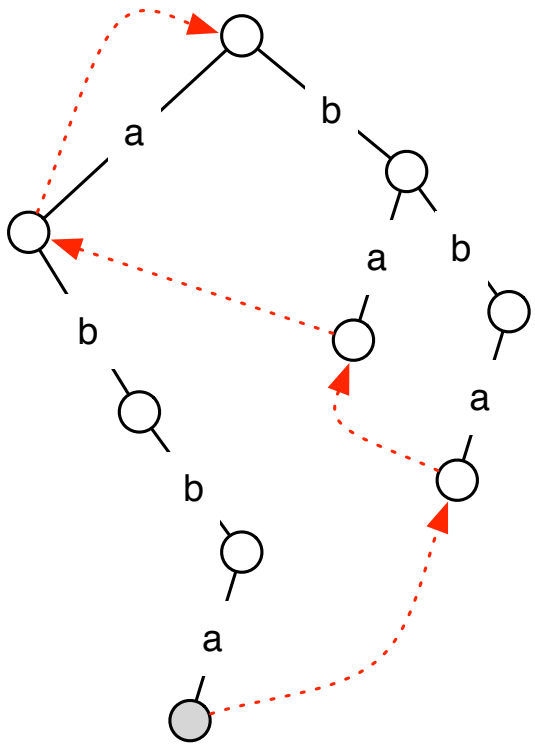
i=4

Need to add nodes for the suffixes:

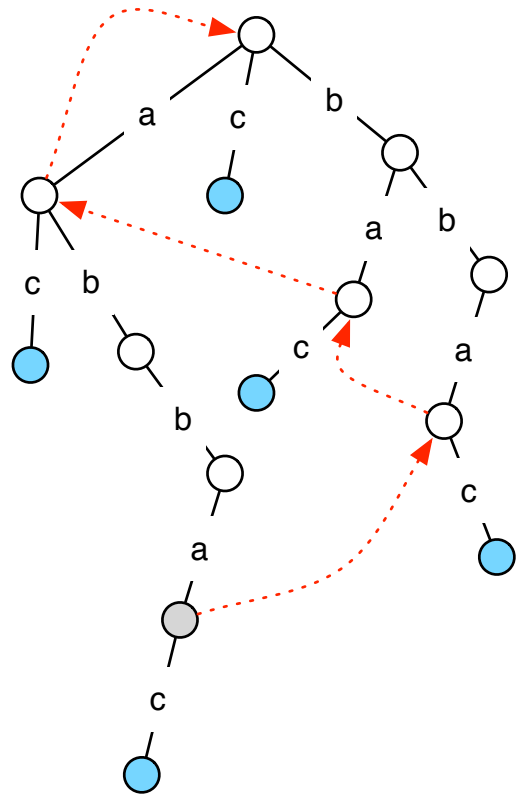
abbac
bbac
bac
ac
c

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

How can we find these suffixes quickly?



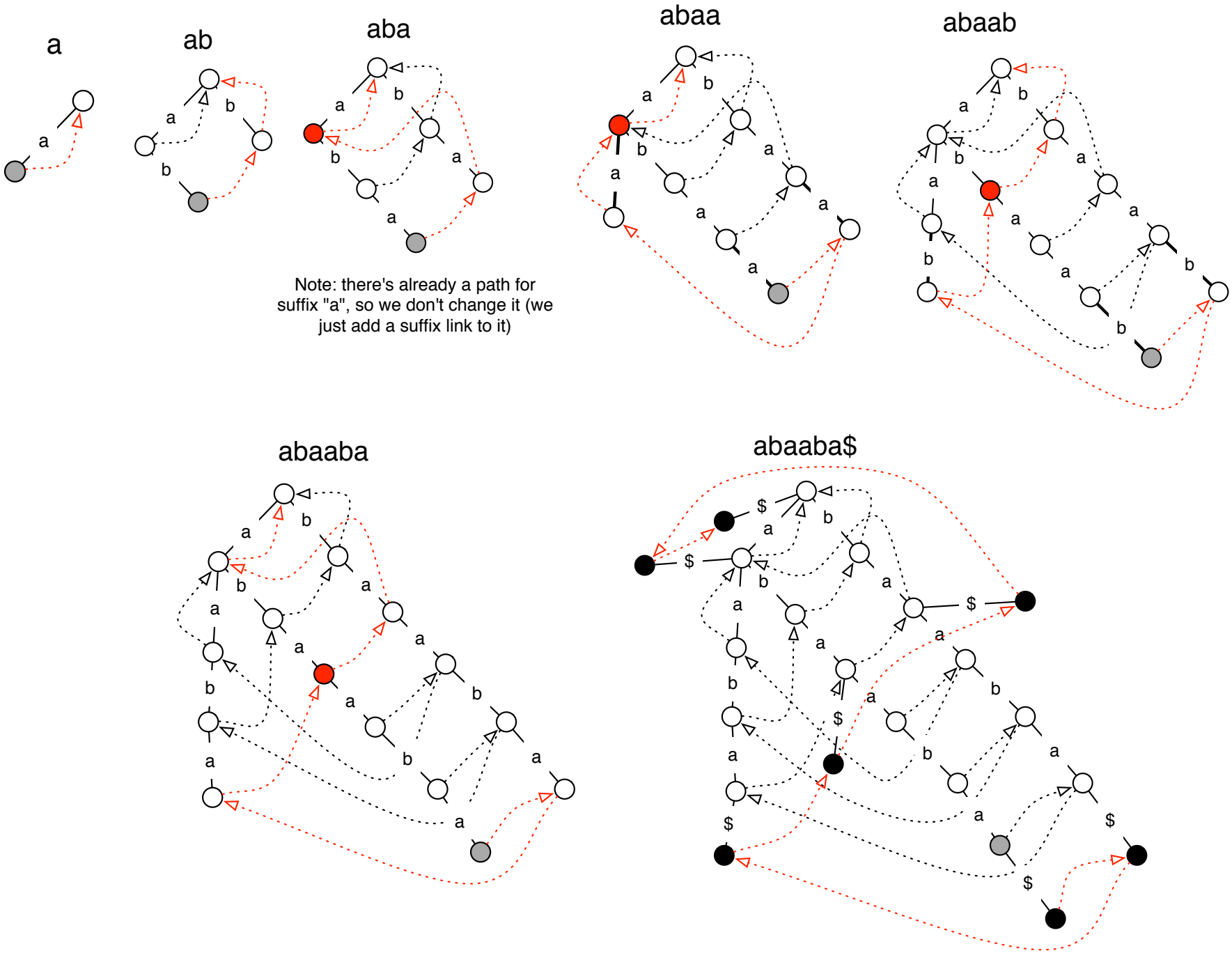
SufTrie(abba)



SufTrie(abbac)

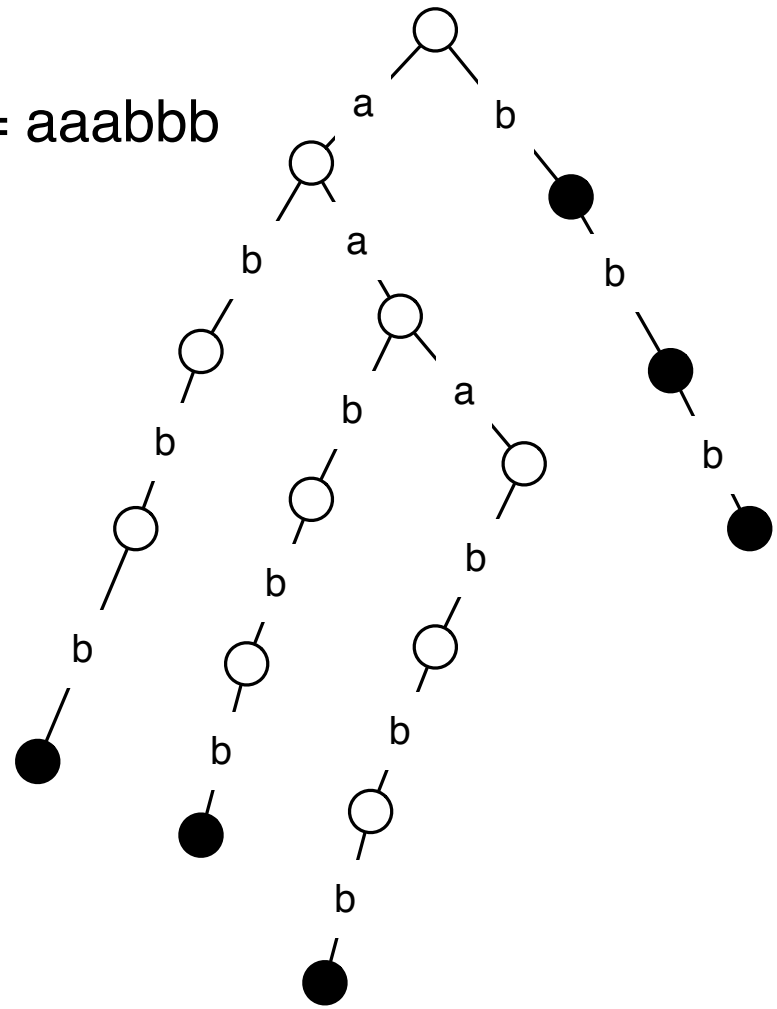
Where is the new deepest node? (aka longest suffix)

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


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)
abaaba
abaaba$

How many nodes can a suffix trie have?

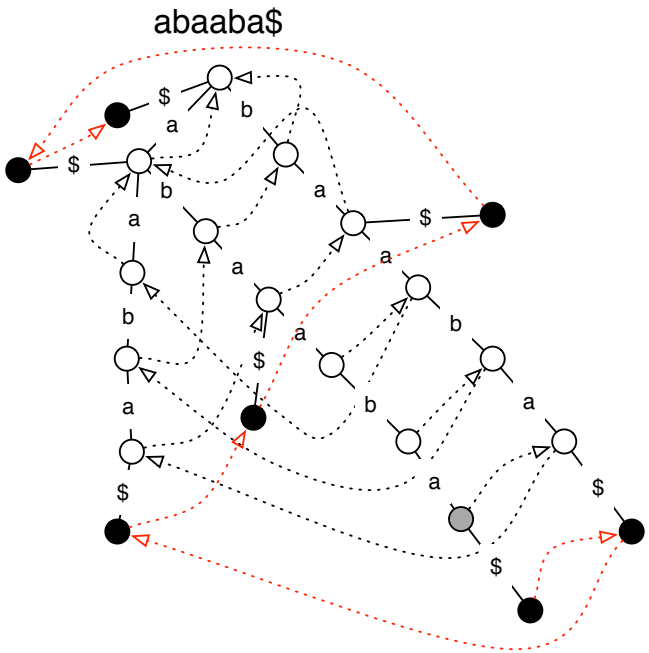
s = aaabbb

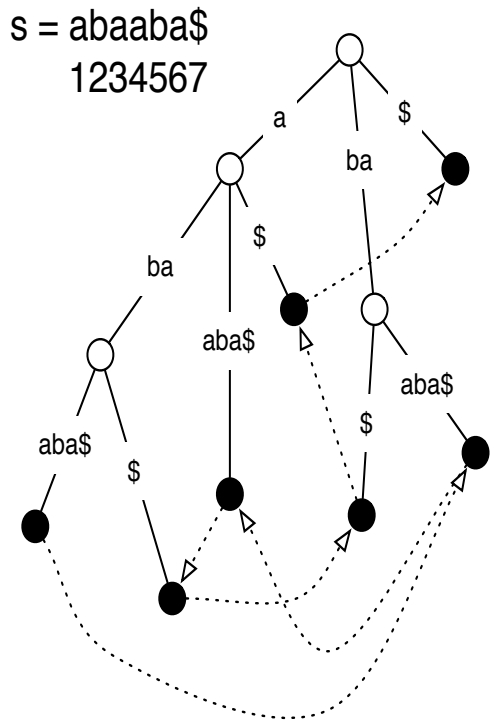


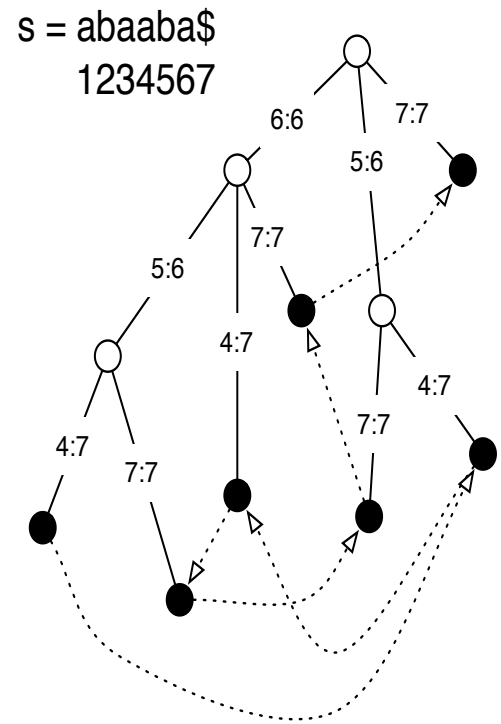
- $s = a^n b^n$ will have
  - 1 root node
  - n nodes in a path of "b"s
  - n paths of n+1 "b" nodes
- Total = $n(n+1)+n+1 = O(n^2)$ nodes.
- This is not very efficient.
- How could you make it smaller?

Suffix trees

A More Compact Representation







- Compress paths where there are no choices.
- Represent sequence along the path using a range [i,j] that refers to the input string s.

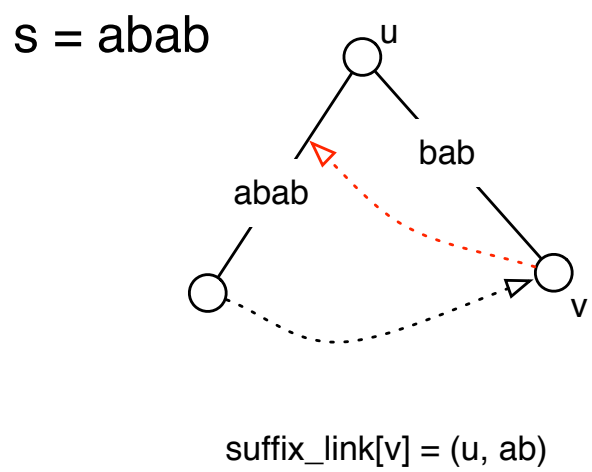
Space usage:

- 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.



Suffix Arrays

- Even though Suffix Trees are O(n) space, the constant hidden by the big-Oh notation is somewhat "big": ≈ 20 bytes / character in good implementations.

- If you have a 10Gb genome, 20 bytes / character = 200Gb to store your suffix tree. "Linear" but large.

- Suffix arrays are a more efficient way to store the suffixes that can do most of what suffix trees can do, but just a bit slower.

- Slight space vs. time tradeoff.

Example Suffix Array

s = attcatg$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

index of suffix	suffix of s
1	attcatg$
2	ttcatg$
3	tcatg$
4	catg$
5	atg$
6	tg$
7	g$
8	$

sort the suffixes alphabetically

→

the indices just "come along for the ride"

8	$
5	atg$
1	attcatg$
4	catg$
7	g$
3	tcatg$
6	tg$
2	ttcatg$

Another Example Suffix Array

s = cattcat$

- Idea: lexicographically sort all the suffixes.
- Store the starting indices of the suffixes in an array.

index of suffix	suffix of s
1	cattcat$
2	attcat$
3	ttcat$
4	tcat$
5	cat$
6	at$
7	t$
8	$

sort the suffixes alphabetically

→

the indices just "come along for the ride"

8
6
2
5
1
7
4
3

Search via Suffix Arrays

s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

- Does string "at" occur in s? ← √
- Binary search to find "at". ←
- What about "tt"?

Counting via Suffix Arrays

s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

- How many times does "at" occur in the string?
- All the suffixes that start with "at" will be next to each other in the array.
- Find one suffix that starts with "at" (using binary search).
- Then count the neighboring sequences that start with at.

Constructing Suffix Arrays

- Easy $O(n^2 \log n)$ algorithm:

  sort the n suffixes, which takes $O(n \log n)$ comparisons, where each comparison takes $O(n)$.

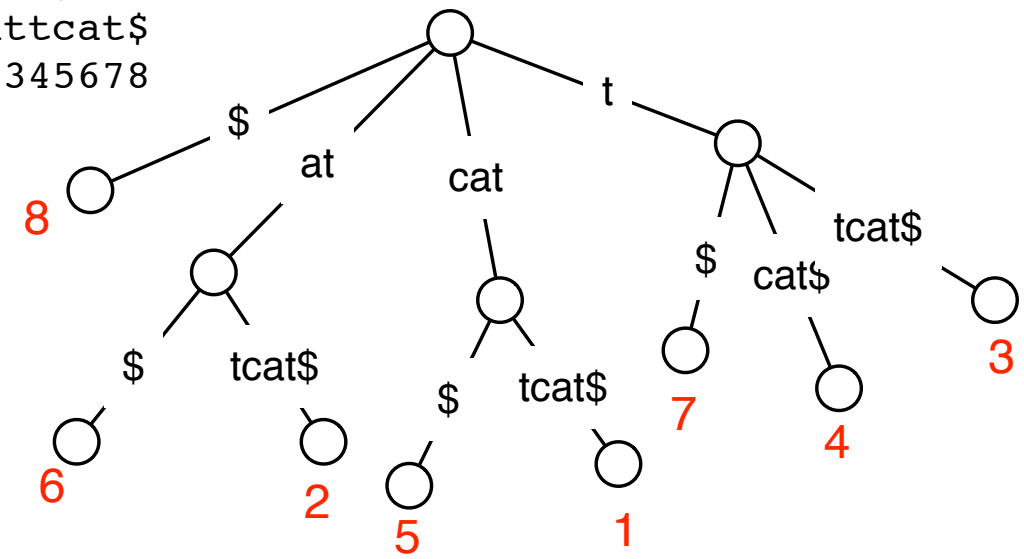
- There are several direct $O(n)$ algorithms for constructing suffix arrays that use very little space.
- The Skew Algorithm is one that is based on divide-and-conquer.

- An simple $O(n)$ algorithm: build the suffix tree, and exploit the relationship between suffix trees and suffix arrays (next slide)

Relationship Between Suffix Trees & Suffix Arrays

$\Sigma = \{\$,a,c,t\}$

s = `cattcat$`
`12345678`



s = cattcat$

8	$
6	at$
2	attcat$
5	cat$
1	cattcat$
7	t$
4	tcat$
3	ttcat$

Red #s = starting position of the suffix ending at that leaf

Leaf labels left to right: 86251743

Edges leaving each node are sorted by label (left-to-right).