

Suffix trees

Ben Langmead



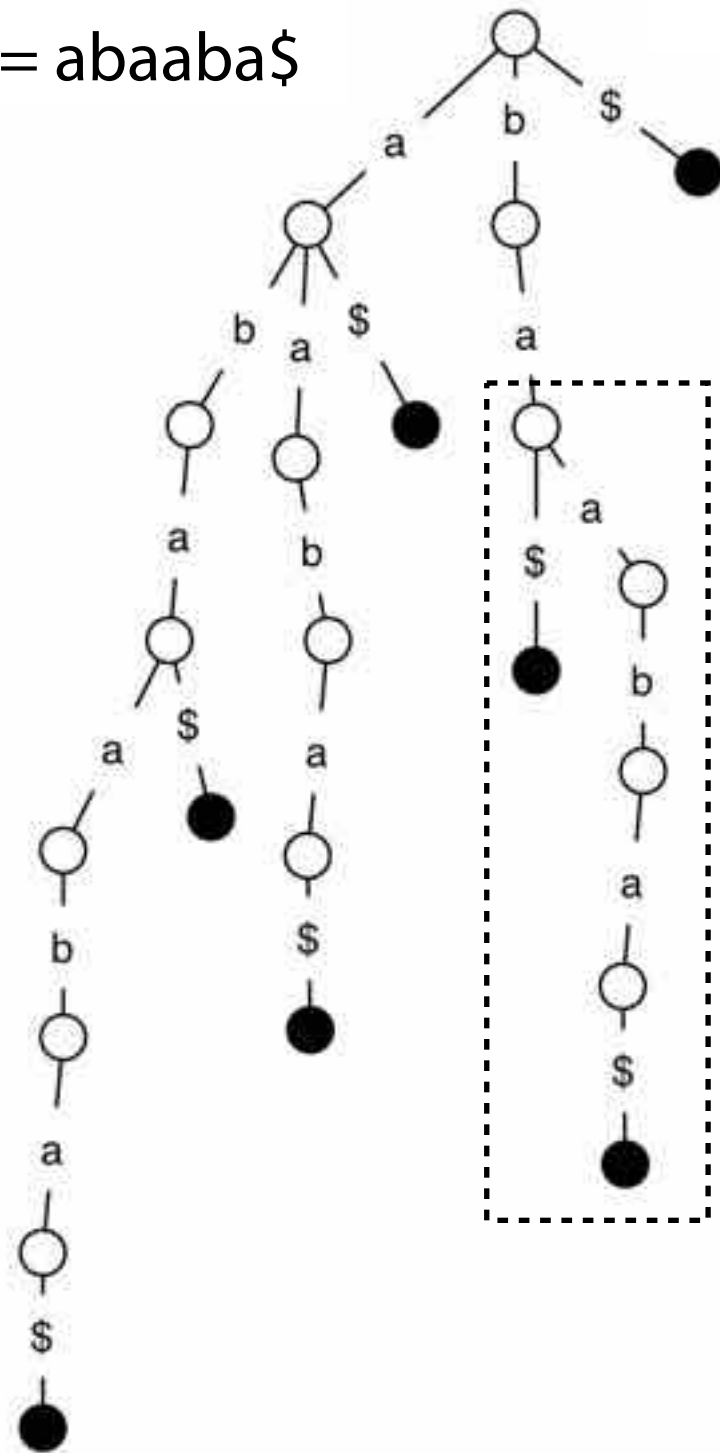
JOHNS HOPKINS

WHITING SCHOOL
of ENGINEERING

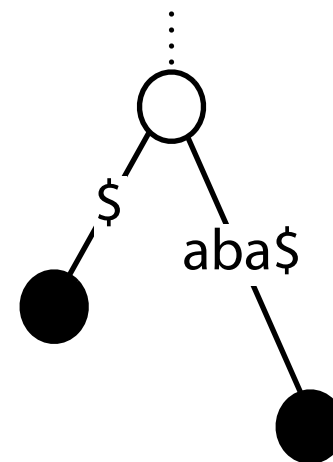
You are free to use these slides. If you do, please sign the guestbook (www.langmead-lab.org/teaching-materials), or email me (ben.langmead@gmail.com) and tell me briefly how you're using them. For original Keynote files, email me.

Suffix trie: making it smaller

$T = \text{abaaba\$}$



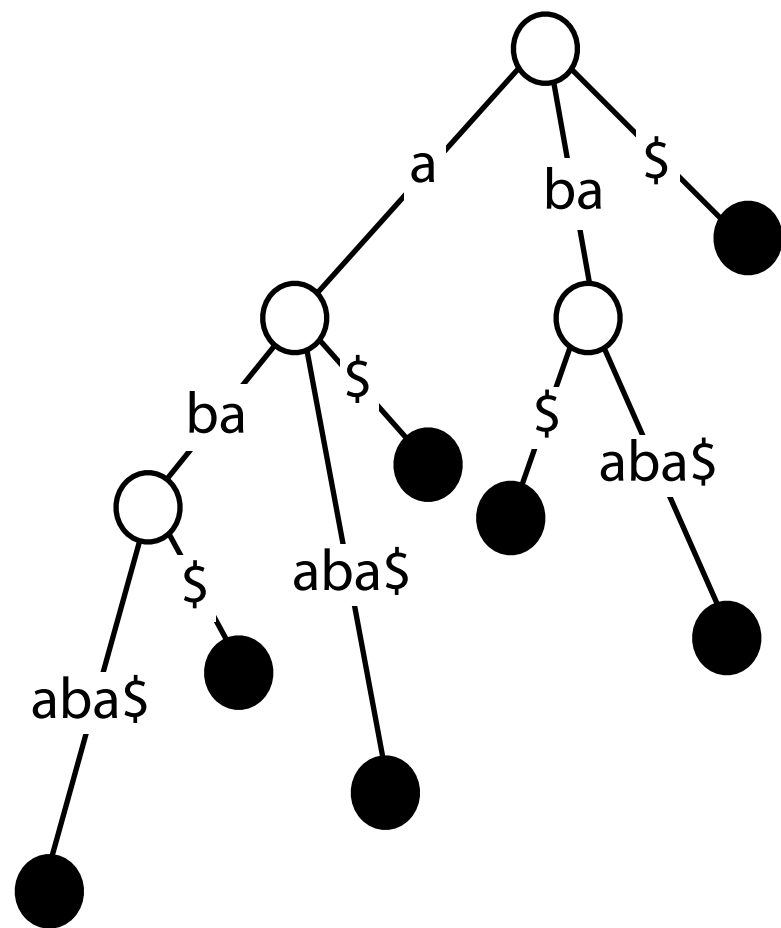
Idea 1: Coalesce non-branching paths into a *single edge* with a *string* label



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

Suffix tree

$T = \text{abaaba}\$$



With respect to m :

How many leaves?

m

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

 $\leq m - 1$

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

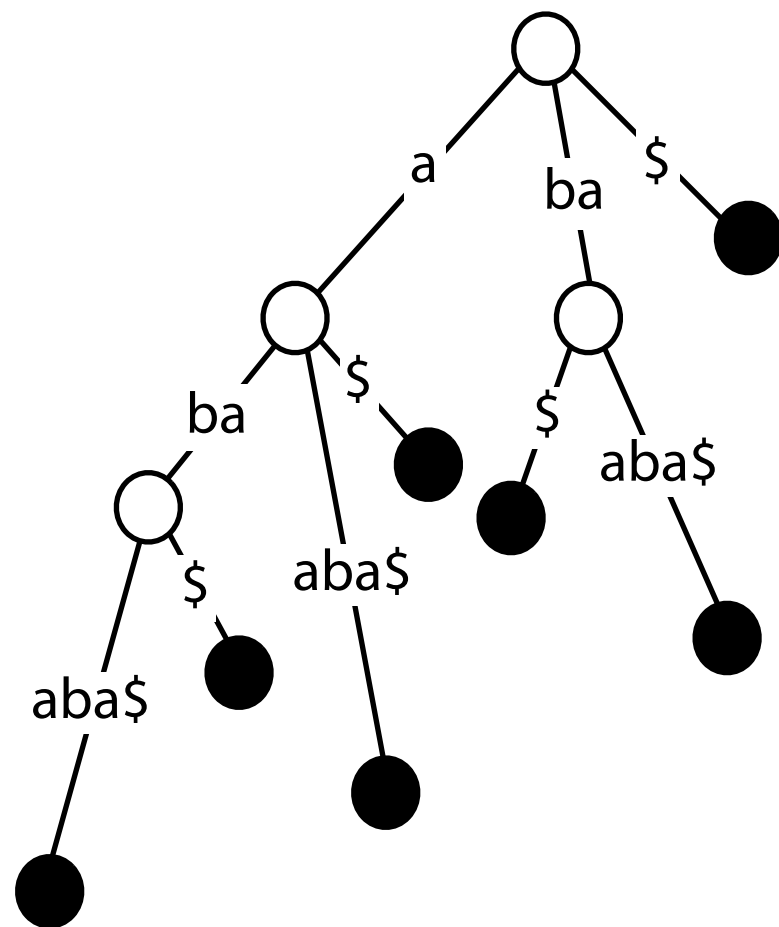
Is the total size $O(m)$ now?

No: total length of edge labels is quadratic in m

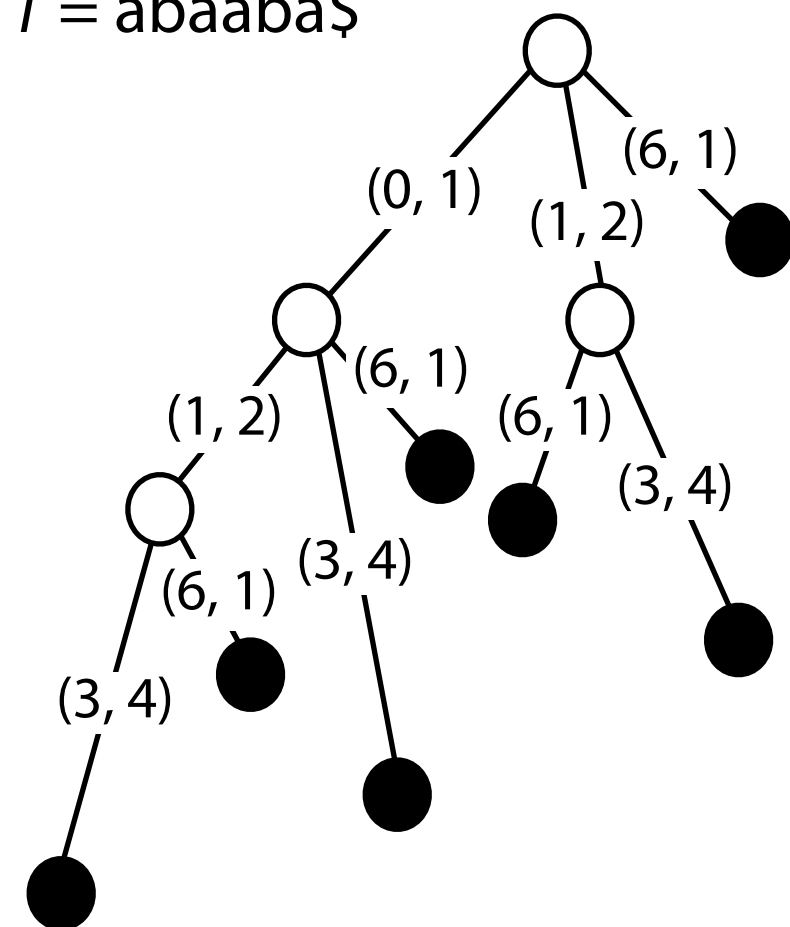
Suffix tree

$T = \text{abaaba}\$$

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

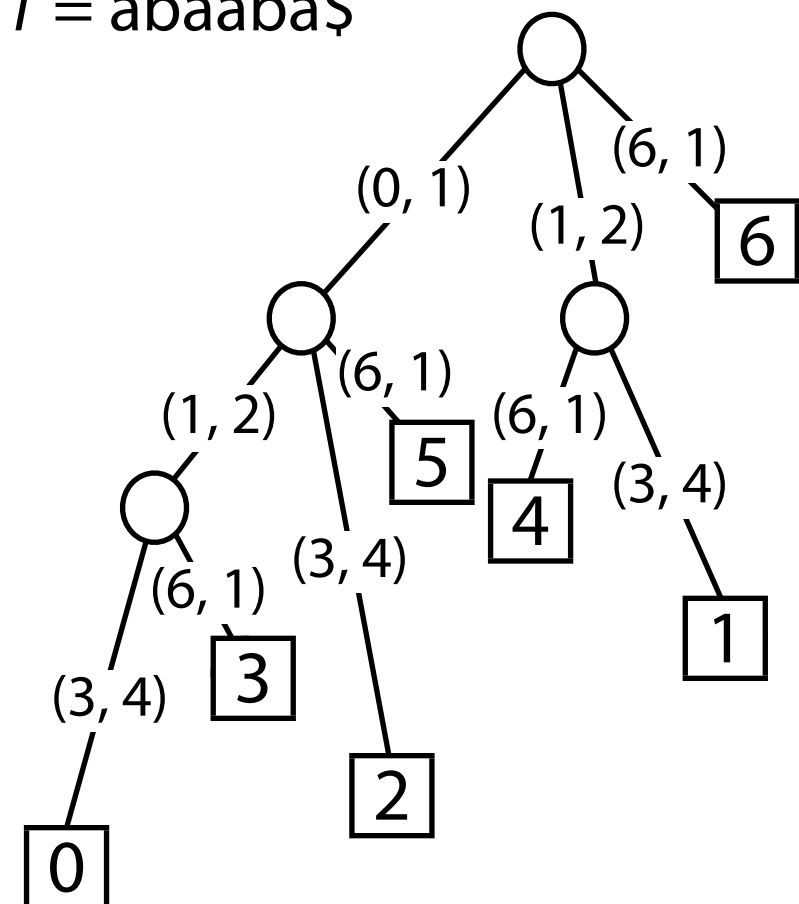
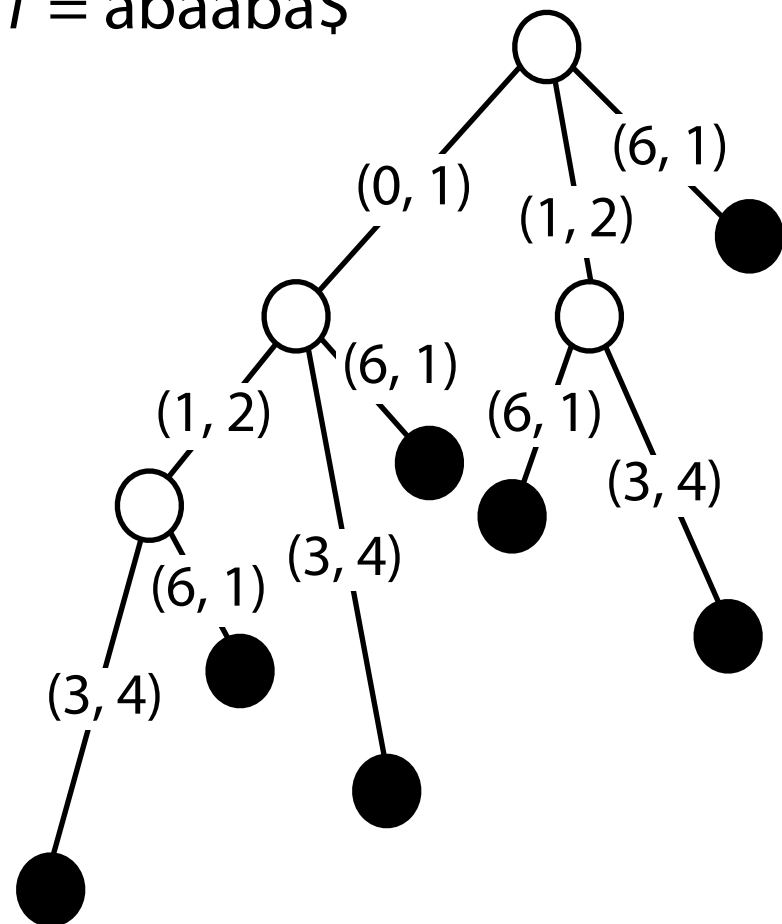


$T = \text{abaaba}\$$



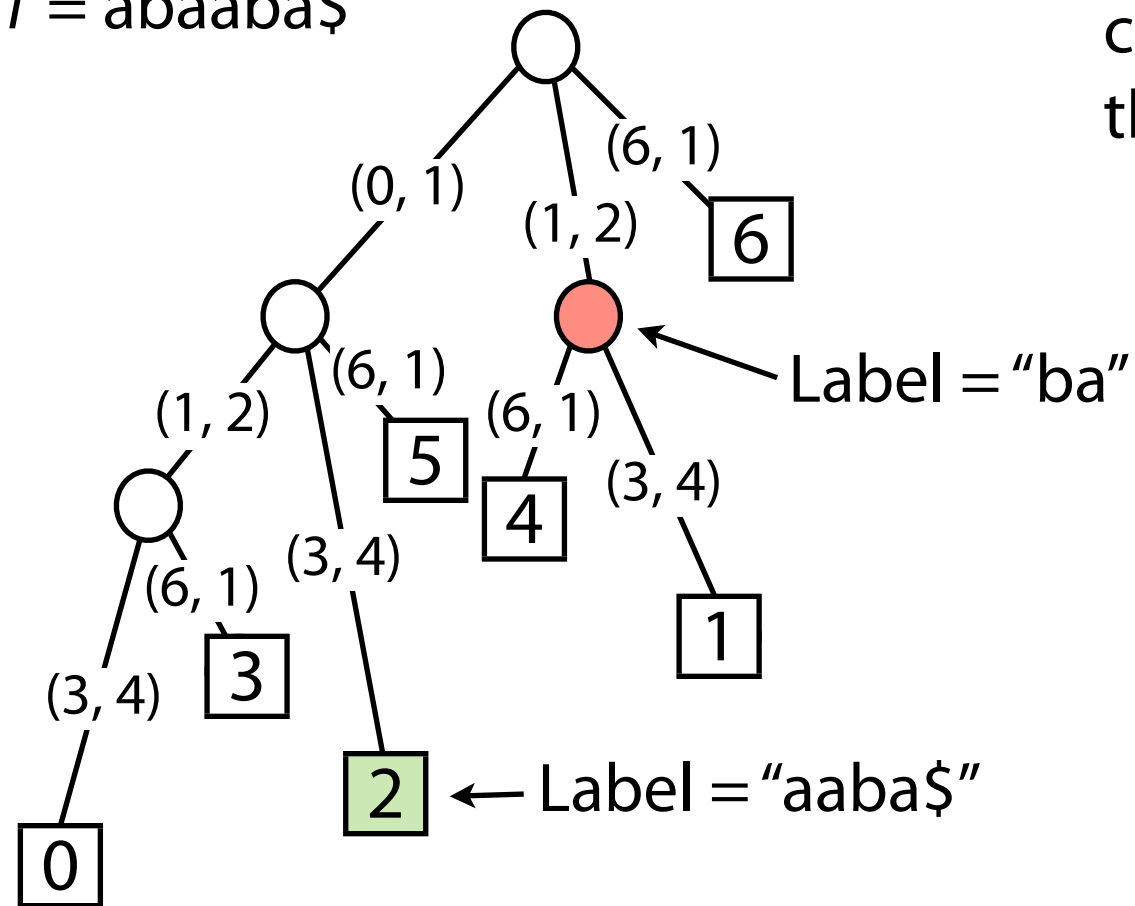
Space required for suffix tree is now $O(m)$

Suffix tree: leaves hold offsets



Suffix tree: labels

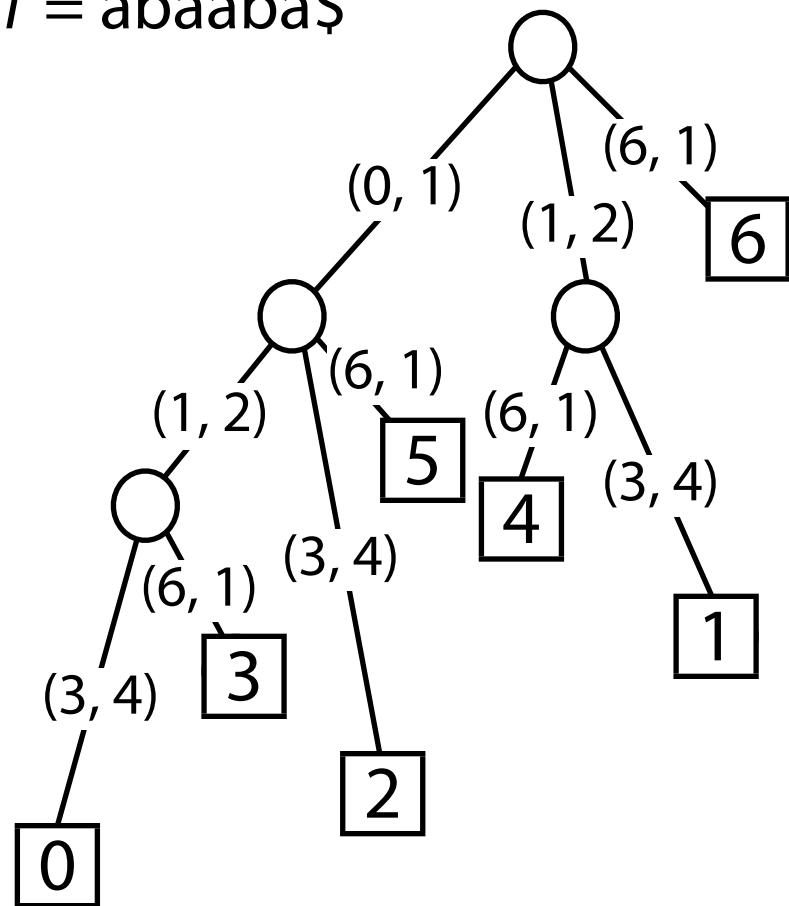
$T = \text{abaaba}\$$



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

Suffix tree: labels

$T = \text{abaaba}\$$

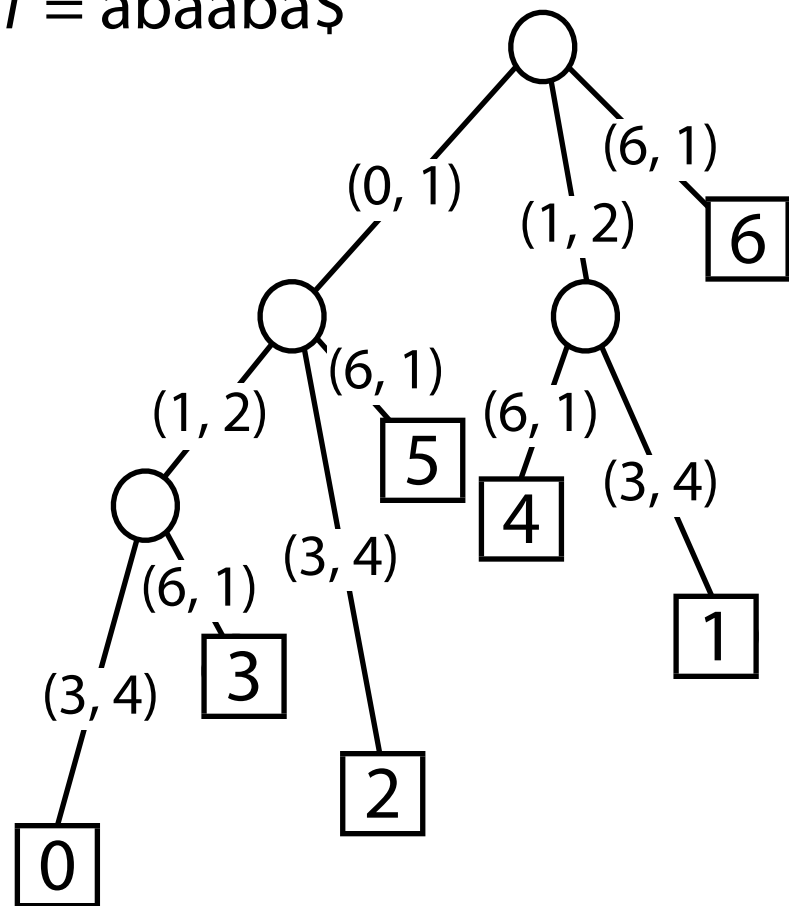


Because edges can have string labels, we must distinguish two notions of “depth”

- **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: space caveat

$T = \text{abaaba}\$$



Minor point:

We say the space taken by the edge labels is $O(m)$, because we keep 2 integers per edge and there are $O(m)$ edges

To store one such integer, we need enough bits to distinguish m positions in T , i.e. $\text{ceil}(\log_2 m)$ bits. We usually ignore this factor, since 64 bits is plenty for all practical purposes.

Similar argument for the pointers / references used to distinguish tree nodes.

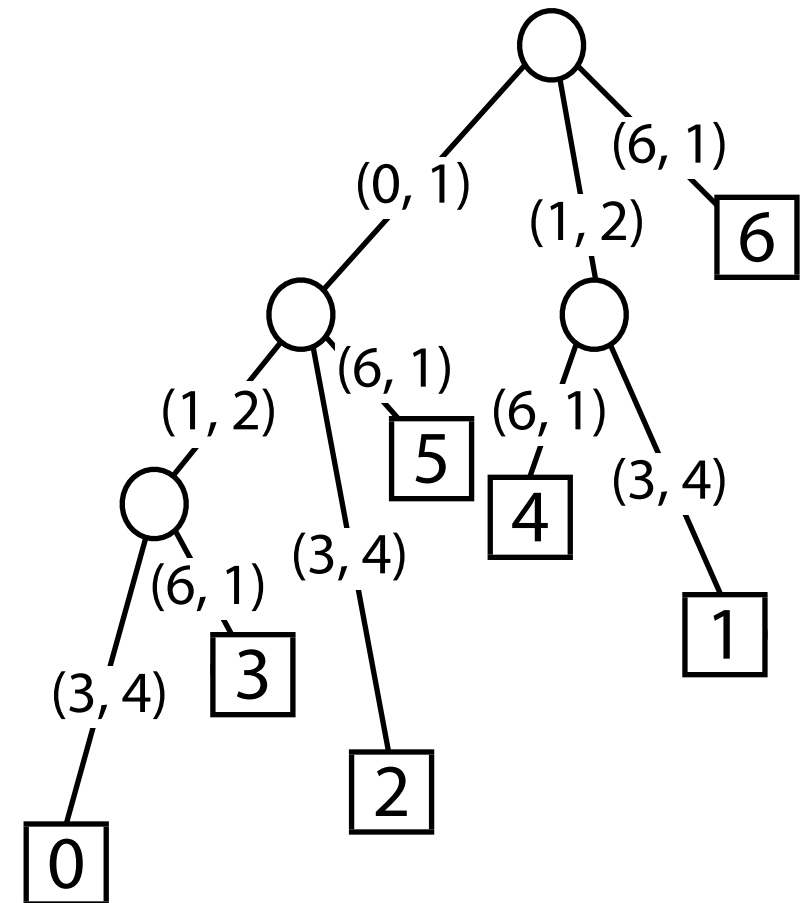
Suffix tree: building

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 $O(m^2)$ time, but first uses $O(m^2)$ space while second uses $O(m)$

Naive method 2 is described in Gusfield 5.4



Suffix tree: implementation

```
class SuffixTree(object):

    class Node(object):
        def __init__(self, lab):
            self.lab = lab # label on path leading to this node
            self.out = {} # outgoing edges; maps characters to nodes

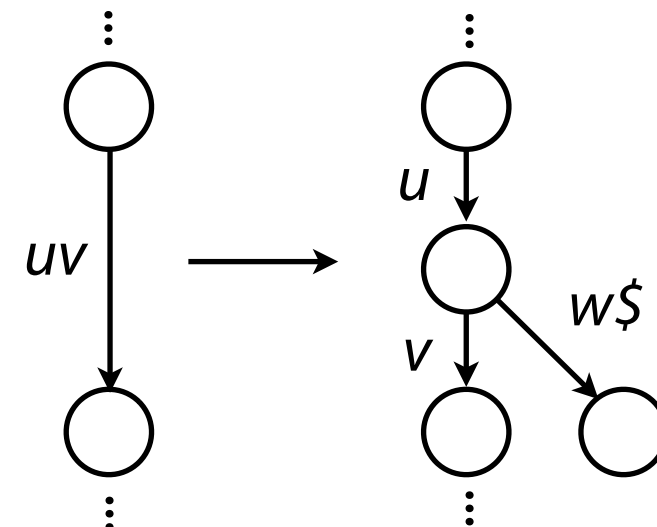
    def __init__(self, s):
        """ Make suffix tree, without suffix links, from s in quadratic time
            and linear space """
        s += '$'
        self.root = self.Node(None)
        self.root.out[s[0]] = self.Node(s) # trie for just longest suf
        # add the rest of the suffixes, from longest to shortest
        for i in xrange(1, len(s)):
            # start at root; we'll walk down as far as we can go
            cur = self.root
            j = i
            while j < len(s):
                if s[j] in cur.out:
                    child = cur.out[s[j]]
                    lab = child.lab
                    # Walk along edge until we exhaust edge label or
                    # until we mismatch
                    k = j+1
                    while k-j < len(lab) and s[k] == lab[k-j]:
                        k += 1
                    if k-j == len(lab):
                        cur = child # we exhausted the edge
                        j = k
                    else:
                        # we fell off in middle of edge
                        cExist, cNew = lab[k-j], s[k]
                        # create "mid": new node bisecting edge
                        mid = self.Node(lab[:k-j])
                        mid.out[cNew] = self.Node(s[k:])
                        # original child becomes mid's child
                        mid.out[cExist] = child
                        # original child's label is curtailed
                        child.lab = lab[k-j:]
                        # mid becomes new child of original parent
                        cur.out[s[j]] = mid
                else:
                    # Fell off tree at a node: make new edge hanging off it
                    cur.out[s[j]] = self.Node(s[j:])
```

$O(m^2)$ time, $O(m)$ space

Make 2-node tree for longest suffix

Add rest of suffixes from
long to short, adding 1
or 2 nodes for each

Most complex case:



Suffix tree: implementation

(still in class `SuffixTree`)

```
def followPath(self, s):
    """ Follow path given by s. If we fall off tree, return None. If we
        finish mid-edge, return (node, offset) where 'node' is child and
        'offset' is label offset. If we finish on a node, return (node,
        None). """
    cur = self.root
    i = 0
    while i < len(s):
        c = s[i]
        if c not in cur.out:
            return (None, None) # fell off at a node
        child = cur.out[s[i]]
        lab = child.lab
        j = i+1
        while j-i < len(lab) and j < len(s) and s[j] == lab[j-i]:
            j += 1
        if j-i == len(lab):
            cur = child # exhausted edge
            i = j
        elif j == len(s):
            return (child, j-i) # exhausted query string in middle of edge
        else:
            return (None, None) # fell off in the middle of the edge
    return (cur, None) # exhausted query string at internal node

def hasSubstring(self, s):
    """ Return true iff s appears as a substring """
    node, off = self.followPath(s)
    return node is not None

def hasSuffix(self, s):
    """ Return true iff s is a suffix """
    node, off = self.followPath(s)
    if node is None:
        return False # fell off the tree
    if off is None:
        # finished on top of a node
        return '$' in node.out
    else:
        # finished at offset 'off' within an edge leading to 'node'
        return node.lab[off] == '$'
```

followPath: Given a string, walk down corresponding path. Return a special value if we fall off, or a description of where we end up otherwise.

Has substring? Return true iff **followPath** didn't fall off.

Has suffix? Return true iff **followPath** didn't fall off and we ended just above a "\$".

Suffix tree: implementation

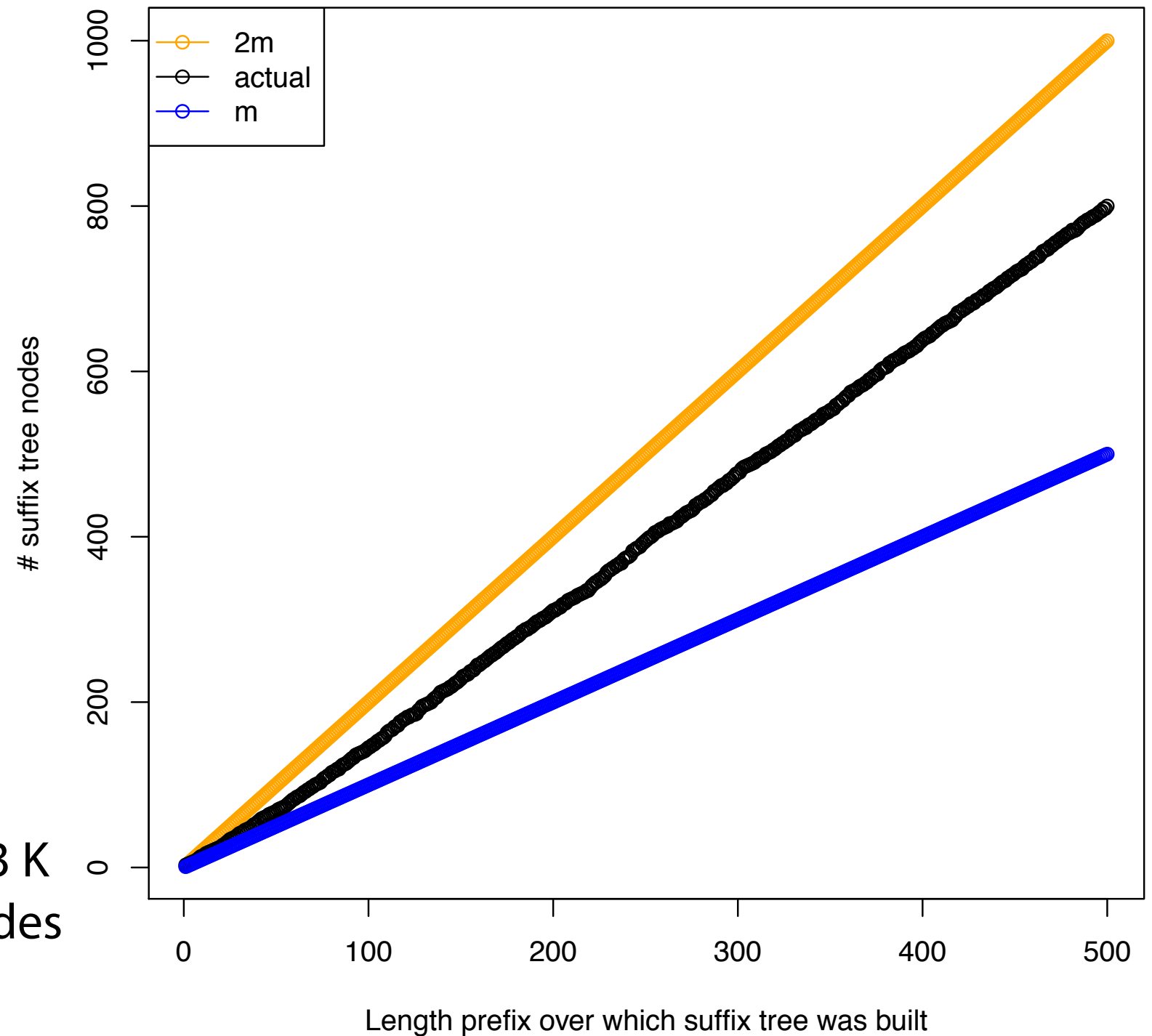
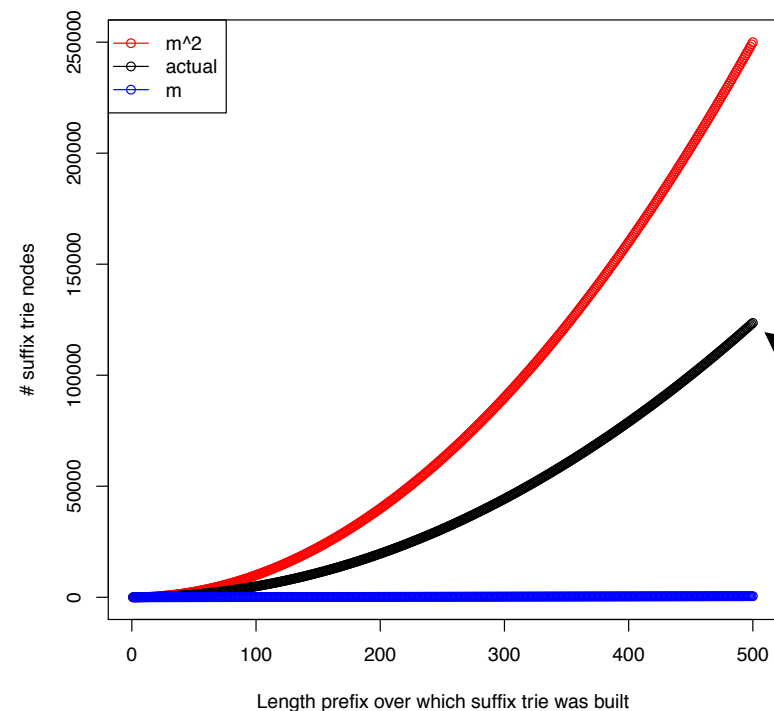
Python example here: <http://nbviewer.ipython.org/6665861>

Suffix tree: actual growth

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

Method of choice: Ukkonen's algorithm

Ukkonen, Esko. "On-line construction of suffix trees."
Algorithmica 14.3 (1995): 249-260.

$O(m)$ time and space

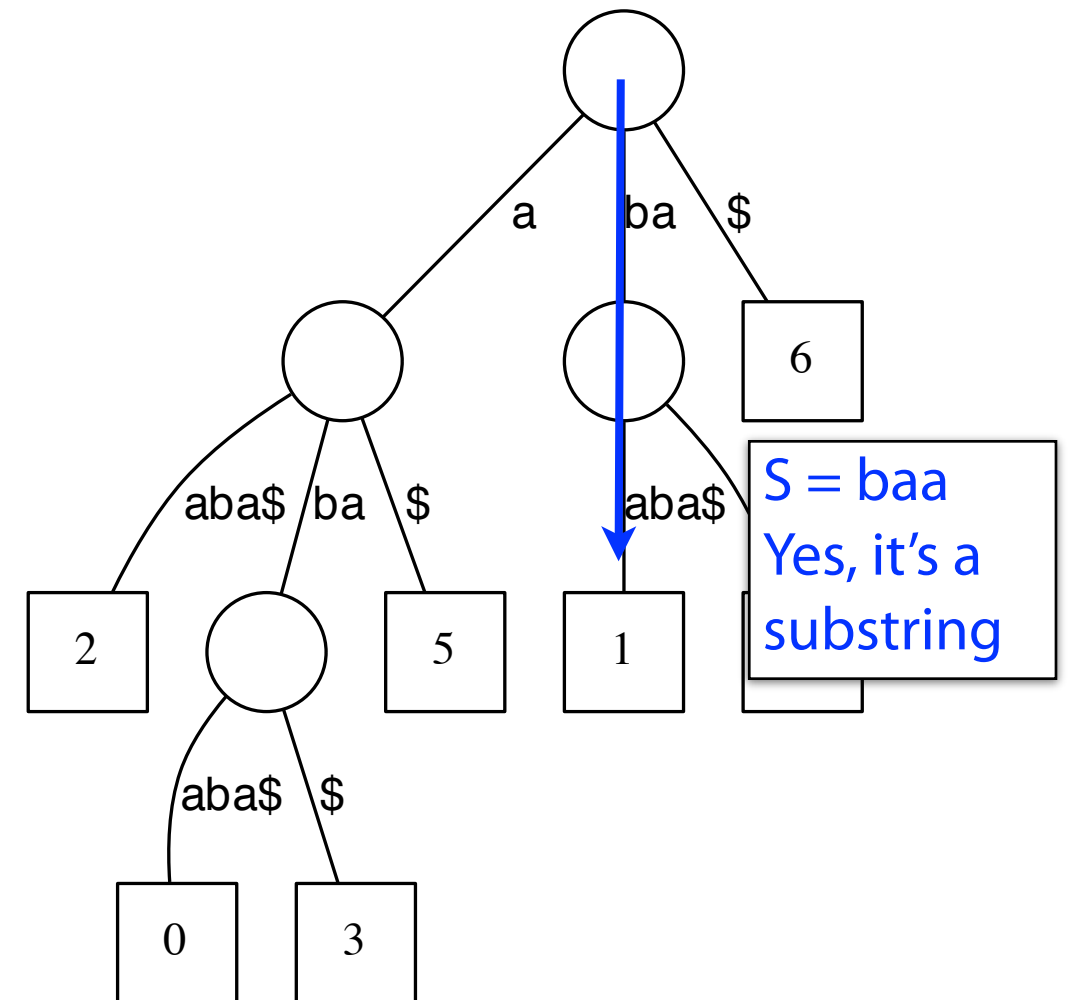
Has *online* property: if T arrives one character at a time, algorithm efficiently updates suffix tree upon each arrival

We won't cover it here; see Gusfield Ch. 6 for details

Suffix tree

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

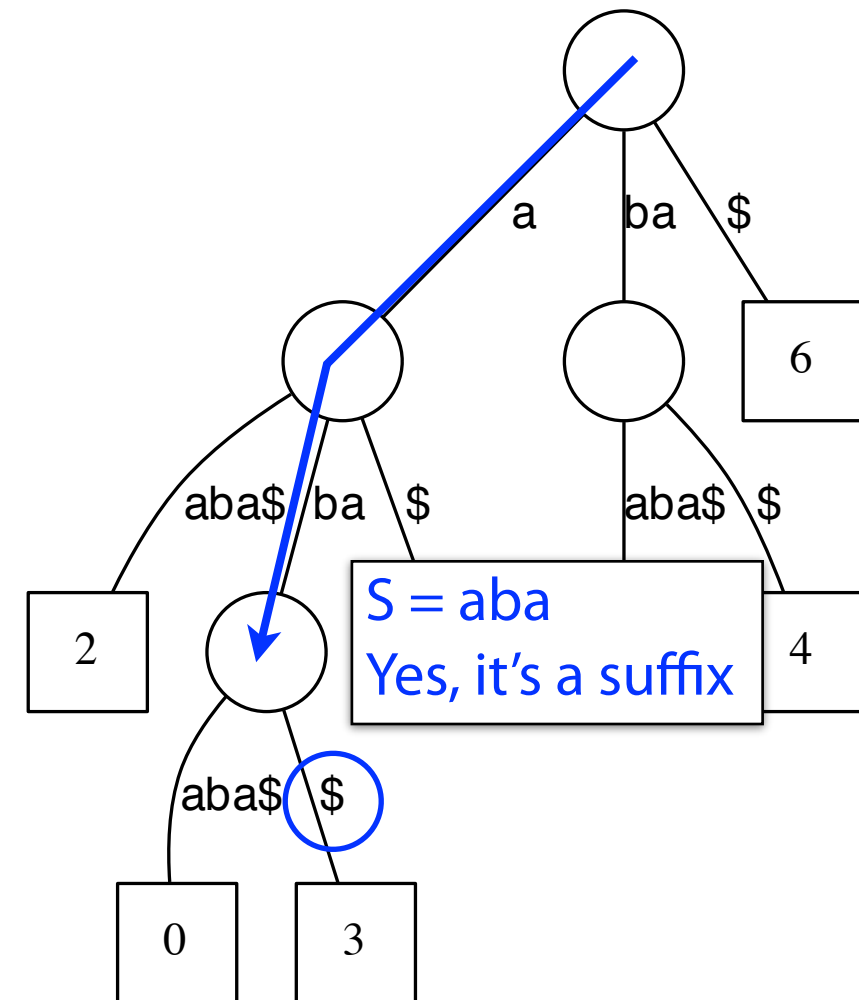
Essentially same procedure as for suffix trie, except we have to deal with coalesced edges



Suffix tree

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

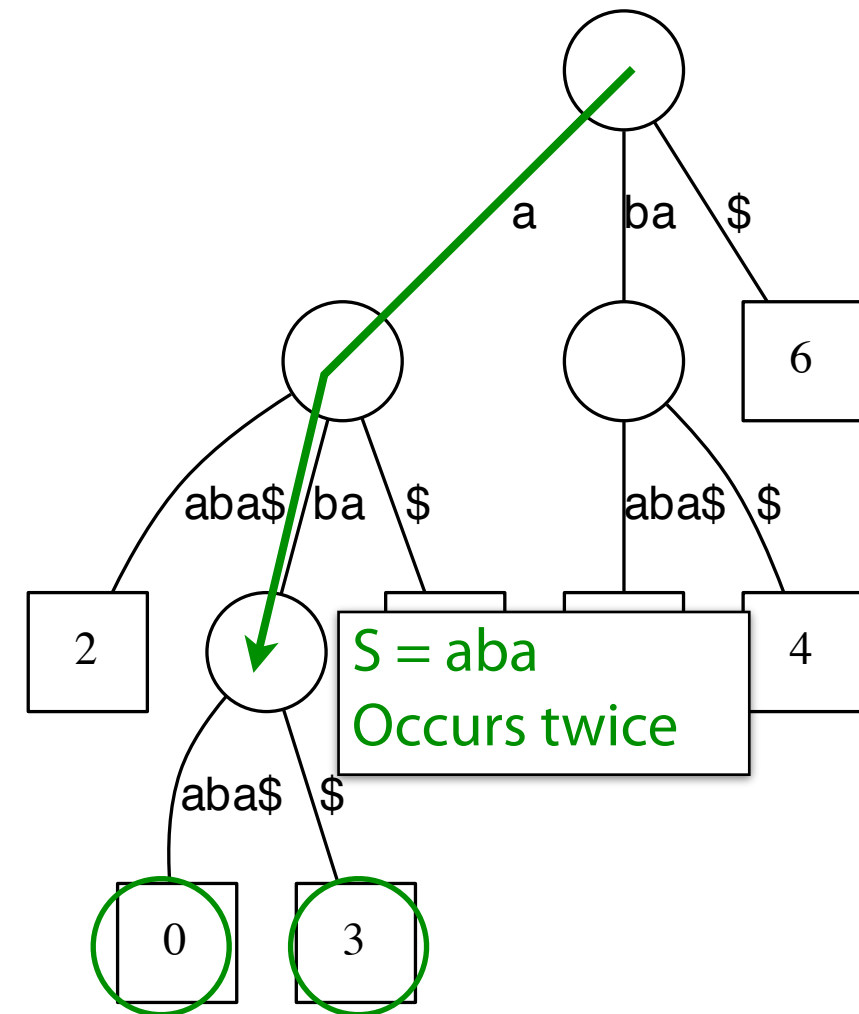
Essentially same procedure as for suffix trie, except we have to deal with coalesced edges



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



Suffix tree: applications

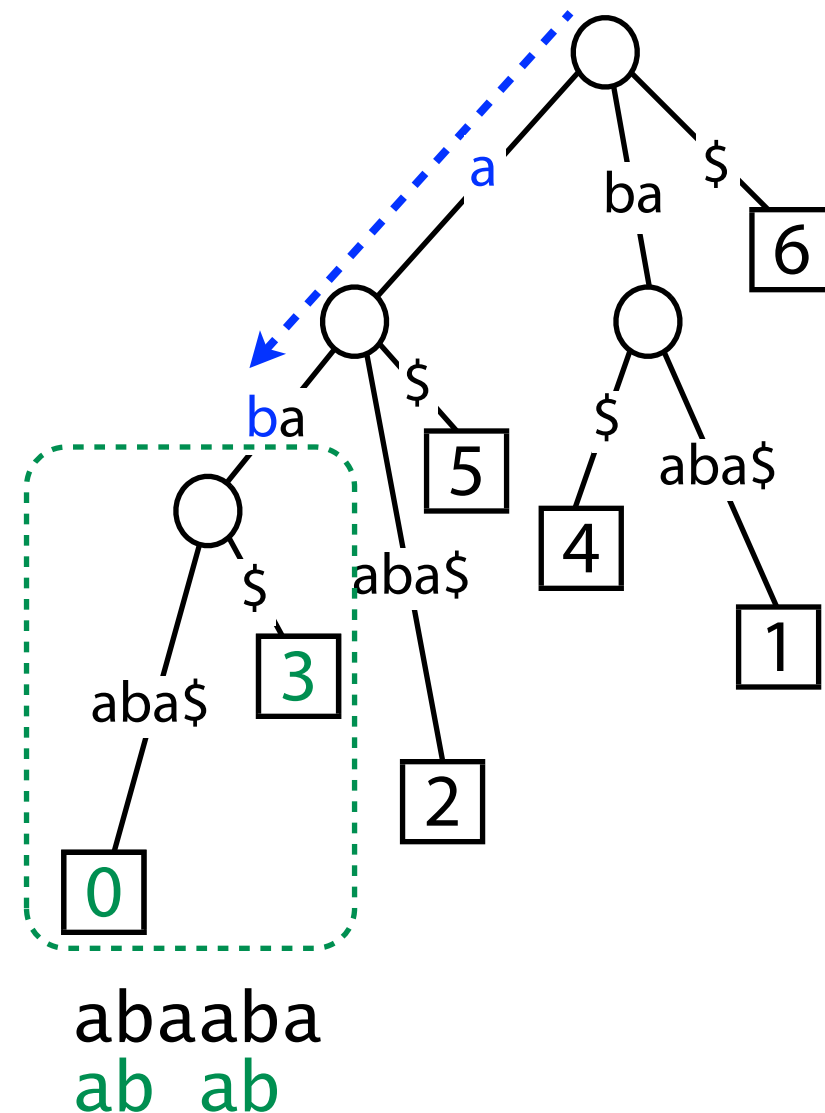
With suffix tree of T , we can find all matches of P to T . Let $k = \#$ matches.

E.g., $P = ab$, $T = abaaba\$$

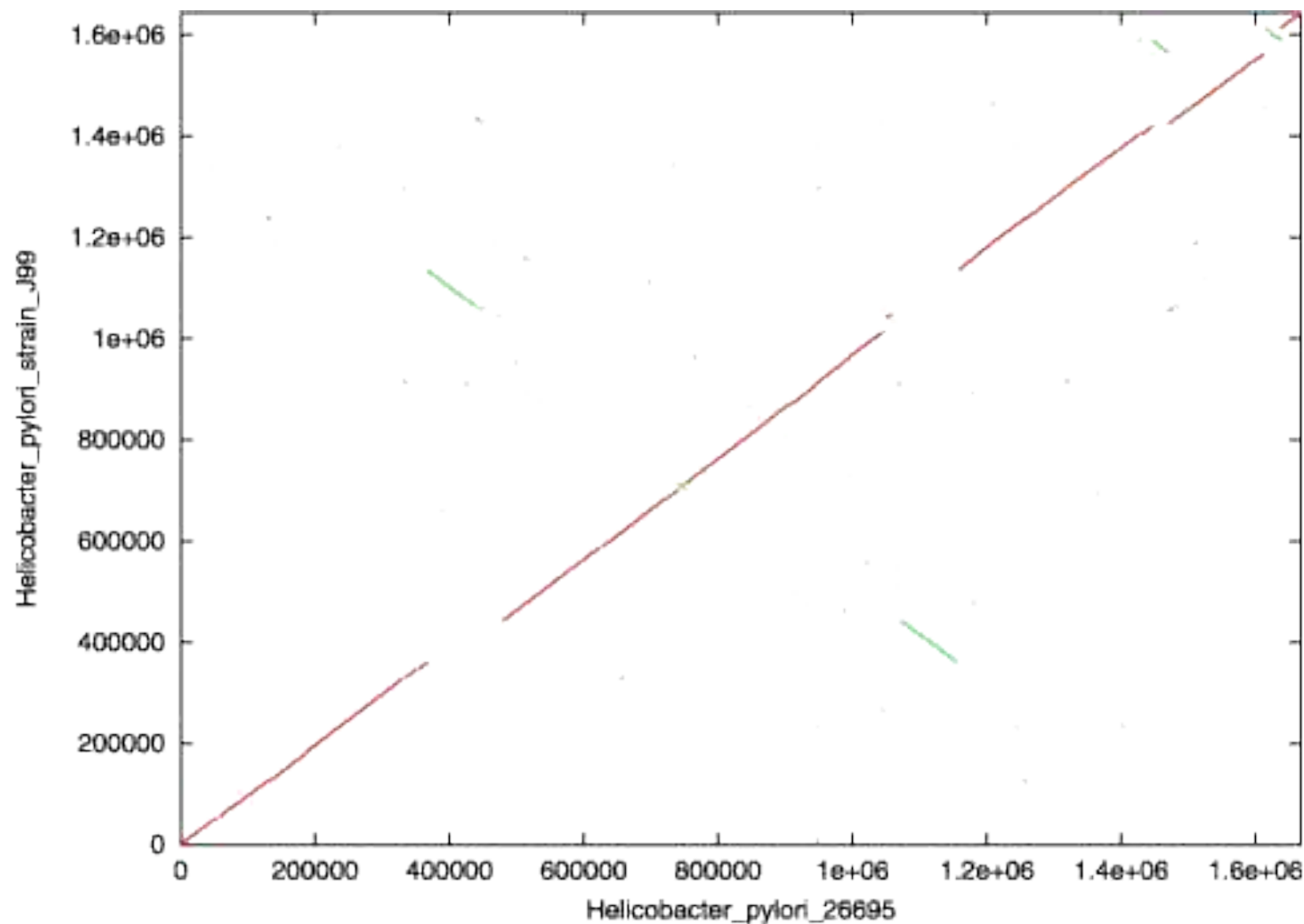
$O(n)$ Step 1: walk down ab path
If we “fall off” there are no matches

$O(k)$ Step 2: visit all leaf nodes below
Report each leaf offset as match offset

$O(n + k)$ time



Suffix tree application: find long common substrings



Dots are *maximal unique matches (MUMs)*, a kind of long substring shared by two sequences

Red = match was between like strands,
green = different strands

Axes show different strains of *Helicobacter pylori*, a bacterium found in the stomach and associated with gastric ulcers

Suffix tree application: find longest common substring

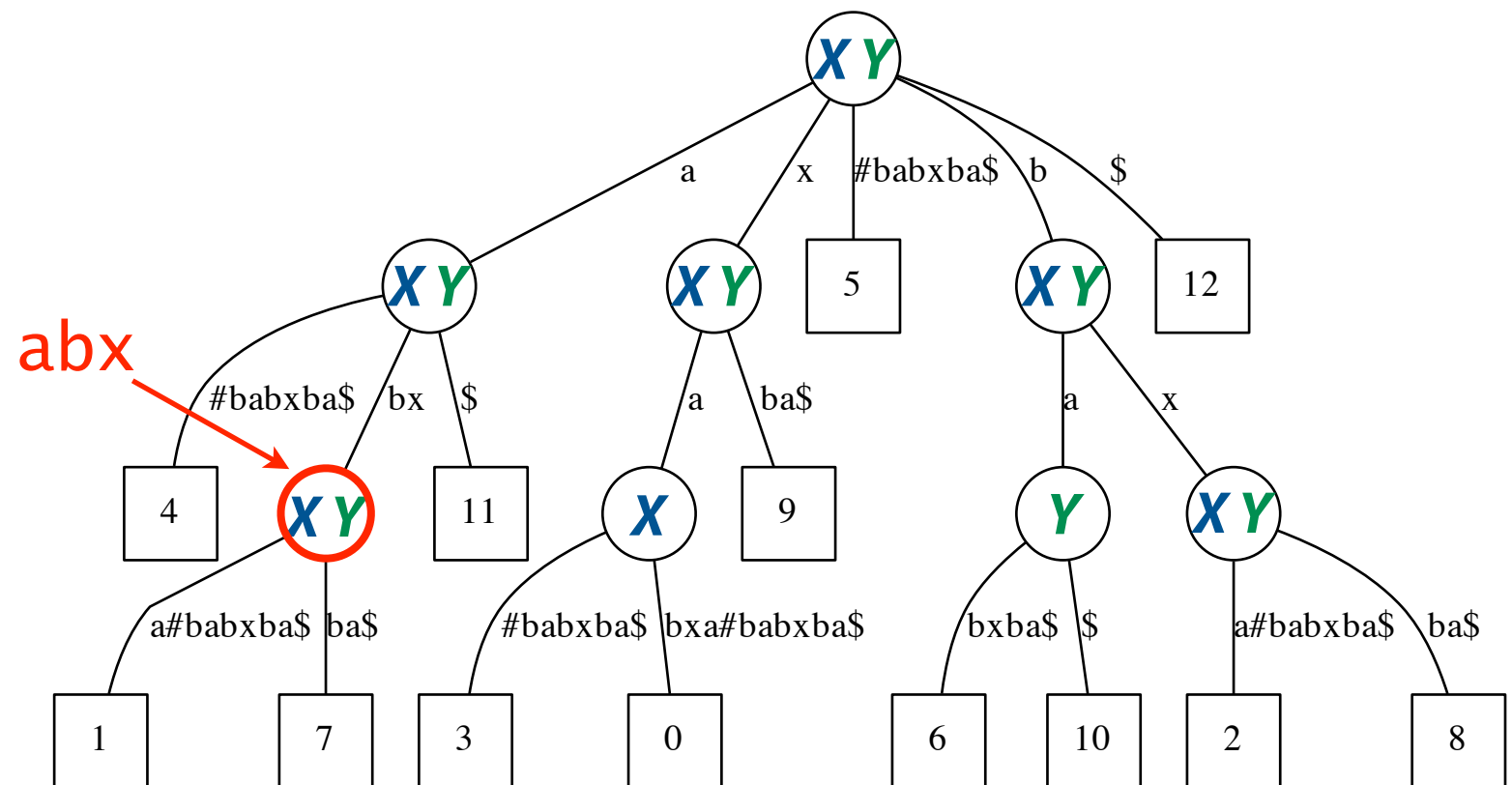
To find the longest common substring (LCS) of X and Y , make a new string $X\#Y\$$ where $\#$ and $\$$ are both terminal symbols. Build a suffix tree for $X\#Y\$$.

$X = \text{xabxa}$

$Y = \text{babxba}$

$X\#Y\$ = \text{xabxa}\#\text{babxba}\$$

Consider leaves:
offsets in $[0, 4]$ are
suffixes of X , offsets in
 $[6, 11]$ are suffixes of Y



Traverse the tree and annotate each node according to whether leaves below it include suffixes of X , Y or both

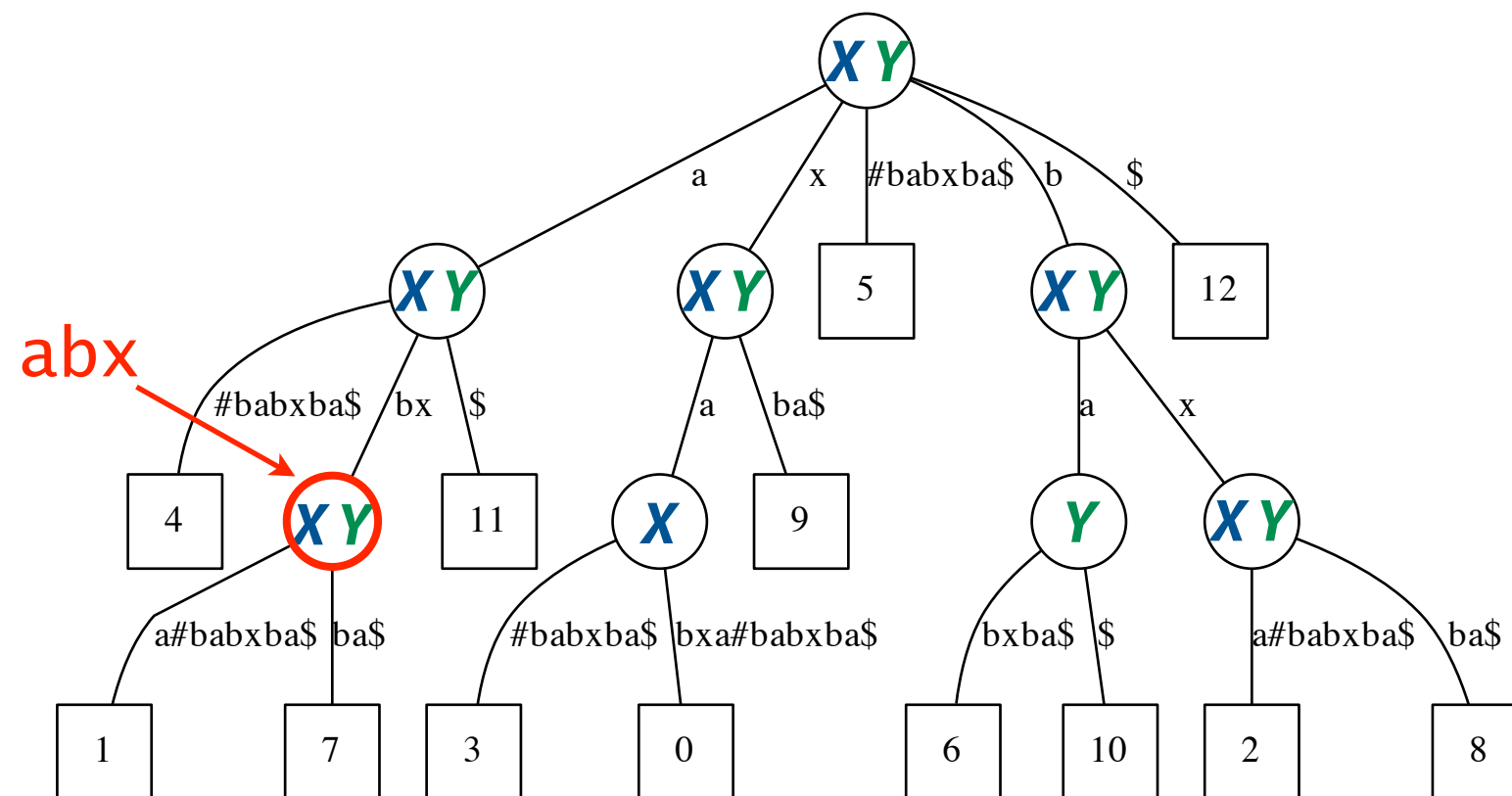
The deepest node annotated with both X and Y has LCS as its label.

$O(|X| + |Y|)$ time and space.

Suffix tree application: generalized suffix trees

This is one example of many applications where it is useful to build a suffix tree over many strings at once

Such a tree is called a *generalized suffix tree*. These are introduced in *Gusfield* 6.4.



Suffix trees in the real world: MUMmer

FASTA file containing "reference" ("text")

FASTA file containing
ALU string

Indexing
phase: ~2
minutes

Matching
phase:
very fast

```
mummer — langmead@lgm1:~ — bash — 120x31
Bens-MacBook-Pro:mummer langmead$ cat alu50.fa
>Alu
GCGCGGTGGCTCACGCCTGTAATCCAGCACTTTGGGAGGCCGAGGCGGG
Bens-MacBook-Pro:mummer langmead$ $HOME/software/MUMmer3.23/mummer -maxmatch $HOME/fasta/hg19/chr1.fa alu50.fa
# reading input file "/Users/langmead/fasta/hg19/chr1.fa" of length 249250621
# construct suffix tree for sequence of length 249250621
# (maximum reference length is 536870908)
# (maximum query length is 4294967295)
# process 2492506 characters per dot
# .....
# CONSTRUCTIONTIME /Users/langmead/software/MUMmer3.23/mummer /Users/langmead/fasta/hg19/chr1.fa 125.30
# reading input file "alu50.fa" of length 50
# matching query-file "alu50.fa"
# against subject-file "/Users/langmead/fasta/hg19/chr1.fa"
> Alu
61769671      1      22
219929011     1      22
162396657     1      22
109737840     1      22
82615090      1      22
32983678      1      22
84730371      1      22
248036256     1      22
150558745     1      22
11127213      1      22
236885661     1      22
31639677      1      22
16027333      1      22
21577225      1      22
26327837      1      22
243352583     1      22
⋮
```

Columns:

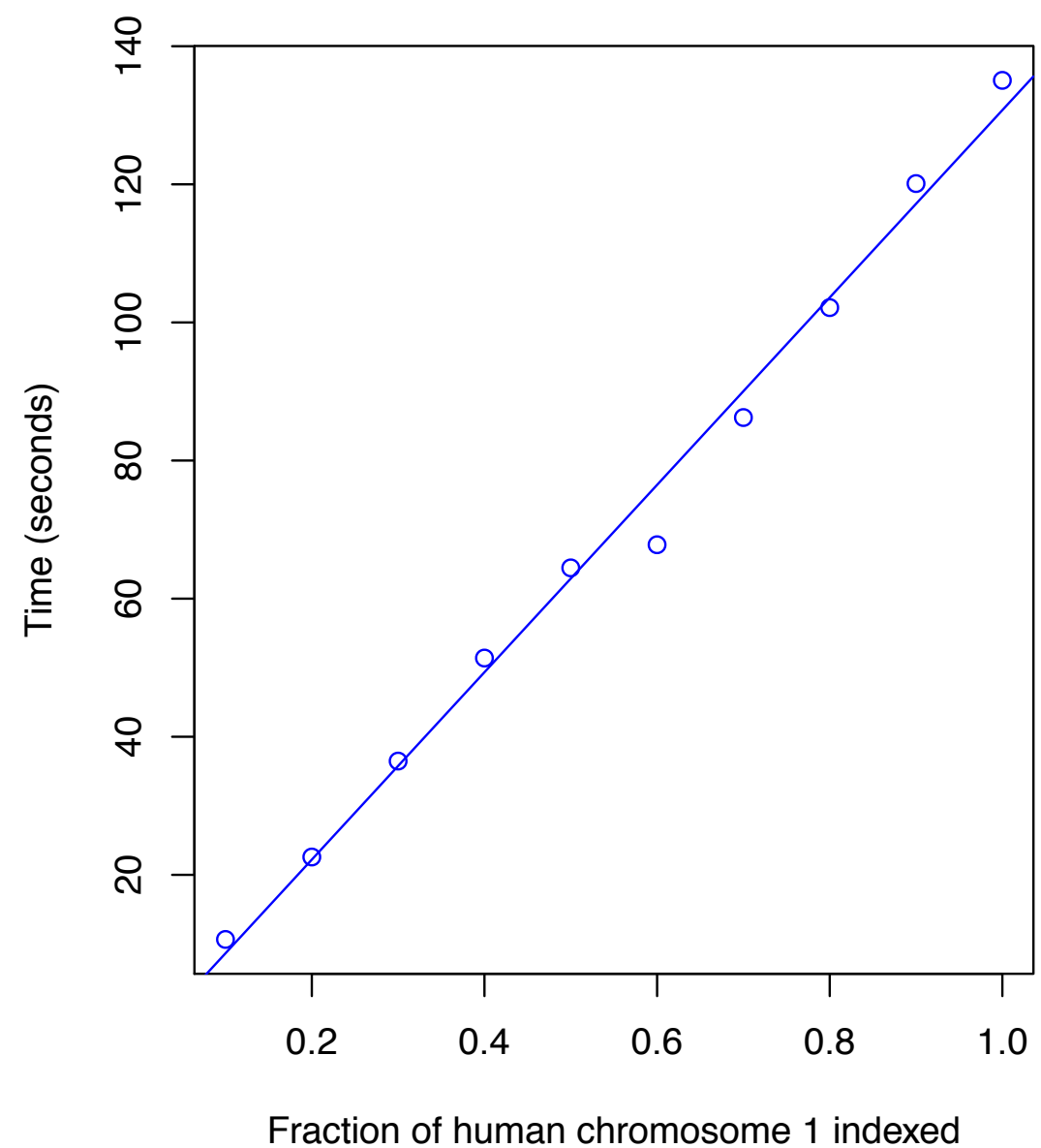
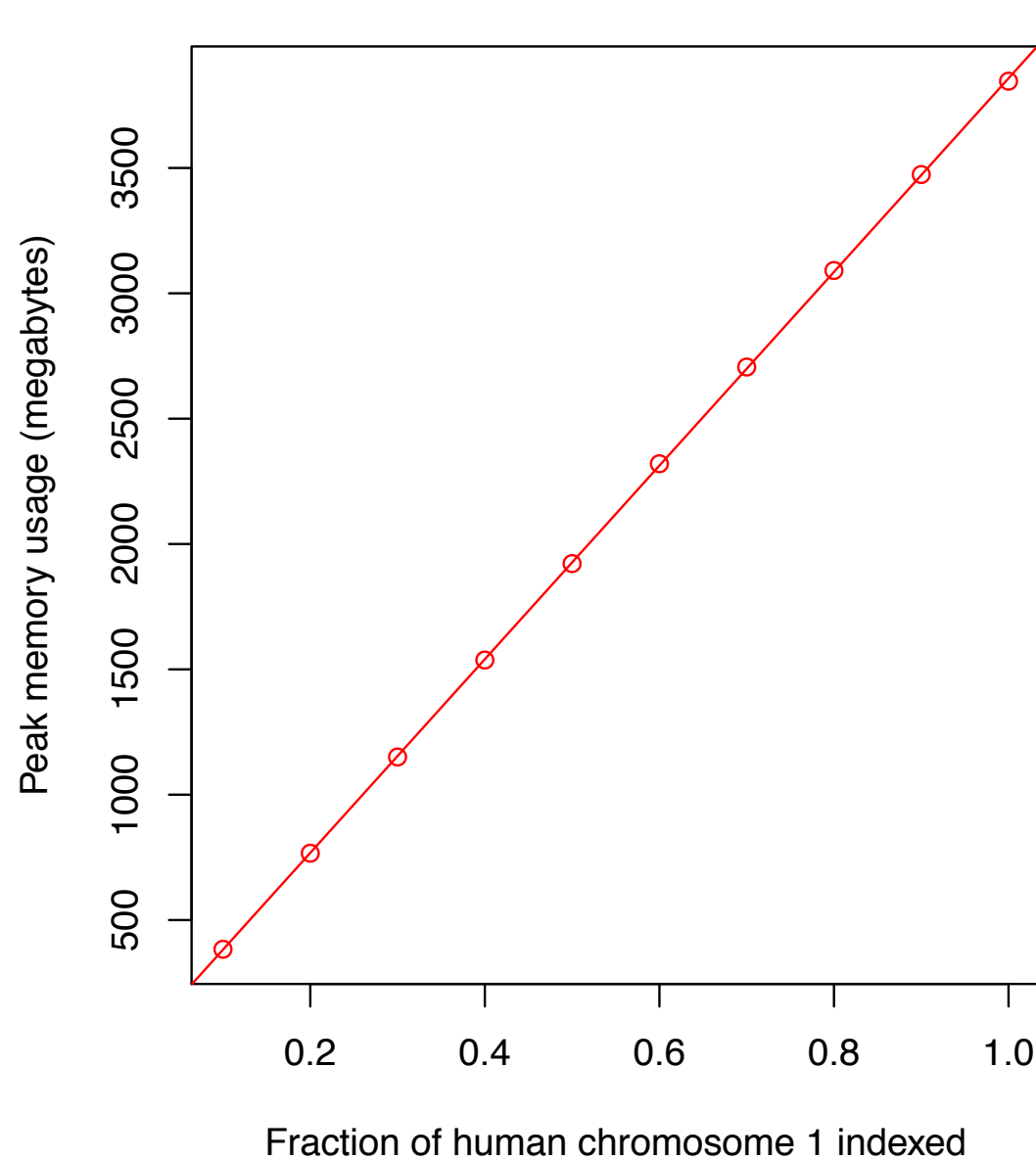
1. Match offset in T

2. Match offset in P

3. Length of exact match

Suffix trees in the real world: MUMmer

MUMmer v3.32 time and memory scaling when indexing increasingly larger fractions of human chromosome 1



For whole chromosome 1, took 2m:14s and used 3.94 GB memory

Suffix trees in the real world: MUMmer

Attempt to build index for whole human genome reference:

```
mummer: suffix tree construction failed: textlen=3101804822  
larger than maximal textlen=536870908
```

We can predict it would have taken about 47 GB of memory

Suffix trees in the real world: the constant factor

While $O(m)$ is desirable, the constant in front of the m limits wider use of suffix trees in practice

Constant factor varies depending on implementation:

Estimate of MUMmer's constant factor = 3.94 GB / 250 million nt
 \approx **15.75 bytes per node**

Literature reports implementations achieving as little as 8.5 bytes per node, but no implementation used in practice that I know of is better than \approx **12.5 bytes per node**

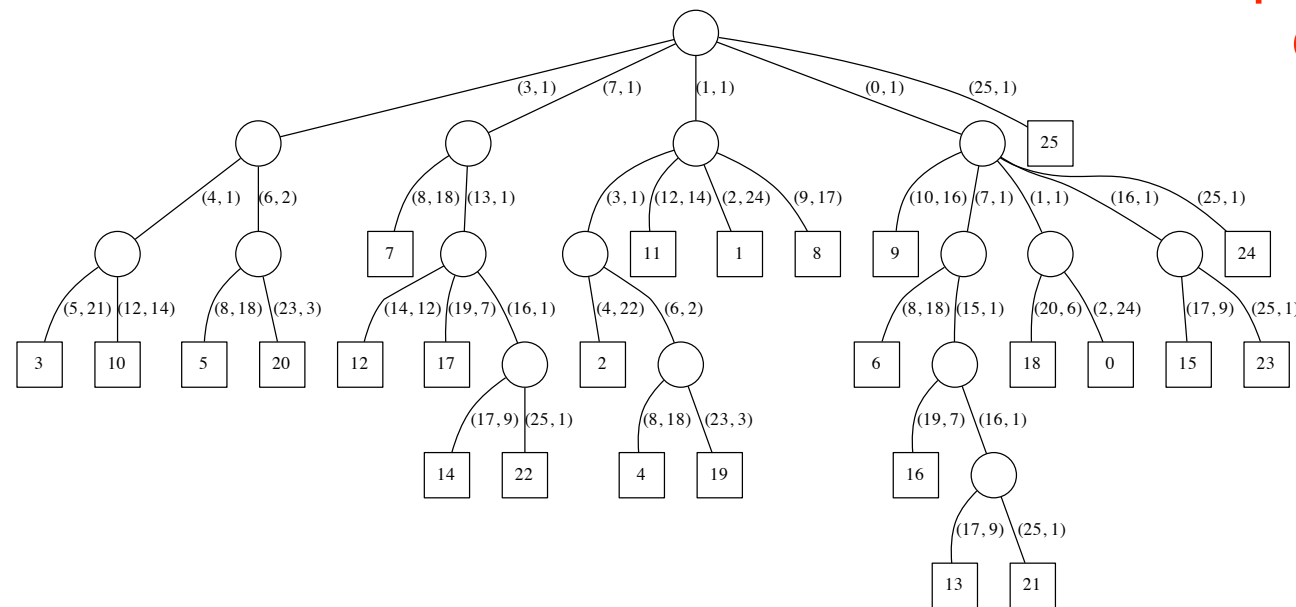
Kurtz, Stefan. "Reducing the space requirement of suffix trees." *Software Practice and Experience* 29.13 (1999): 1149-1171.

Suffix tree: summary

Organizes all suffixes into an incredibly useful, flexible data structure, in $O(m)$ time and space

A naive method (e.g. suffix trie) could easily be quadratic or worse

Used in practice for whole genome alignment, repeat identification, etc



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

$m(m+1)/2$ chars

Actual memory footprint (bytes per node) is quite high, limiting usefulness