Lesson 03 Genome Informatics

School of Electrical Engineering
Feb 28th 2019

Recapitulation

$$\begin{tikzcd}
 & \text{Exact string matching algorithms} \arrow[dl, "\text{on-line}"'] \arrow[dr, "\text{off-line}"] & \\
\text{Boyer-Moore} & & \text{Multimap (sorted index)}
\end{tikzcd}$$

Boyer-Moore

Multimap (sorted index)

Hash table

Suffix array

Suffix trie

Suffix tree

Burrows Wheeler FM index

Tries

A trie (pronounced “try”) is a tree representing a collection of strings with one node per common prefix

Smallest tree such that:

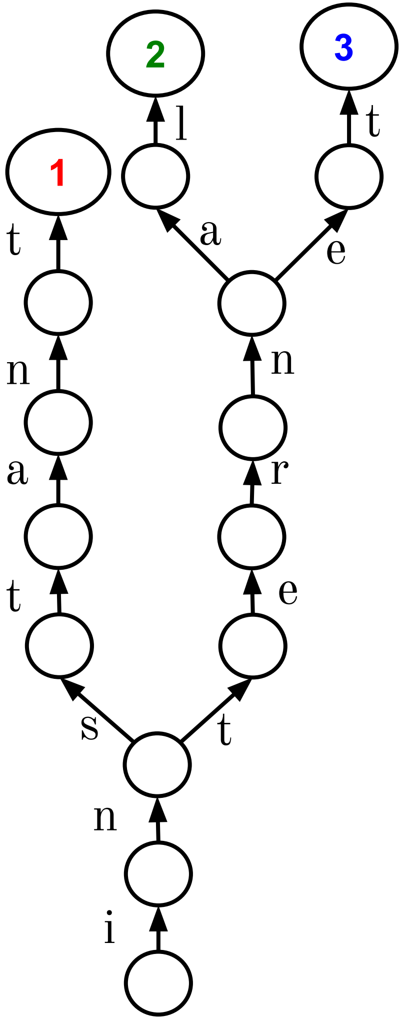
- Each **edge** is labeled with a character $c \in \Sigma$
- A **node** has at most one outgoing edge labeled c, for $c \in \Sigma$
- Each key is “spelled out” along some path starting at the root

Natural way to represent either a set or a map where keys are strings

Tries: example

Represent this map with a trie:

key	value
instant	1
internal	2
internet	3

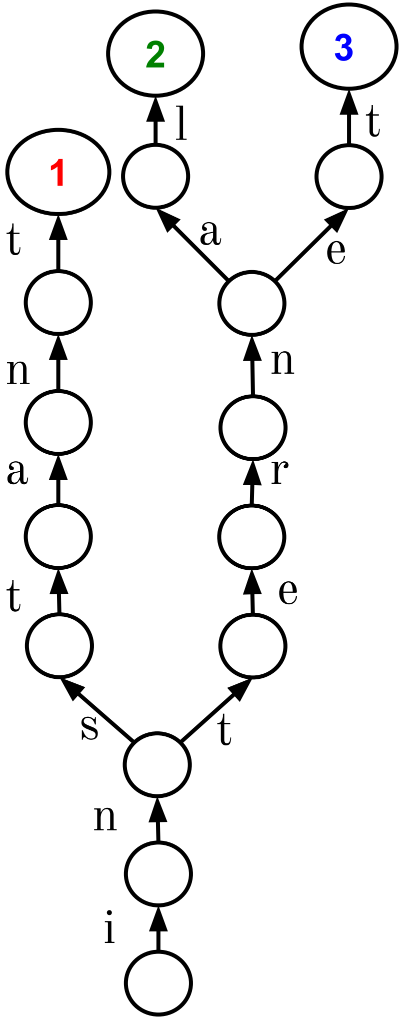


- Each **edge** is labeled with a character c ∈ Σ
- A **node** has at most one outgoing edge labeled c, for c ∈ Σ
- Each key is "spelled out" along some path starting at the root

Tries: example

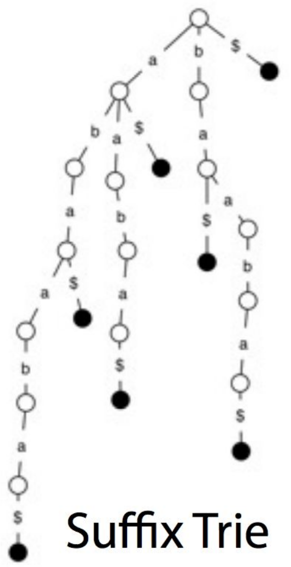
- Checking for presence of a key P, where n = | P |, is O(n) time
- If total length of all keys is N, trie has O(N) nodes

Tries implementation

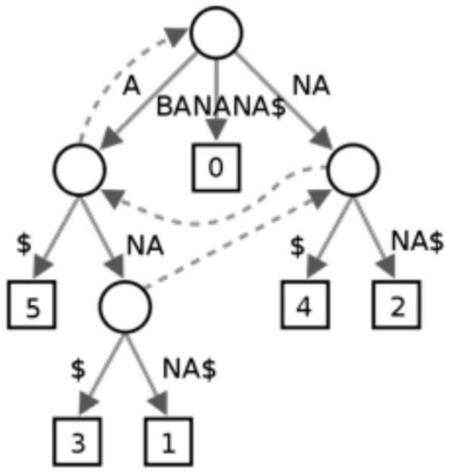


Indexing with suffixes

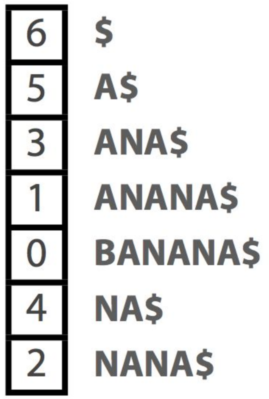
- Until now, our indexes have been based on extracting substrings from T
  A very different approach is to extract suffixes from T. This will lead us to some interesting and practical index data structures



Suffix Trie

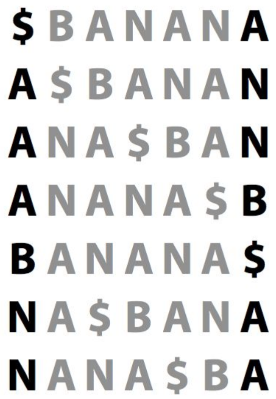


Suffix Tree



6	$
5	A$
3	ANA$
1	ANANA$
0	BANANA$
4	NA$
2	NANA$

Suffix Array



$ B A N A N A
A $ B A N A N
A N A $ B A N
A N A N A $ B
B A N A N A $
N A $ B A N A
N A N A $ B A

FM Index

Suffix Trie

Build a **trie** containing all **suffixes** of a text T!

T = GTTATAGCTGATCGCGGCGTAGCGG$

```
GTTATAGCTGATCGCGGCGTAGCGG$
 TTATAGCTGATCGCGGCGTAGCGG$
  TATAGCTGATCGCGGCGTAGCGG$
   ATAGCTGATCGCGGCGTAGCGG$
    TAGCTGATCGCGGCGTAGCGG$
     AGCTGATCGCGGCGTAGCGG$
      GCTGATCGCGGCGTAGCGG$
       CTGATCGCGGCGTAGCGG$
        TGATCGCGGCGTAGCGG$
         GATCGCGGCGTAGCGG$
          ATCGCGGCGTAGCGG$
           TCGCGGCGTAGCGG$
            CGCGGCGTAGCGG$
             GCGGCGTAGCGG$
              CGGCGTAGCGG$
               GGCGTAGCGG$
                GCGTAGCGG$
                 CGTAGCGG$
                  GTAGCGG$
                   TAGCGG$
                    AGCGG$
                     GCGG$
                      CGG$
                       GG$
                        G$
                         $
```

n(n+1)/2 chars

Suffix Trie

First add special terminal character \$ to the end of T

\$ is a character that does not appear elsewhere in T, and we define it to be less than other characters (for DNA: $\$ < A < C < G < T$)

\$ enforces a rule we're all used to using: e.g. "as" comes before "ash" in the dictionary.

\$ guarantees no suffix is a prefix of any other suffix.

T = GTTATAGCTGATCGCGGCGTAGCGG\$

```
GTTATAGCTGATCGCGGCGTAGCGG$
 TTATAGCTGATCGCGGCGTAGCGG$
  TATAGCTGATCGCGGCGTAGCGG$
   ATAGCTGATCGCGGCGTAGCGG$
    TAGCTGATCGCGGCGTAGCGG$
     AGCTGATCGCGGCGTAGCGG$
      GCTGATCGCGGCGTAGCGG$
       CTGATCGCGGCGTAGCGG$
        TGATCGCGGCGTAGCGG$
         GATCGCGGCGTAGCGG$
          ATCGCGGCGTAGCGG$
           TCGCGGCGTAGCGG$
            CGCGGCGTAGCGG$
             GCGGCGTAGCGG$
              CGGCGTAGCGG$
               GGCGTAGCGG$
                GCGTAGCGG$
                 CGTAGCGG$
                  GTAGCGG$
                   TAGCGG$
                    AGCGG$
                     GCGG$
                      CGG$
                       GG$
                        G$
                         $
```

n(n+1)/2 chars

Slide adapted from Ben Langmead

Tries

Smallest tree such that:

Each **edge** is labeled with a character from Σ

A **node** has at most one outgoing edge

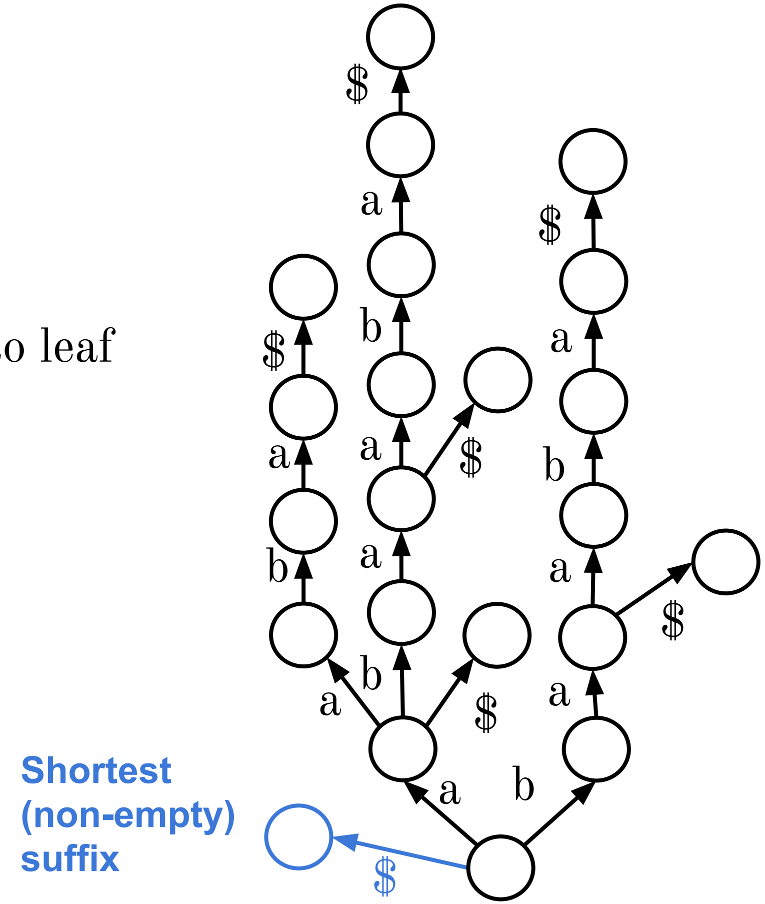
Each key is “spelled out” along some path starting at the root

Suffix trie

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 $?



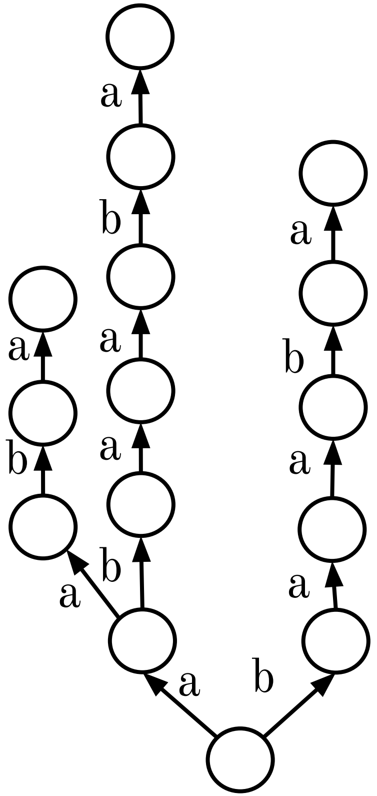
Suffix trie

T: abaaba   T$: abaaba$

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

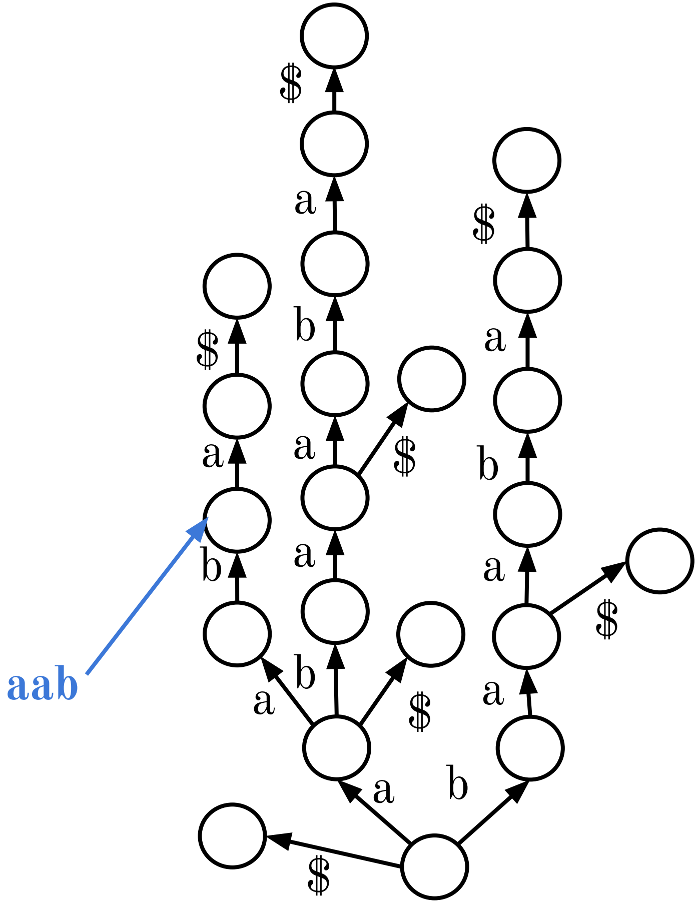
Would this still be the case if we hadn't added $?

No. Lost suffixes: "aba", "ba", etc.



Suffix trie

We can think of nodes as having labels, where the label spells out characters on the path from the root to the node



Suffix trie

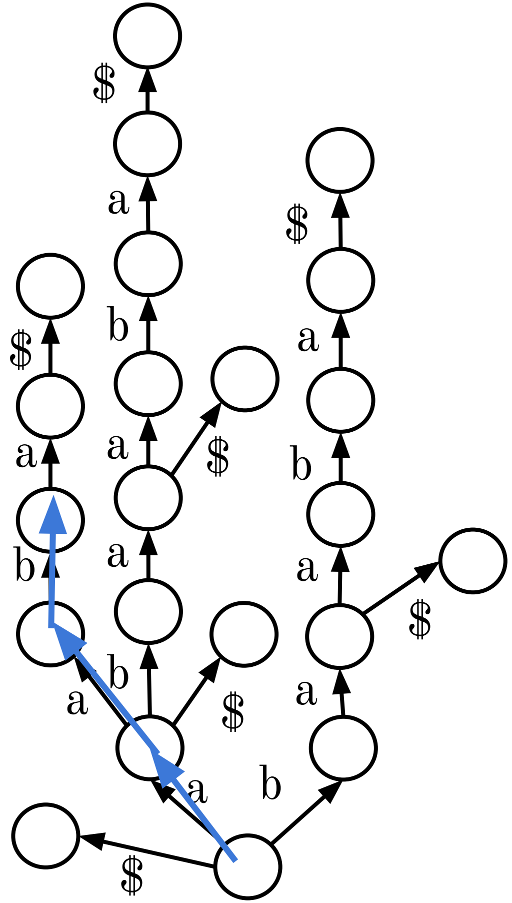
How do we check whether a string S is a substring of T?

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:

1. If we "fall off" the trie (there is no outgoing edge for next character of S) then S is not a substring of T
2. If we exhaust S without falling off, S is a substring of T

YES! "aab" is a substring of T



Suffix trie

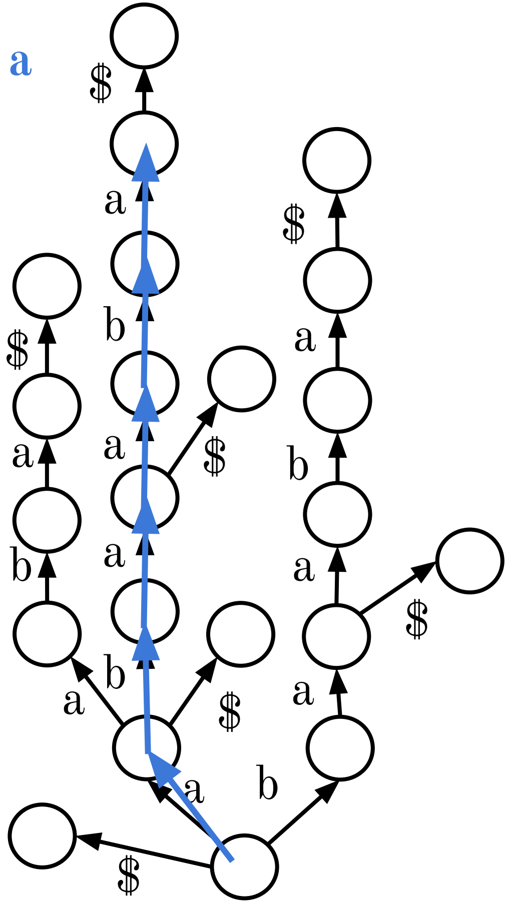
How do we check whether a string S is a substring of T?

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:

1. If we "fall off" the trie (there is no outgoing edge for next character of S) then S is not a substring of T
2. If we exhaust S without falling off, S is a substring of T

YES! "abaaba" is a substring of T



Suffix trie

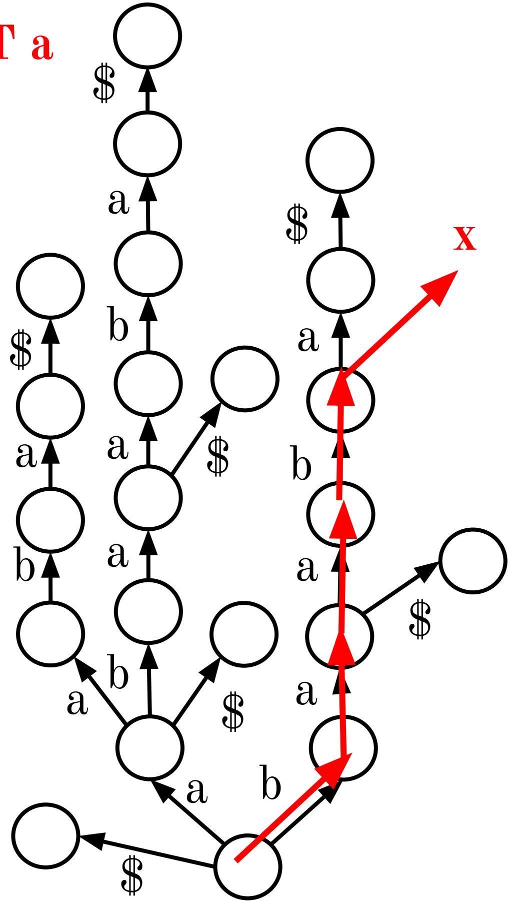
How do we check whether a string S is a substring of T?

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:

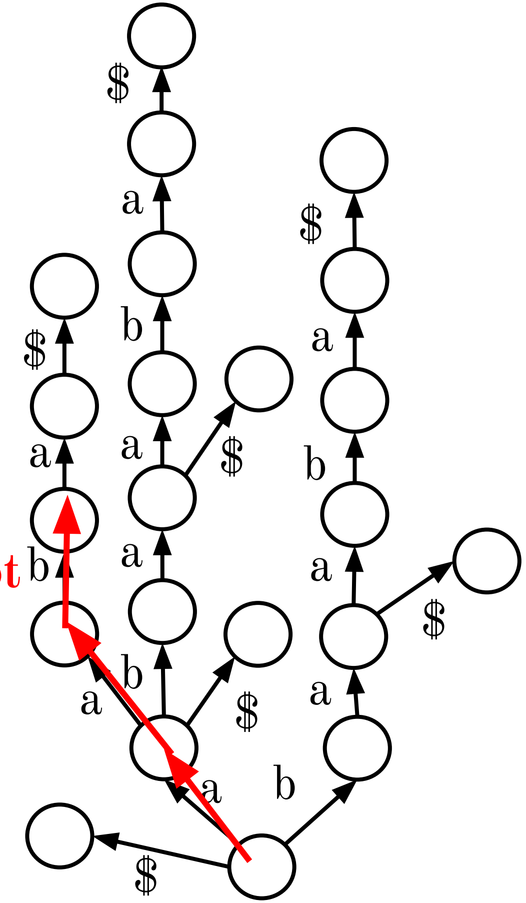
1. If we "fall off" the trie (there is no outgoing edge for next character of S) then S is not a substring of T
2. If we exhaust S without falling off, S is a substring of T

"baabb" is NOT a substring of T



Suffix trie

How do we check whether a string S is a suffix of T?
Same procedure as for substring, but additionally check whether the **final node** in the walk has an outgoing edge labeled $

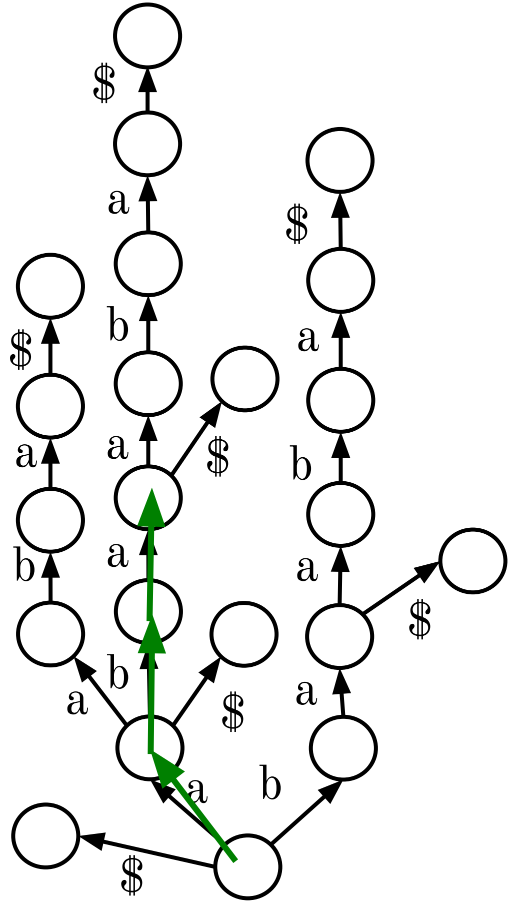


NO! "aab" is not a suffix of T

Suffix trie

How do we check whether a string S is a suffix of T?
Same procedure as for substring, but additionally check whether the **final node** in the walk has an outgoing edge labeled $

YES! "aba" is a suffix of T



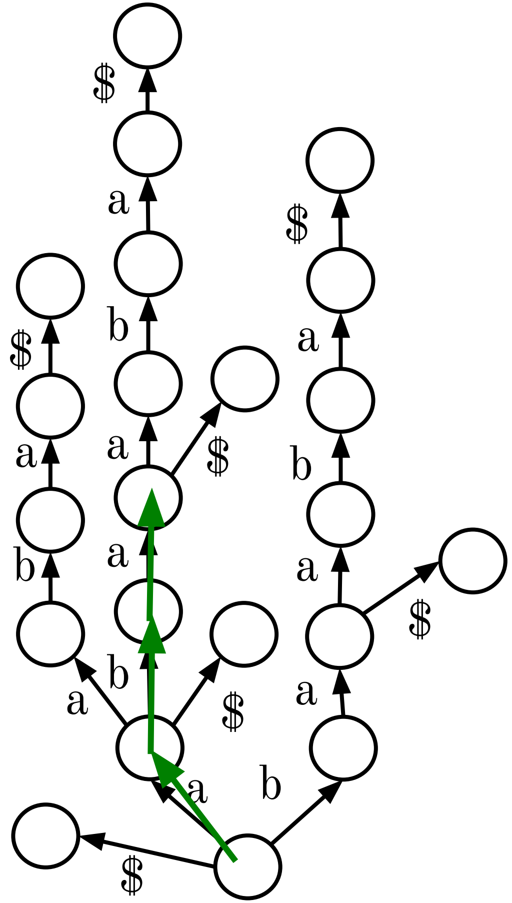
Suffix trie

How do we count the number of times a string S occurs as a substring of T?

Follow path corresponding to S. Either we fall off, in which case answer is 0, or we end up at node n and the answer = # of leaf nodes in the subtree rooted at n.

Leaves can be counted with depth-first traversal.

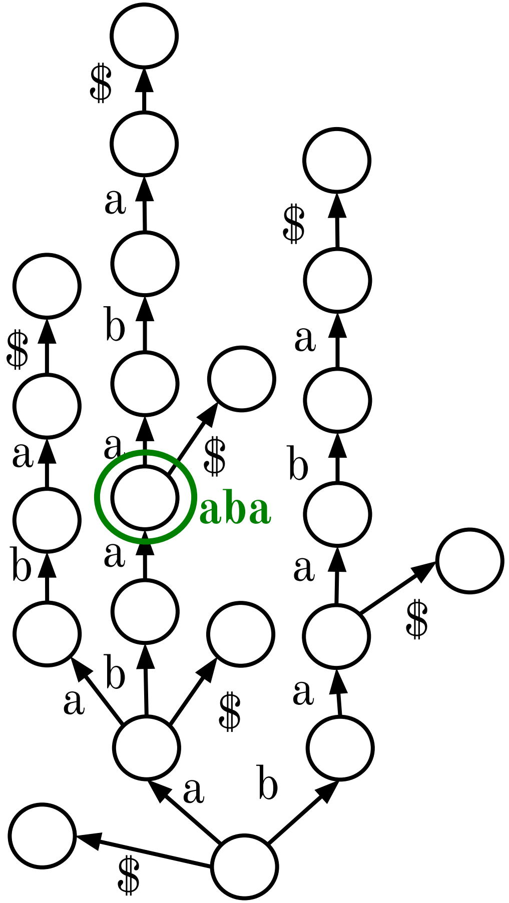
“aba” - 2 occurrences



Suffix trie

How do we find the **longest repeated substring** of T?

Find the deepest node with more than one child



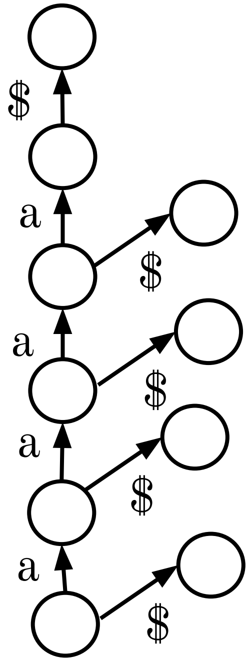
Suffix trie

How many nodes does the suffix trie have?

Is there a class of string where the number of suffix trie nodes grows linearly with m?

Yes: e.g. a string of m a's in a row

T = aaaa



Total nodes:

1 Root

m nodes with incoming **a** edge

m + 1 nodes with incoming $ edge

Total: 2m + 2 nodes

Suffix trie

Is there a class of string where the number of suffix trie nodes grows with $O(m^2)$?
Yes: $a^n b^n$

Total nodes:
- 1 root
- n nodes along "b chain," right
- n nodes along "a chain," middle
- n chains of n "b" nodes hanging off each"a chain" node
- $2n + 1$ \$ leaves (not shown)

Total: $n^2 + 4n + 2$ nodes, where $m = 2n$

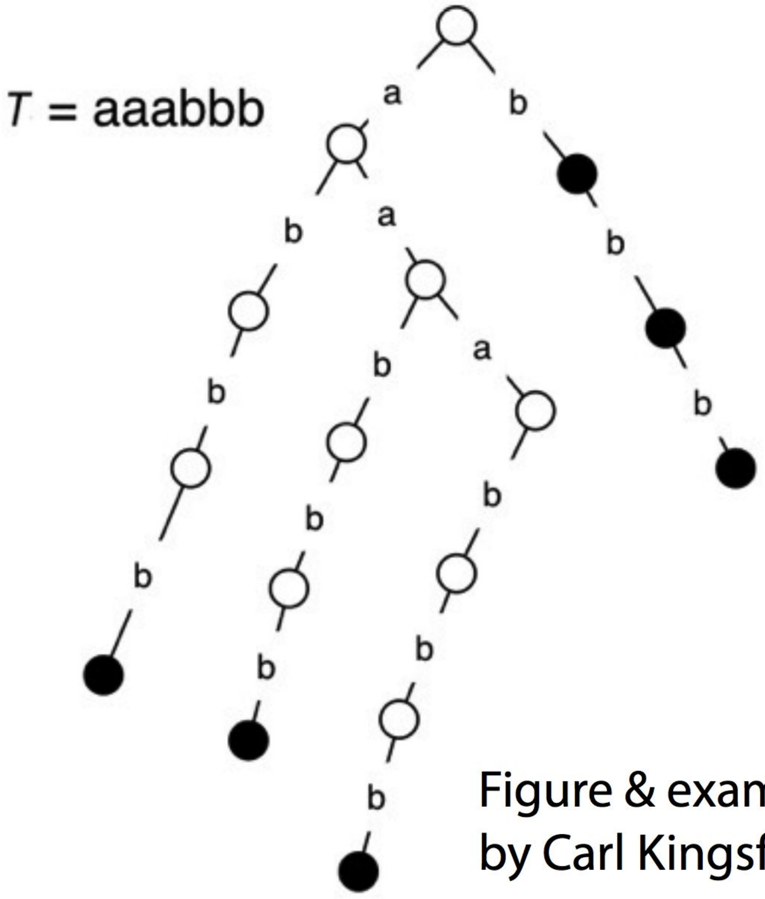


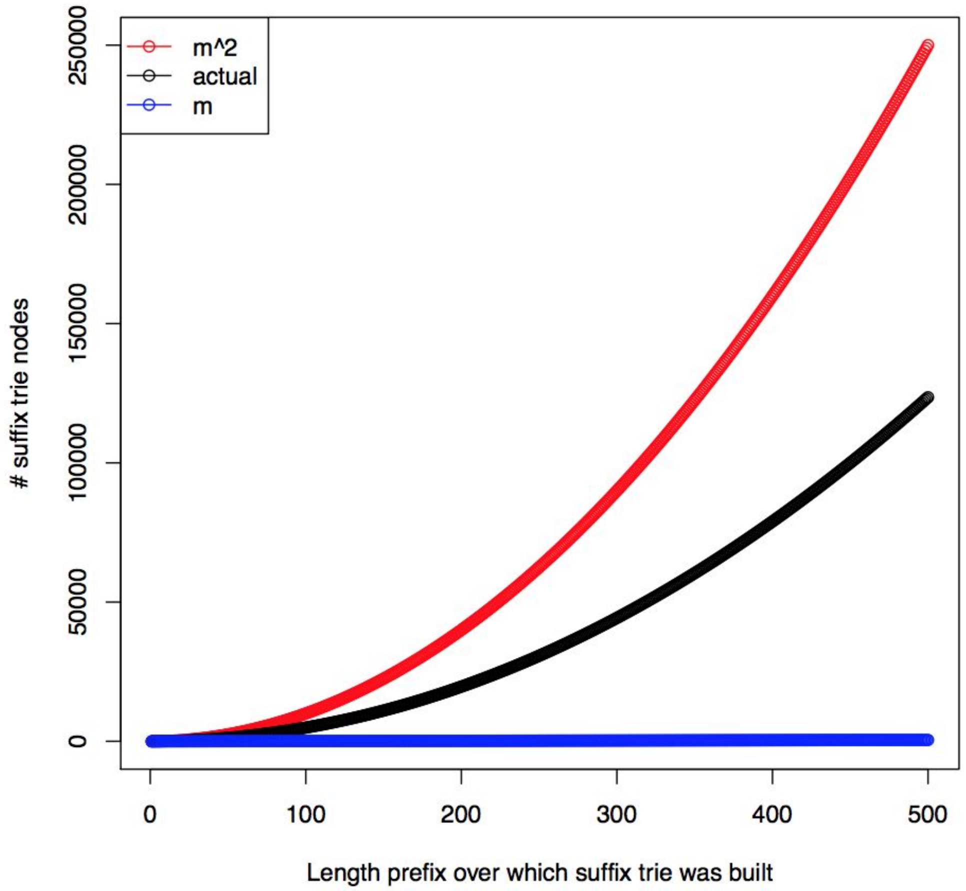
Figure & example by Carl Kingsford

Suffix trie

Built suffix tries for the first 500 prefixes of the lambda phage virus genome

Black curve shows how # nodes increases with prefix length

Suffix trie implementation

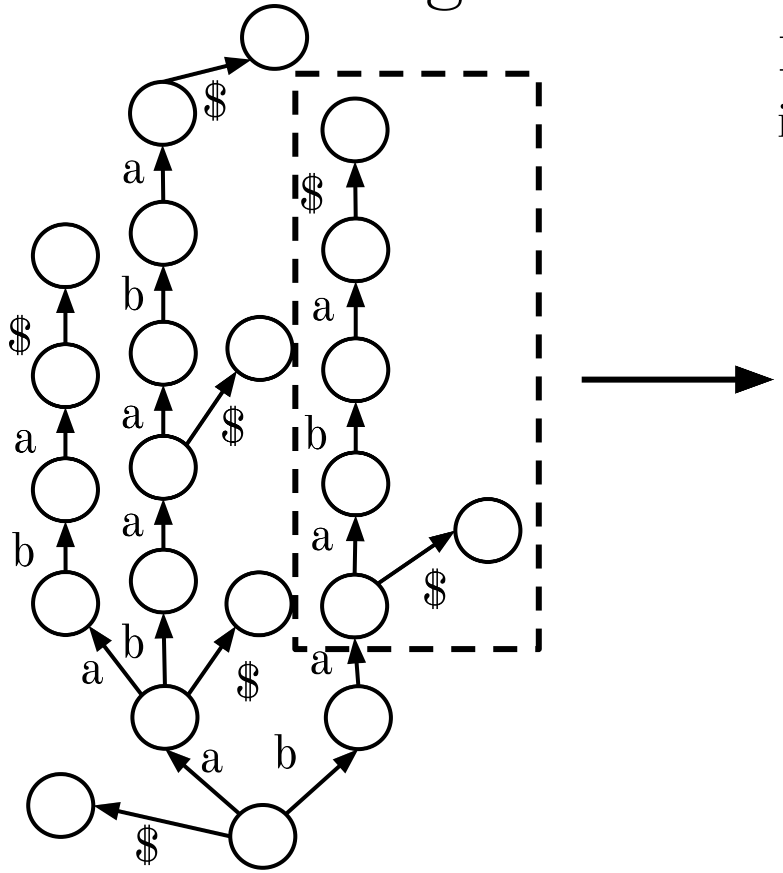


Suffix Tree

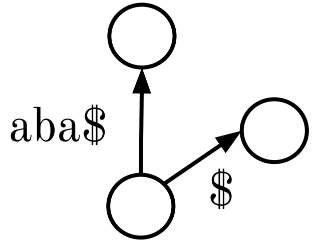
Lesson 3.2

Suffix trie: making it smaller

T: abaaba$

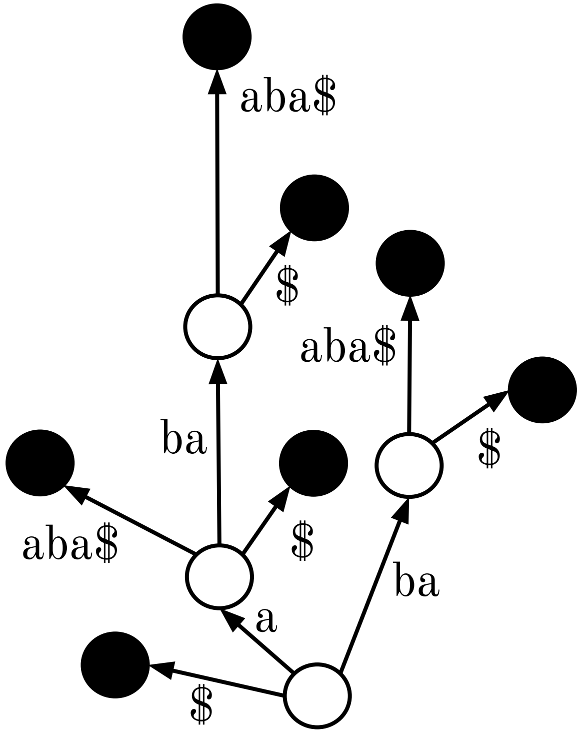


Idea 1: Coalesce non-branching paths into a single edge with a string label



Reduces # nodes, edges, guarantees internal nodes have >1 child

Suffix trie: making it smaller

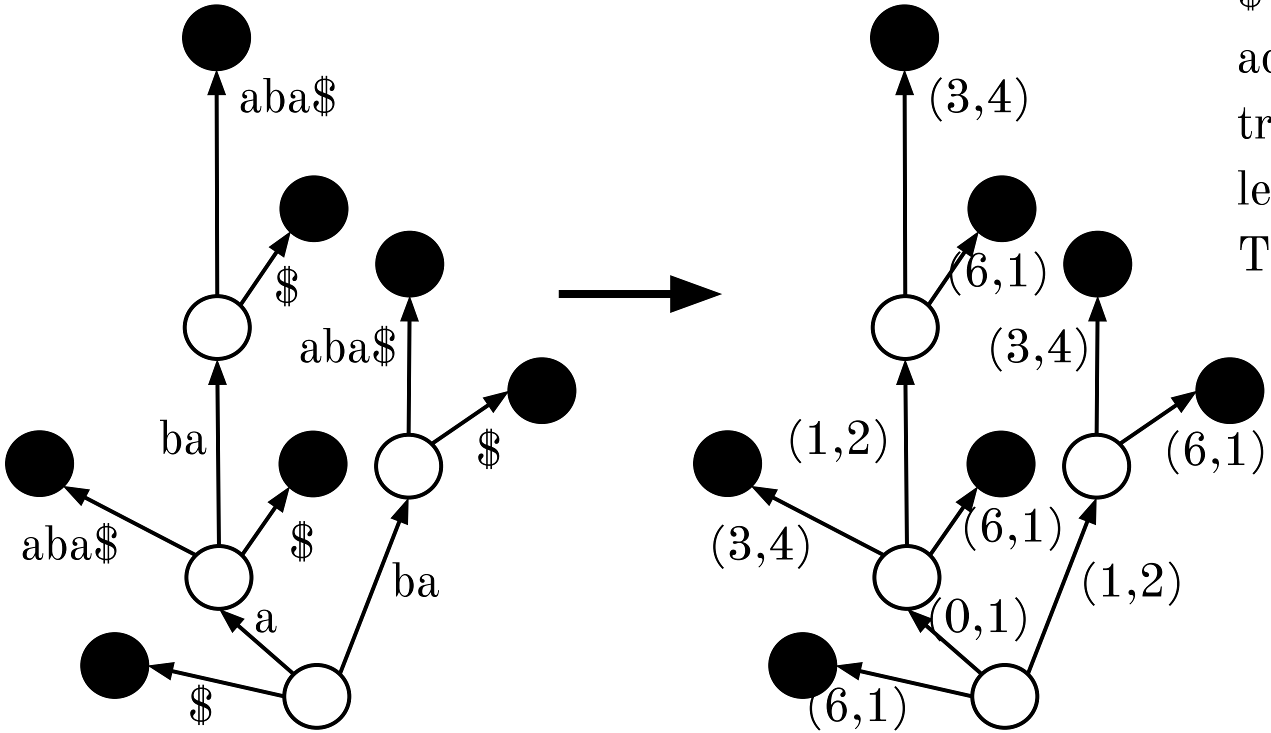


How many leaves? M

How many non-leaf nodes? $\leq$ m - 1

$\leq$ 2m -1 nodes total - O(m) nodes

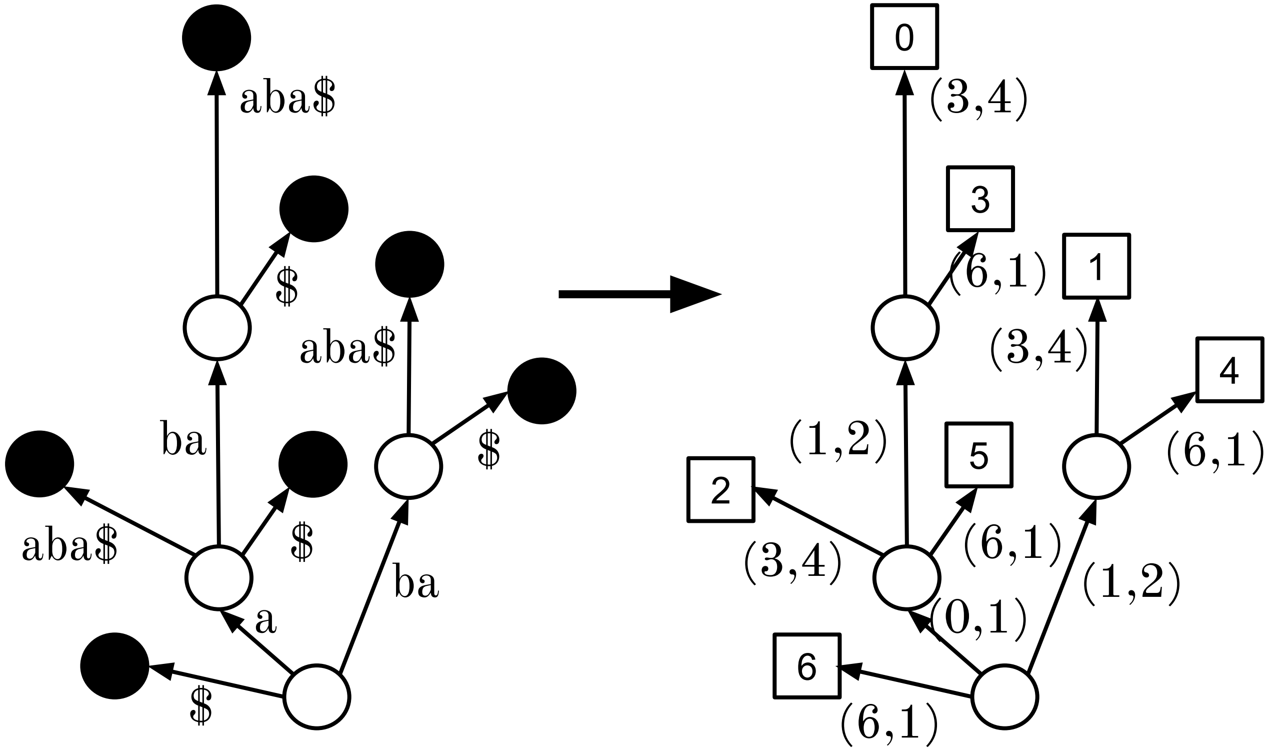
Suffix trie: making it smaller



$ Idea 2: Store T itself in addition to the tree. Convert tree's edge labels to (offset, length) pairs with respect to T

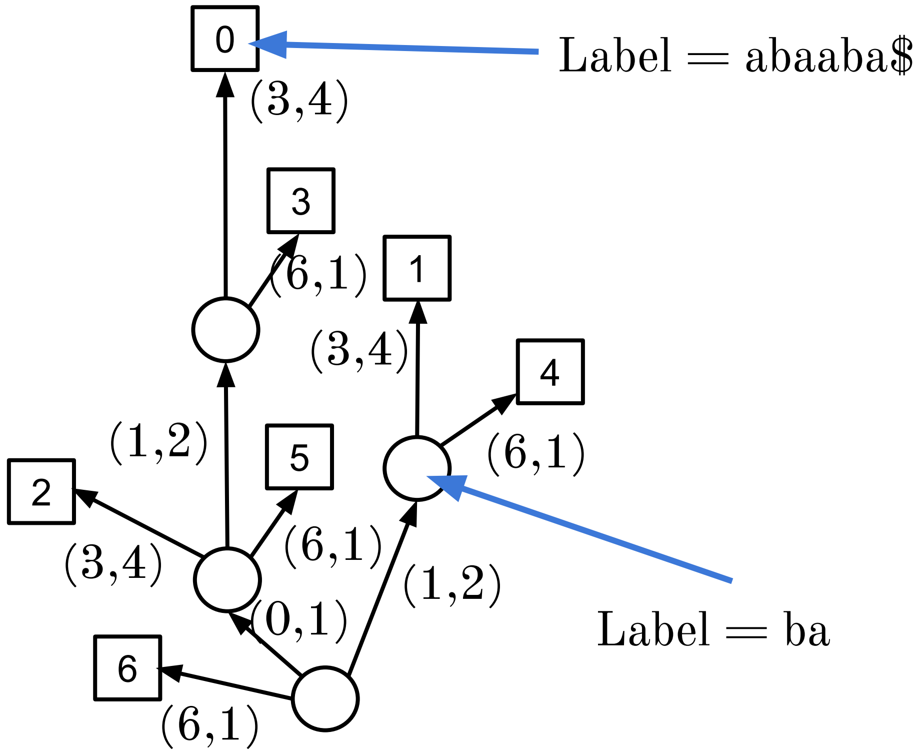
T = abaaba$

Suffix trie: leaves hold offsets of suffixes



T = abaaba$

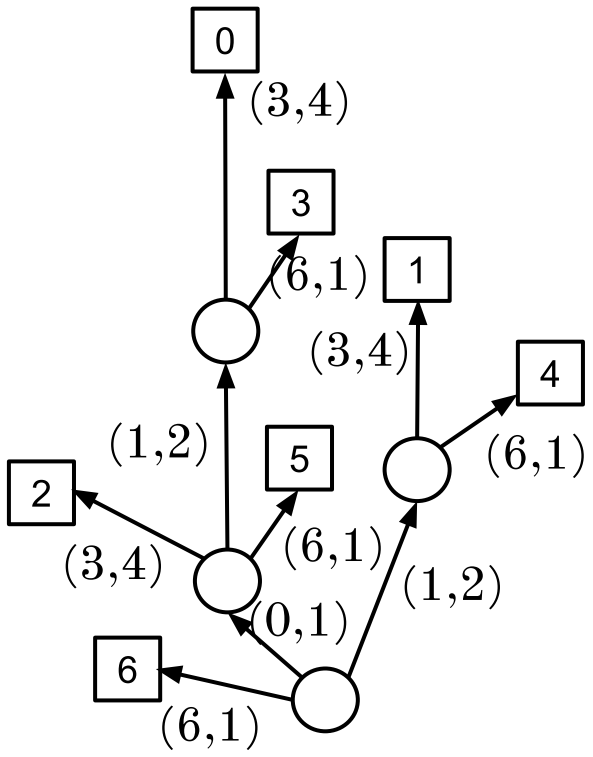
Suffix trie: labels



T = abaaba$

Again, each node's label equals the concatenated edge labels from the root to the node. These aren't stored explicitly

...and we get: Suffix tree



T = abaaba$

Because edges can have string labels, we must distinguish two notions of “depth”

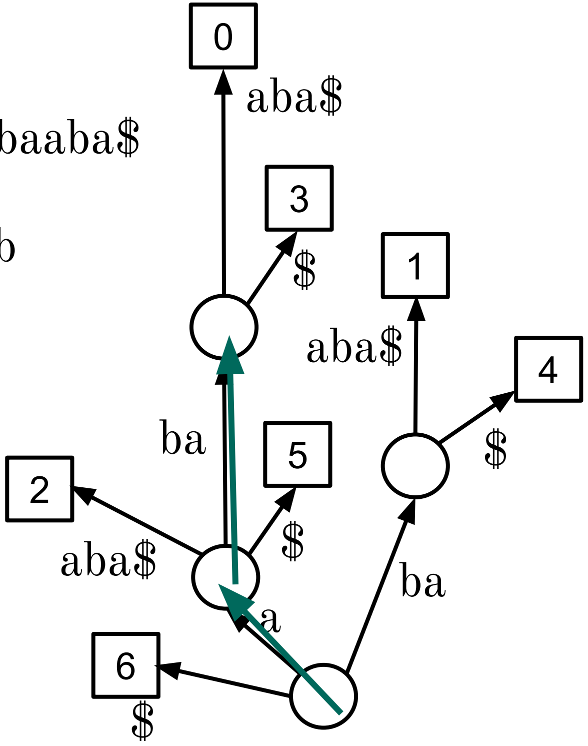
- **Node depth**: how many edges we must follow from the root to reach the node
- **Label depth**: total length of edge labels for edges on path from root to node

Suffix tree

- How do we check whether a string S is a **substring** of T?
- Essentially same procedure as for suffix trie, except we have to deal with coalesced edges

$T = \text{abaaba\$}$

$P = \text{ab}$



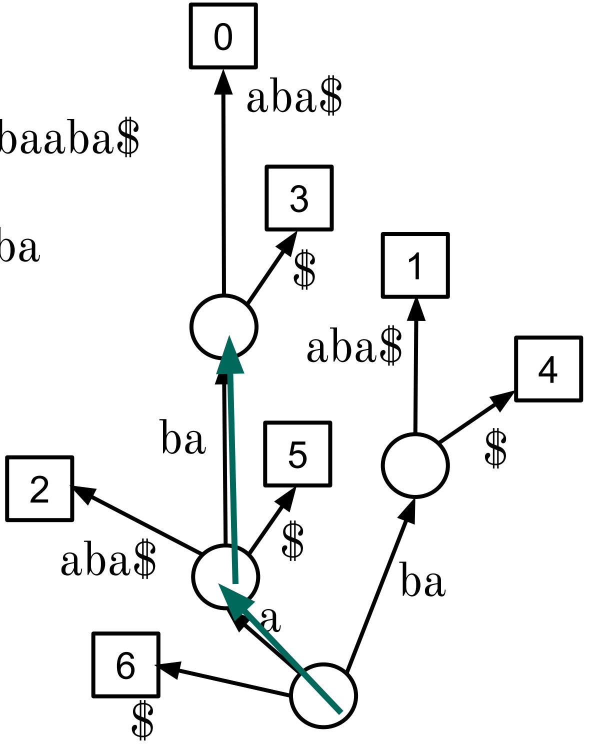
Suffix tree

- How do we check whether a string S is a **suffix** of T?
- Essentially same procedure as for suffix trie, except we have to deal with coalesced edges

```
Fall_off, ended_in_node = climb_the_tree()
if fall_off:
    substring = False
    suffix = false
Else:
    substring = True
    if ended_in_node:
        suffix = True
    Else:
        suffix = False
```

$T = abaaba\$$

$P = aba$

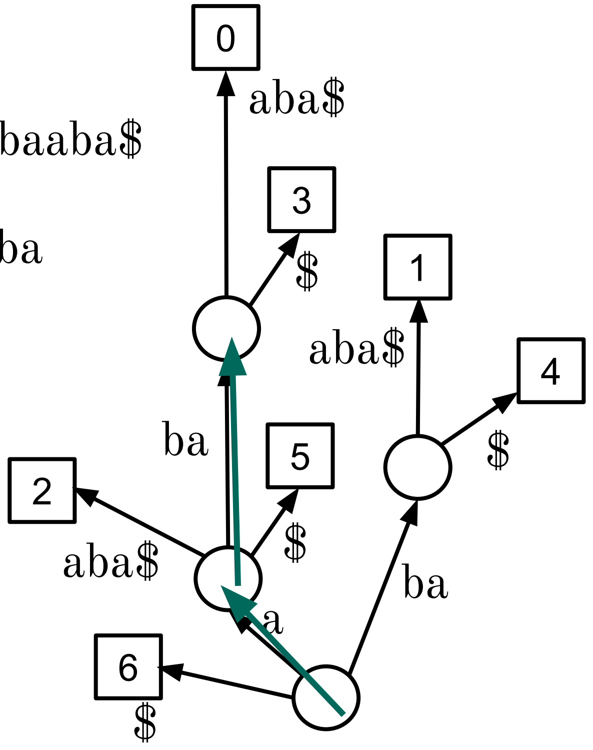


Suffix tree

- How do we count the number of times a string S occurs as a substring of T?
- Same procedure as for suffix trie: Count the number of branches going out of the node in which we ended following the path
- This is common application of suffix tree: Find all matches of P in T

T = abaaba$

P = aba



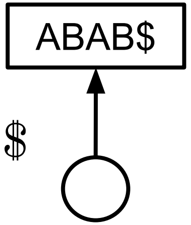
Suffix tree: building

- Naive method 1: build a suffix trie, then coalesce non-branching paths and relabel edges
- Naive method 2: build a single-edge tree representing only the longest suffix, then augment to include the 2nd-longest, then augment to include 3rd-longest, etc
- Both are O(m2) time, but first uses O(m^2) space while second uses O(m)

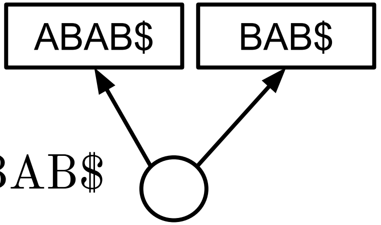
Suffix tree: building Naive method 2

- T = ABAB$

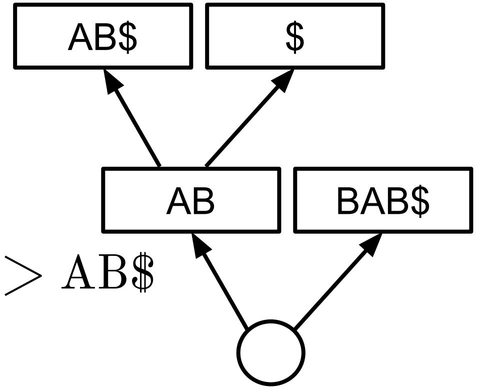
I -> ABAB$



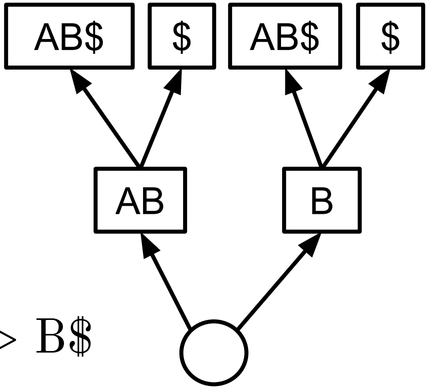
II -> BAB$



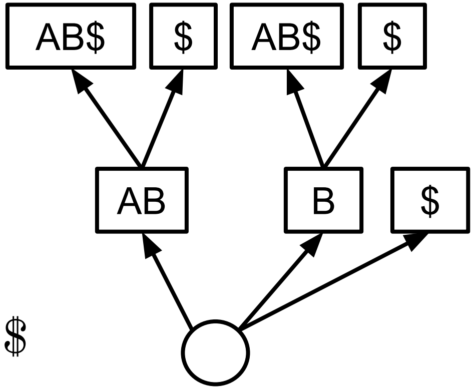
III -> AB$



IV -> B$



V -> $



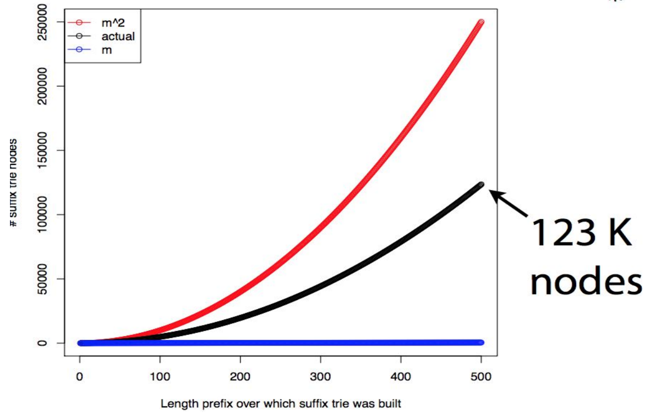
Suffix tree implementation

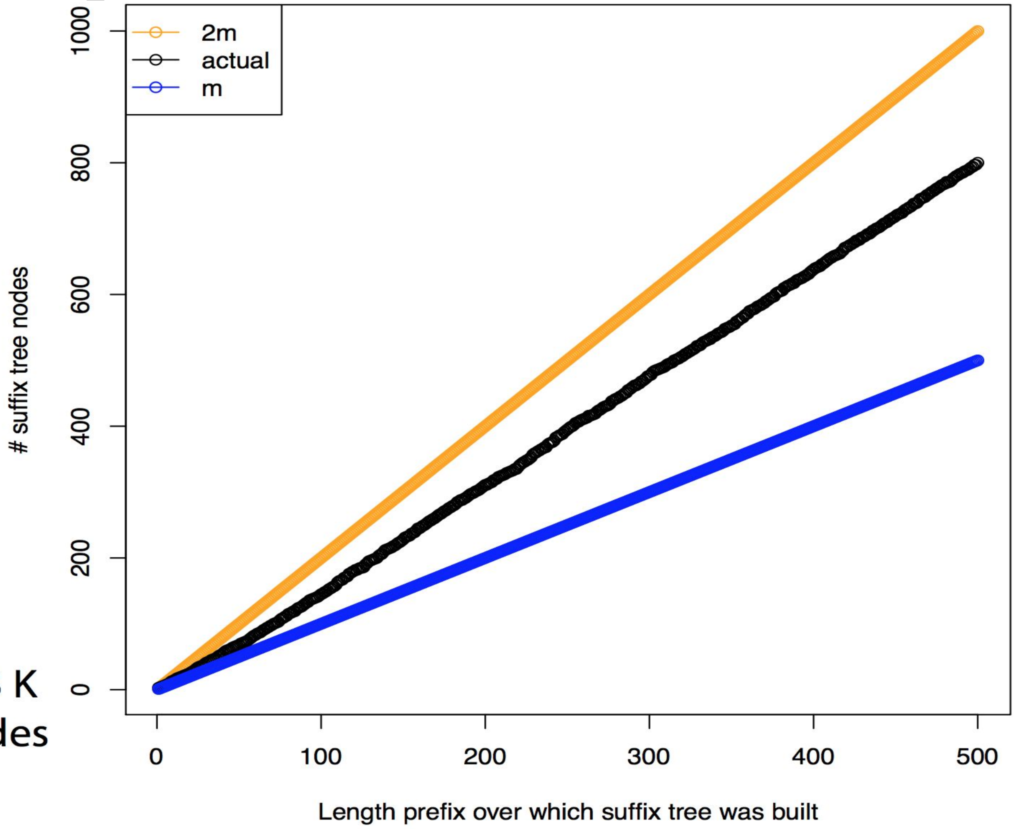
Suffix tree: building - performance

Built suffix trees for the first 500 prefixes of the lambda phage virus genome

Black curve shows # nodes increasing with prefix length

Compare with suffix trie:





Suffix tree: building

- **Ukkonen's** algorithm - O(m) time and space:
  Ukkonen, Esko. "On-line construction of suffix trees."
  Algorithmica 14.3 (1995): 249-260
- Has online property: if T arrives one character at a time, algorithm efficiently updates suffix tree upon each arrival
- We won't cover it here; see Gusfield Ch. 6 for details

Genome sequence alignment requires approximate

Read

```
CTCAAACTCCTGACCTTTGGTGATCCACCCGCCTNGGCCTTC
```

Reference

```
GATCACAGGTCTATCACCCTATTAACCACTCACGGGAGCTCTCCATGCATTTGGTATTTT
CGTCTGGGGGGTATGCACGCGATAGCATTGCGAGACGCTGGAGCCGGAGCACCCTATGTC
GCAGTATCTGTCTTTGATTCCTGCCTCATCCTATTATTTATCGCACCTACGTTCAATATT
ACAGGCGAACATACTTACTAAAGTGTGTTAATTAATTAATGCTTGTAGGACATAATAATA
ACAATTGAATGTCTGCACAGCCACTTTCCACACAGACATCATAACAAAAAATTTCCACCA
AACCCCCCCTCCCCCGCTTCTGGCCACAGCACTTAAACACATCTCTGCCAAACCCCAAAA
ACAAAGAACCCTAACACCAGCCTAACCAGATTTCAAATTTTATCTTTTGGCGGTATGCAC
TTTTAACAGTCACCCCCCAACTAACACATTATTTTCCCCTCCCACTCCCATACTACTAAT
CTCATCAATACAACCCCCGCCCATCCTACCCAGCACACACACACCGCTGCTAACCCCATA
CCCCGAACCAACCAAACCCCAAAGACACCCCCCACAGTTTATGTAGCTTACCTCCTCAAA
GCAATACACTGACCCGCTCAAACTCCTGGATTTTGGATCCACCCAGCGCCTTGGCCTAAA
CTAGCCTTTCTATTAGCTCTTAGTAAGATTACACATGCAAGCATCCCCGTTCCAGTGAGT
TCACCCTCTAAATCACCACGATCAAAAGGAACAAGCATCAAGCACGCAGCAATGCAGCTC
AAAACGCTTAGCCTAGCCACACCCCCACGGGAAACAGCAGTGATTAACCTTTAGCAATAA
ACGAAAGTTTAACTAAGCTATACTAACCCCAGGGTTGGTCAATTTCGTGCCAGCCACCGC
GGTCACACGATTAACCCAAGTCAATAGAAGCCGGCGTAAAGAGTGTTTTAGATCACCCCC
TCCCCAATAAAGCTAAAACTCACCTGAGTTGTAAAAAACTCCAGTTGACACAAAATAGAC
TACGAAAGTGGCTTTAACATATCTGAACACACAATAGCTAAGACCCAAACTGGGATTAGA
TACCCCACTATGCTTAGCCCTAAACCTCAACAGTTAAATCAACAAAACTGCTCGCCAGAA
CACTACGAGCCACAGCTTAAAACTCAAAGGACCTGGCGGTGCTTCATATCCCTCTAGAGG
AGCCTGTTCTGTAATCGATAAACCCCGATCAACCTCACCACCTCTTGCTCAGCCTATATA
CCGCCATCTTCAGCAAACCCTGATGAAGGCTACAAAGTAAGCGCAAGTACCCACGTAAAG
ACGTTAGGTCAAGGTGTAGCCCATGAGGTGGCAAGAAATGGGCTACATTTTCTACCCCAG
AAAACTACGATAGCCCTTATGAAACTTAAGGGTCGAAGGTGGATTTAGCAGTAAACTAAG
AGTAGAGTGCTTAGTTGAACAGGGCCCTGAAGCGCGTACACACCGCCCGTCACCCTCCTC
AAGTATACTTCAAAGGACATTTAACTAAAACCCCTACGCATTTATATAGAGGAGACAAGT
CGTAACCTCAAACTCCTGCCTTTGGTGATCCACCCGCCTTGGCCTACCTGCATAATGAAG
AAGCACCCAACTTACACTTAGGAGATTTCAACTTAACTTGACCGCTCTGAGCTAAACCTA
GCCCCAAACCCACTCCACCTTACTACCAGACAACCTTAGCCAAACCATTTACCCAAATAA
AGTATAGGCGATAGAAATTGAAACCTGGCGCAATAGATATAGTACCGCAAGGGAAAGATG
GGCCTACTTCACAAAGCGCCTTCCCCCGTAAATGA
```

Sequence differences occur because of...

1. Sequencing errors
2. Natural variation

... 3 billion nucleotides long reference genome...
...and we have to deal with repetitive sequences

Approximate string matching

Looking for places where a P matches T with up to a certain number of mismatches or edits. Each such place is an approximate match.

A mismatch is a single-character substitution (variation) - SNV:

```
T: GGAAAAAGAGGTAGCGGCGTTTAACAGTAG
P:           GTAACGGCG
```

An edit is a single-character substitution or gap (**insertion** or **deletion**):

```
T: GGAAAAAGAGGTAGC-GCGTTTAACAGTAG  (insertion)
P:           GTAGCGGCG
```

```
T: GGAAAAAGAGGTAGCGGCGTTTAACAGTAG (deletion)
P:           GTAGC-GCG
```

Hamming and edit distance

For two same-length strings X and Y, **hamming** distance is the minimum number of single-character substitutions needed to turn one into the other:

X: GAGGTAGCGGCGTTTAAC
Y: GTGGTAACGGGGTTTAAC

Hamming distance = 3

Edit distance (Levenshtein distance): minimum number of edits required to turn one into the other:

X: TGGCCGCGCAAAAACAGC
Y: TGACCGCGCAAAA- CAGC

Edit distance = 2

What would be the Hamming distance here?

Approximate matching

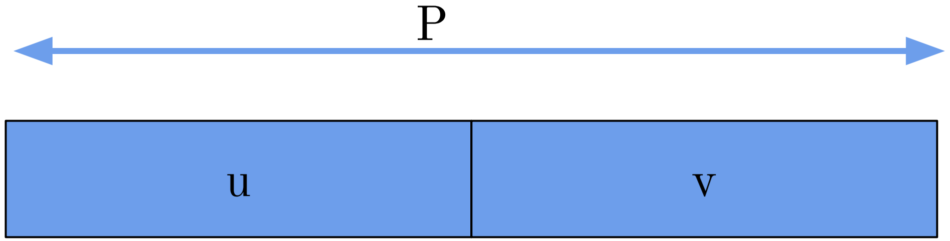
Adapting the naive algorithm with configurable Hamming distance:

```
def naiveApproximate(p, t, maxHammingDistance=1):
    occurrences = []
    for i in range(len(t) - len(p) + 1):     # for all alignments
        nmm = 0
        for j in range(len(p)):              # for all characters
            if t[i+j] != p[j]:               # does it match?
                nmm += 1                     # mismatch
                if nmm > maxHammingDistance:
                    break                    # exceeded maximum distance
        if nmm <= maxHammingDistance:
            # approximate match; return pair where first element is the
            # offset of the match and second is the Hamming distance
            occurrences.append((i, nmm))
    return occurrences
```

Instead of stopping upon first mismatch, stop when maximum distance is exceeded

Approximate matching

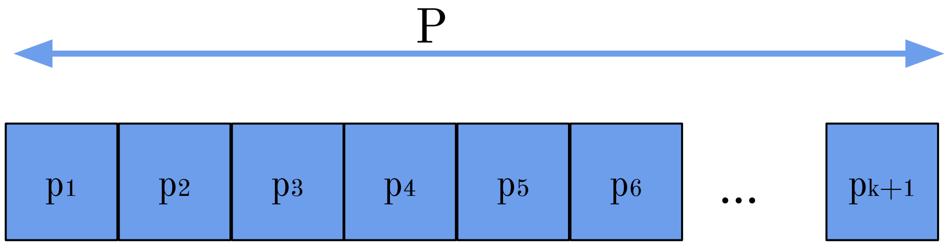
Mission: apply exact matching algorithms to approximate matching problems!



If P occurs in T with 1 edit, then u or v appears with no edits
(u and v are two non-overlapping substrings of P)

Approximate matching

Mission: apply exact matching algorithms to approximate matching problems!

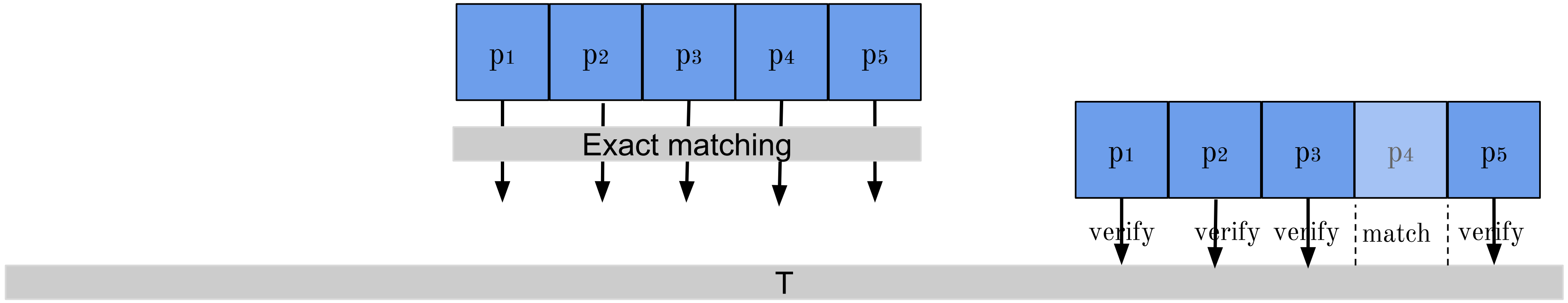


If P occurs in T with up to k edits, at least one of p1, p2, ..., pk+1 must appear with 0 edits

Pigeonhole principle

If n items are put into m containers, with n $>$ m, then at least one container must contain more than one item (Dirichlet's principle).

Pigeonhole principle


p1
p2
p3
p4
p5
Exact matching
p1
p2
p3
p4
p5
verify
verify
verify
match
verify
T