Chapter 7 - Pattern Matching

J. Matthew Holt
holtjma@cs.unc.edu

Sequence Alignment

- Sequencing data
  - Millions to billions of reads
  - Typically 100+ basepairs
- Reference genome - millions to billions of basepairs
- Where does a read best match the reference genome?

```
Reference: ACGAACAAGTAGTCCCAGTACGTACATGCAAGT
Read 1:    GTACATG
Read 2:    CAAGTAG
Read 3:    CCCAGTA
…
```

Pattern Matching Problem

- Goal: Find all occurrences of a pattern in a text
- Input:
  - Pattern p: $p_1 p_2 \ldots p_n$
  - Text t: $t_1 t_2 \ldots t_m$
- Output: All positions $1 \leq i \leq (m - n - 1)$ such that the n-letter substring starting at i matches the pattern p
- Motivation: given a sequencing read (a pattern) can we match it to a reference genome (a text)

Exact pattern matching: Brute-force approach

p - the pattern to search for
t - the text to search through

```
def bruteForcePatternMatching(p, t):
    ret = []
    for x in xrange(0, len(t)-len(p)+1):
        if t[x:x+len(p)] == p:
            ret.append(x)
    return ret
```

Brute-force Example

Text: ACGAGATT
Pattern: GAG

```
01234567
ACGAGATT
GAG
 GAG
  GAG
   GAG
    GAG
     GAG
```

Returns [2, 4]

Brute-force pattern matching performance

- Performance:
  - m - length of the text t
  - n - the length of the pattern p
  - Looping - O(m) total loops
  - Comparison - O(n) per loop
  - Total cost - O(mn) <u>per pattern</u>
- In practice, total cost is closer to O(m) because comparisons terminate early
- Worst-case example:
  - p = “AAAT”
  - t = “AAAAAAAAAAAAAAAAAAAAAAT”
- What if we have multiple patterns?

Exact Pattern Matching Can we do better?

```
Text:    MISSISSIPPISSI
Pattern: ISSI
```

- What if we do some pattern pre-processing?
- We can skip some start points

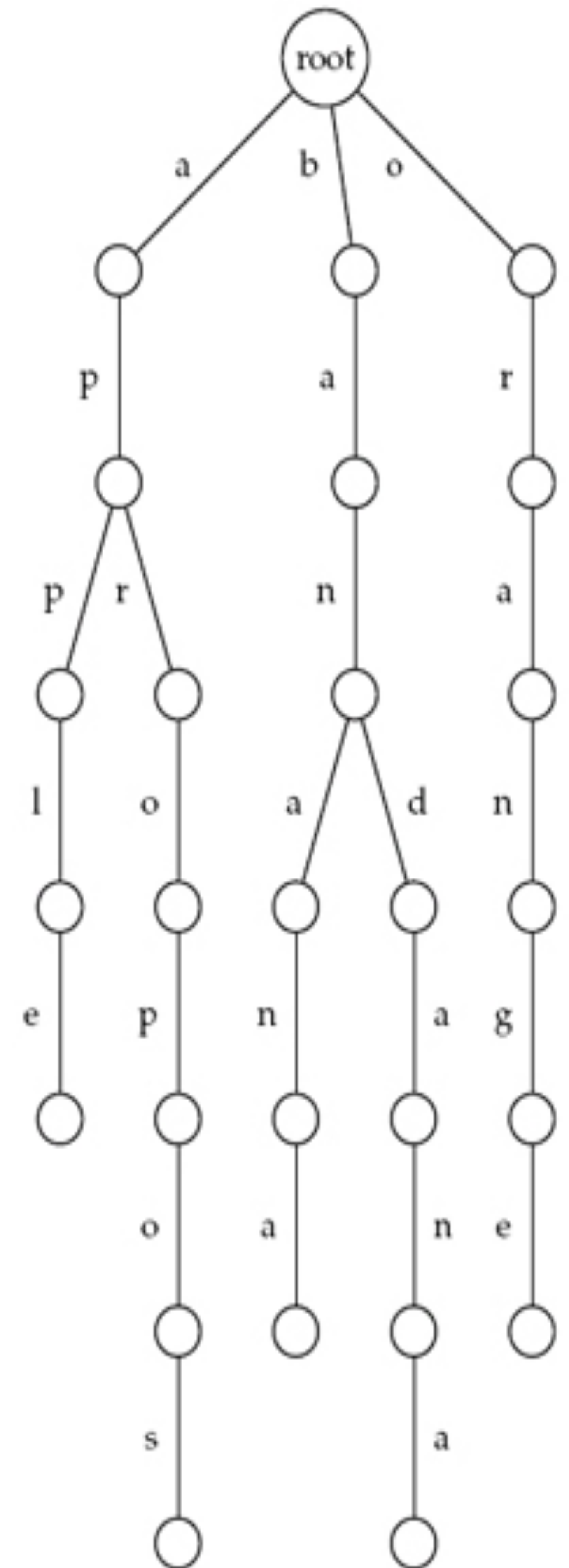
```
MISSISSIPPISSI
ISSI
 ISSI
    ISSI
       ISSI
         ISSI
          ISSI
```

Multiple Pattern Matching Problem

- Goal: find all occurrences of a set of patterns (reads) in a text (reference genome)
- Input:
  - A collection of patterns P that are at most d character long
  - A text t of length m
- Output: All pairs of index $0 \leq i \leq m-1$ and pattern p in P such that the suffix starting at i begins with pattern p

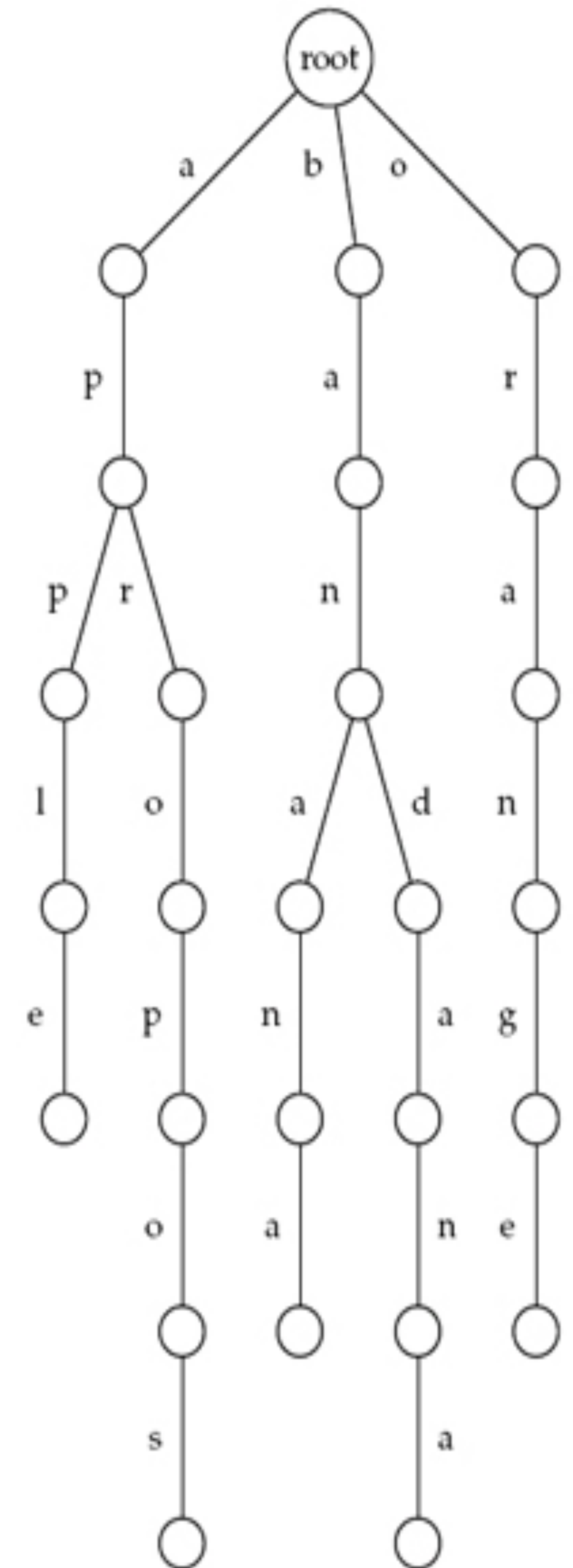
Keyword Tries

- Input: a collection of patterns or keywords
- Output: a tree-like structure where edges are characters and terminal nodes represent a pattern
- Example:
  - apple
  - apropos
  - banana
  - bandana
  - orange

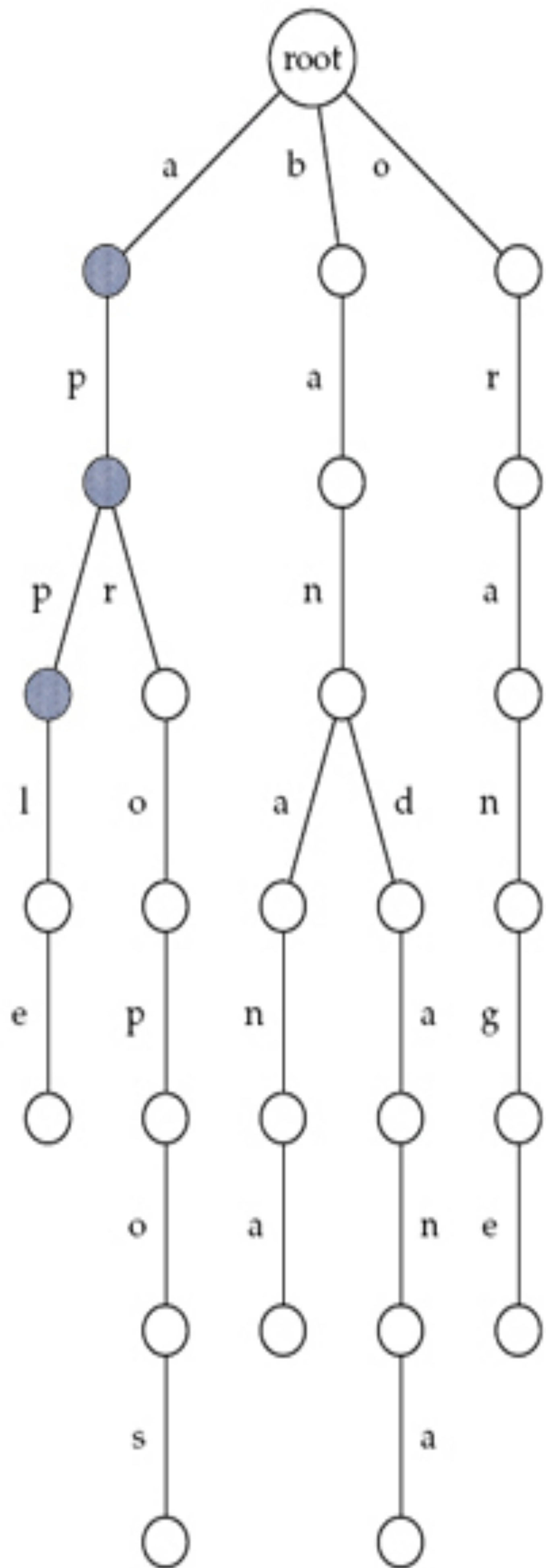


Prefix Trie Matching

- Input: a text *t* and a trie *P* of patterns
- Output: any patterns that match a prefix of *t*



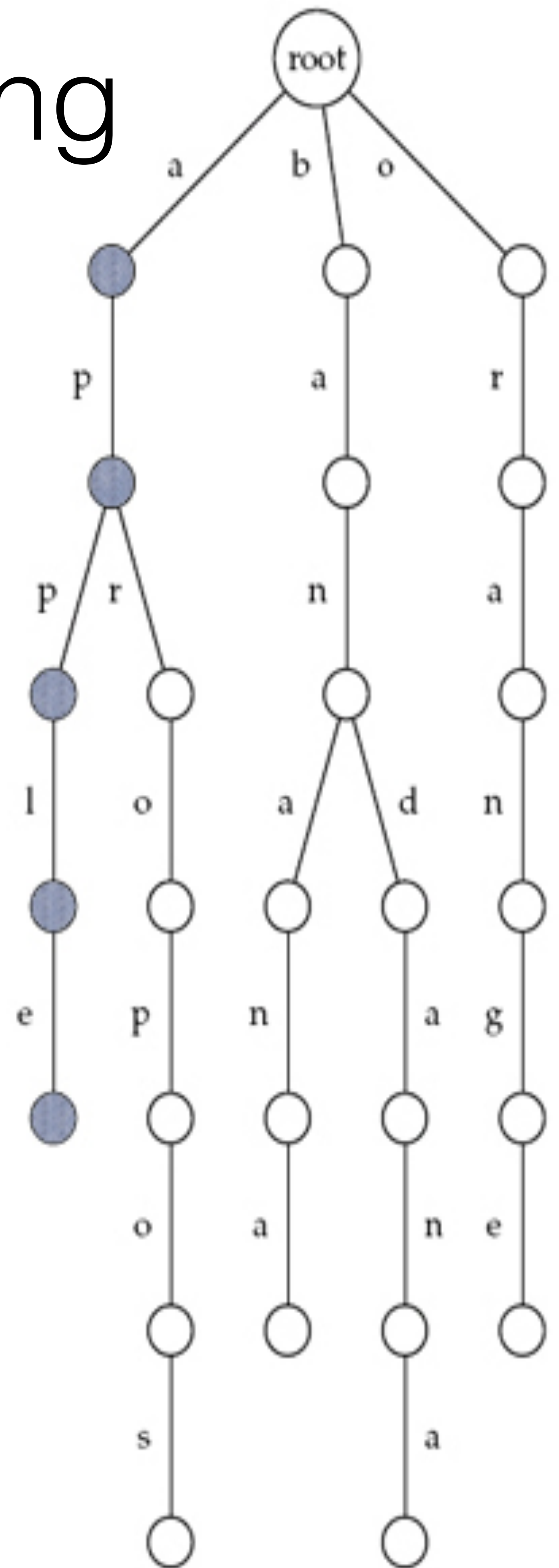
Prefix Trie Matching Examples



Thread appeal
Not terminal

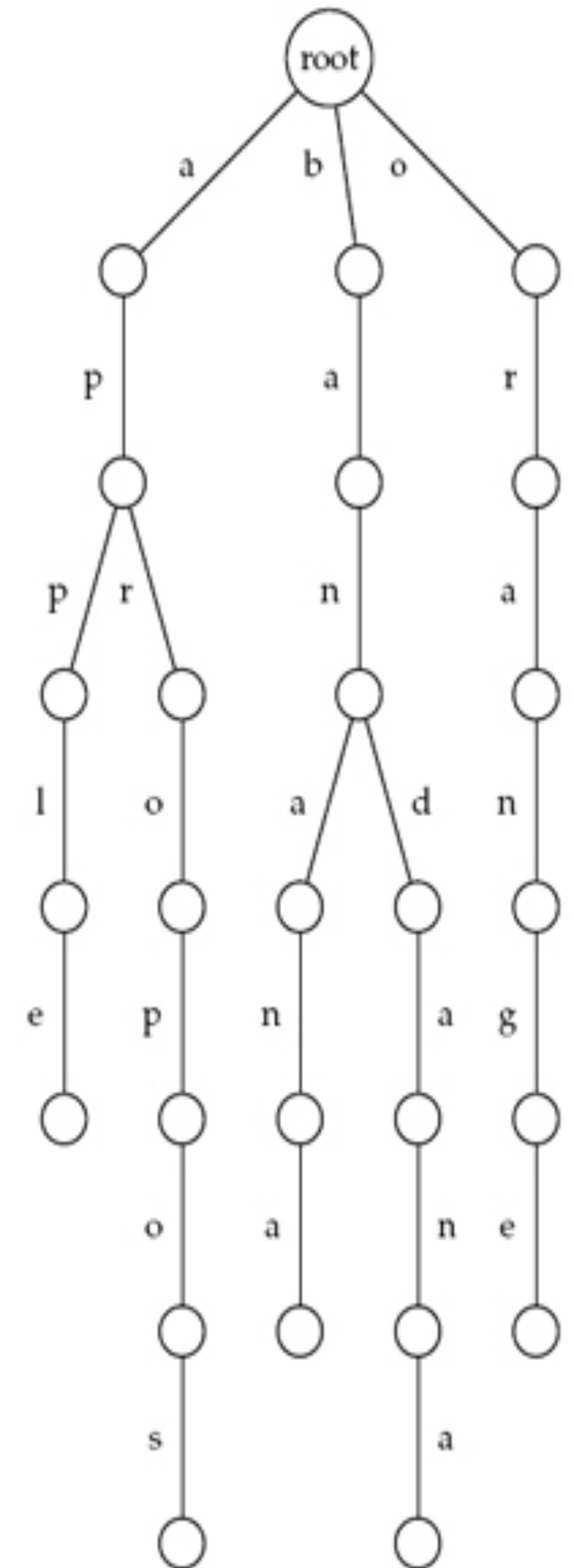
Thread apple
Terminal

Thread apples
Terminal



Prefix Trie Matching (cont.)

- Input: a text *t* and a trie *P* of patterns
- Output: any patterns that match a prefix of *t*
- Computation:
  - Requires threading *t* through *P*
  - Worst case is length of longest pattern in *P* (or the depth *d* of *P*)
  - O(*d*) time



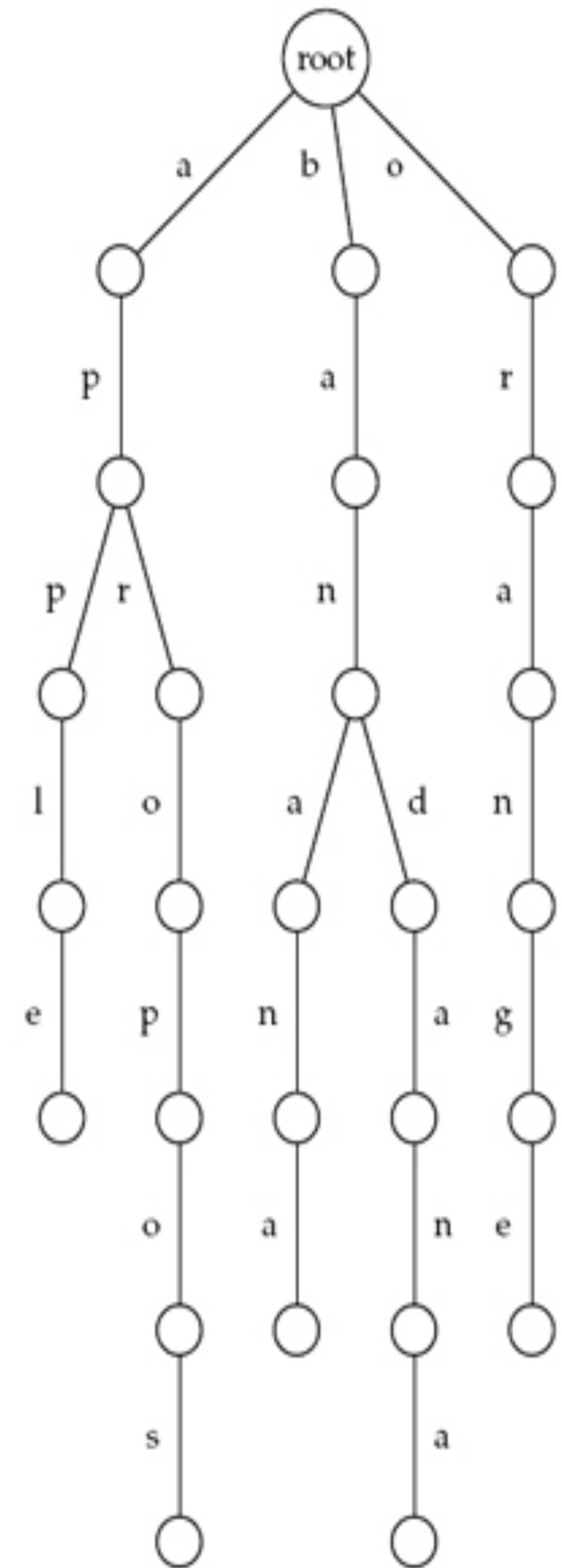
Multiple Pattern Matching

t - the text to search through
P - the trie of patterns to search for

```
def multiplePatternMatching(t, P):
    ret = []
    for x in xrange(0, len(t)):
        pattern = PrefixTrieMatching(t[x:], P)
        if pattern != None:
            ret.append((x, pattern))
    return ret
```

Multiple Pattern Matching (cont.)

```
multiplePatternMatching("bananapple", P)
  PrefixTrieMatching("bananapple", P) = "banana"
  PrefixTrieMatching("ananapple", P) = None
  PrefixTrieMatching("nanapple", P) = None
  PrefixTrieMatching("anapple", P) = None
  PrefixTrieMatching("napple", P) = None
  PrefixTrieMatching("apple", P) = "apple"
  PrefixTrieMatching("pple", P) = None
  PrefixTrieMatching("ple", P) = None
  PrefixTrieMatching("le", P) = None
  PrefixTrieMatching("e", P) = None
ret = [(0, "banana"), (5, "apple")]
```

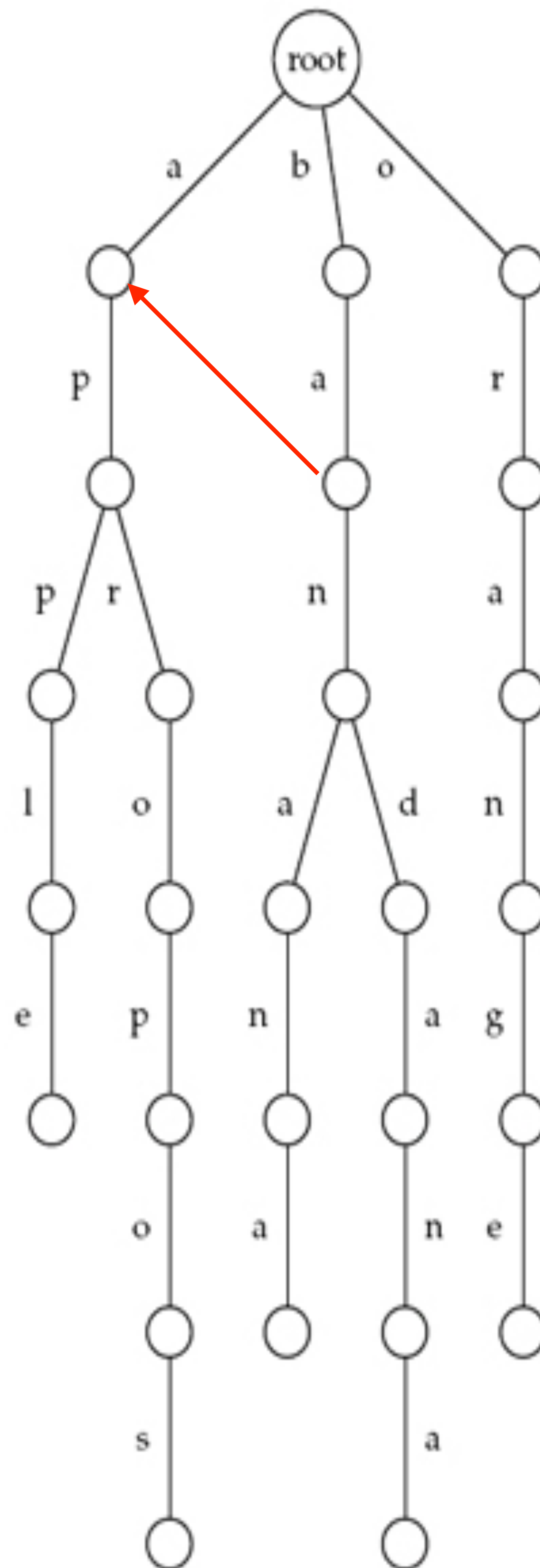


Can we do better?

Remember our earlier speedup

Can add “failure edges” to our trie

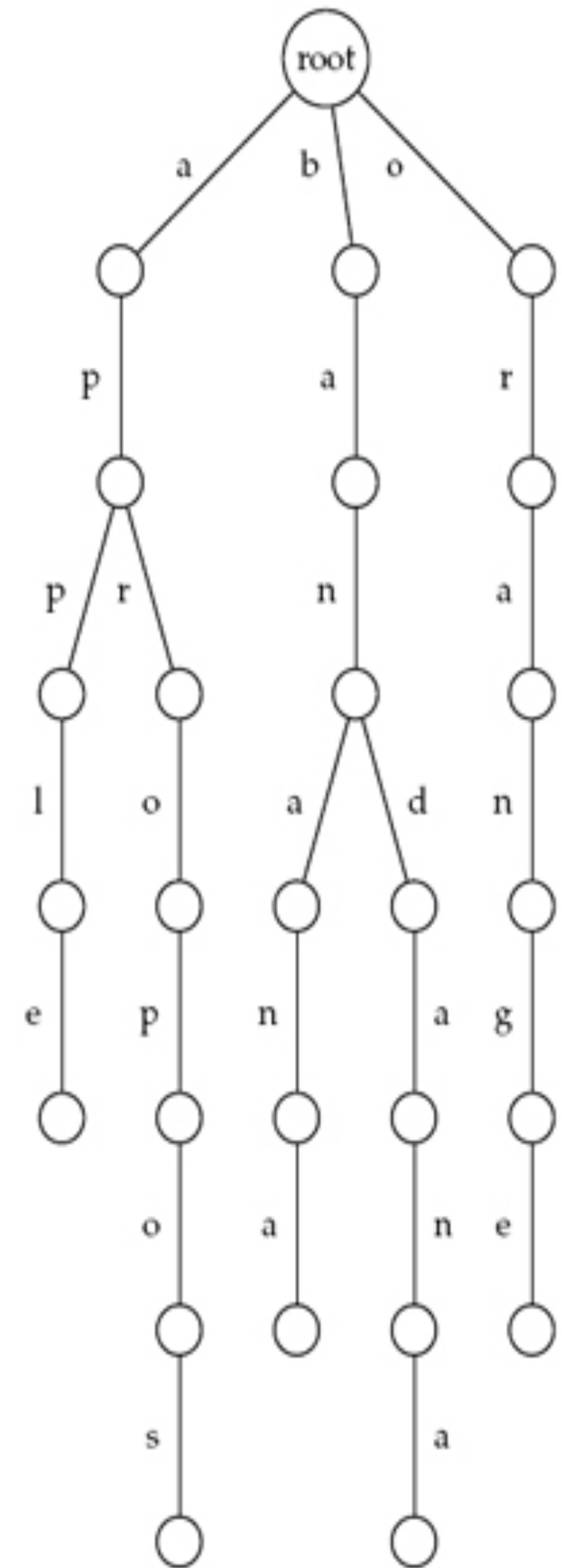
“Aho-Corasick” Algorithm



bapple
bap
apple

Multiple Pattern Matching (cont.)

- Run time
    - m - len(t)
    - d - max depth of P (longest pattern in P)
    - O(md) to find all patterns
    - Can be decreased further to O(m) using Aho-Corasick Algorithm (see pg 353)
- Memory issues
    - Tries require a lot of memory
    - Practical implementation is challenging
    - Genomic reads - millions to billions of patterns typically of length $\geq$ 100



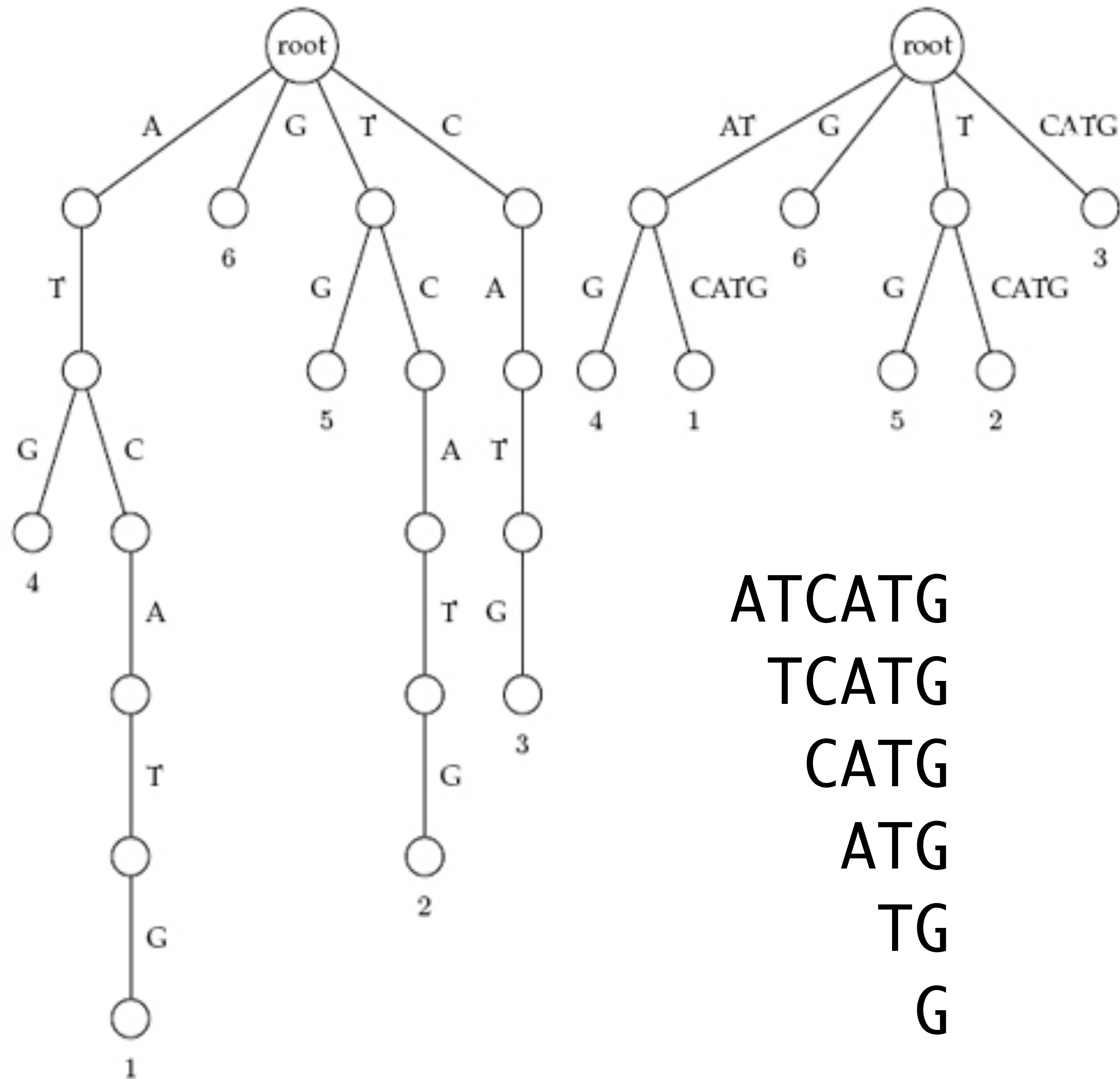
Preprocess genome into suffix tree

- Input:
  - A single long string (reference genome)
  - Example - “ATCATG”
- Output:
  - A tree containing all suffixes of the input
  - The tree is also compressed

```
ATCATG
 TCATG
  CATG
   ATG
    TG
     G
```

Preprocess genome into suffix tree (cont.)

- A **suffix trie** is a keyword trie of all suffixes (left figure)
- A **suffix tree** compresses the trie by combining nodes with out degree 1
  - Edges represent a substring of text
  - All internal nodes have at least 3 edges
  - All leaf nodes are labeled with an index

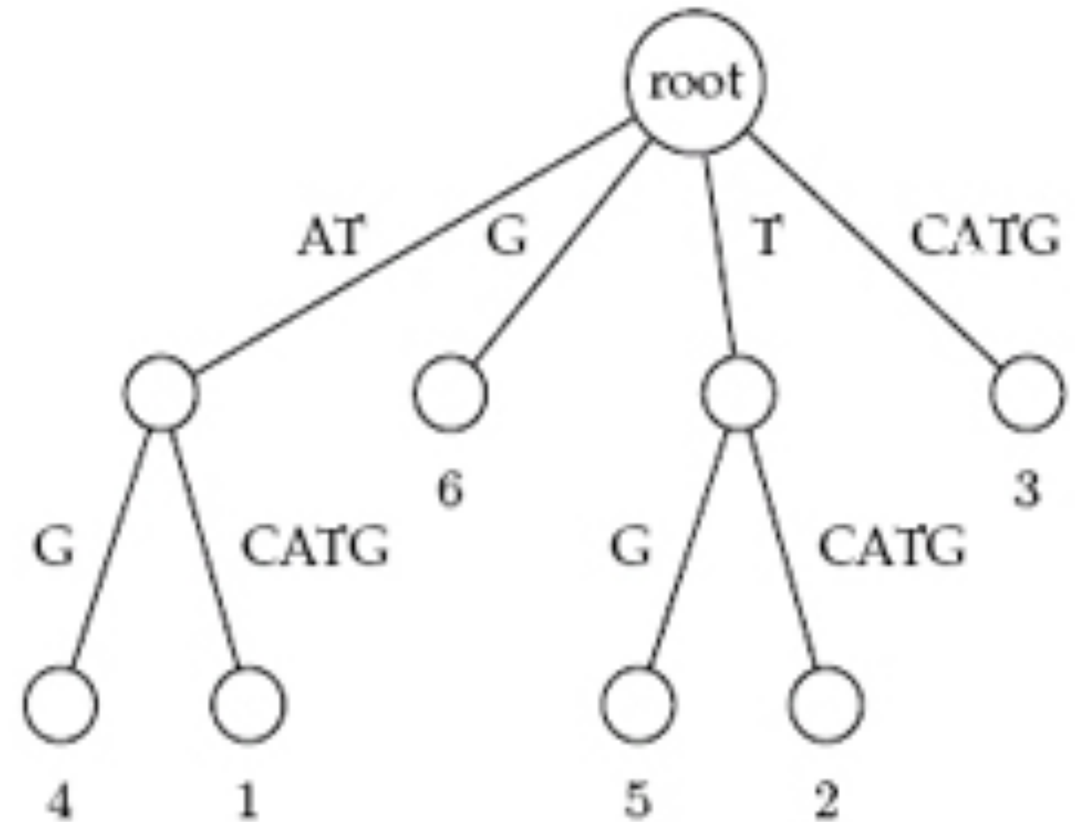


(a) Keyword tree

(b) Suffix tree

Suffix Tree (cont.)

- Nodes (fixed alphabet) - pointer to edge for each letter
  - DNA - 4 letters
  - ~16 bytes per node
- Edges are typically stored as (offset, length) pairs
  - "AT" = (0, 2)
  - "G" = (5, 1)
  - "CATG" = (2, 4)
  - ~8 bytes per edge




```
ATCATG
012345
```

Suffix trees summary

- Input: text of length m
- Computation
  - $O(m)$ to compute a suffix tree
  - Does not require building the suffix trie first
- Memory
  - $O(m)$ - nodes are stored as offsets and lengths
  - Huge hidden constant, best implementations requires about $20*m$ bytes
  - 3 GB human genome = 60 GB RAM

Suffix arrays

- Related to suffix trees, but reduced memory
  - Keep string on disk O(*m*)
  - Keep array of pointers indicating sorted order of suffixes O(*m**log(*m*)) bits
  - In practice the log(*m*) is typically 4 bytes
- Computation
  - Can also be done in O(*m*) time
  - Uses O(*m*) memory; approximately 12 GB for the human genome

```
13 $
5  abananas$
3  amabananas$
1  anamabananas$
7  ananas$
9  anas$
11 as$
6  bananas$
4  mabananas$
2  namabananas$
8  nanas$
10 nas$
0  panamabananas$
12 s$
```

The suffix array (and suffixes) for the string "panamabananas$"

Constructing Suffix Array (naive method)

```
def buildSuffixArray(t):
    sa = sorted(range(len(t)), cmp=lambda i,j: (-1 if t[i:] < t[j:] else 1))
    return sa
```

- Creates a list of indices
- Sorts the list using a custom comparator
- Returns a list of indices sorted by the suffix starting at that position
- Note: this basic code is not recommended for real applications; runs in $O(m^2 * \log(m))$ time

Searching suffix arrays

- Binary search
  - Pattern length *d*
  - O(log(*m*)) comparisons
  - Worst case is O(*d*) operations per comparison
  - O(*d**log(*m*)) per pattern

```
13 $
5  abananas$
3  amabananas$
1  anamabananas$
7  ananas$
9  anas$
11 as$
6  bananas$
4  mabananas$
2  namabananas$
8  nanas$
10 nas$
0  panamabananas$
12 s$
```

Searching suffix arrays

```
def searchFirst(sa, t, p):
  l = 0
  h = len(t)
  while l < h:
    m = (l+h)/2
    if t[sa[m]:] < p:
      l = m+1
    else:
      h = m
  return l
```

searchFirst(sa, t, 'am')
(0, 14) -> (0, 7) -> (0, 3) -> (2, 3) -> (2, 2)
return 2

i	sa[i]	t[sa[i]]
0	13	$
1	5	abananas$
2	3	amabananas$
3	1	anamabananas$
4	7	ananas$
5	9	anas$
6	11	as$
7	6	bananas$
8	4	mabananas$
9	2	namabananas$
10	8	nanas$
11	10	nas$
12	0	panamabananas$
13	12	s$

Searching suffix arrays

```
def searchLast(sa, t, p):
  l = 0
  h = len(t)
  while l < h:
    m = (l+h)/2
    if t[sa[m]:sa[m]+len(p)] <= p:
      l = m+1
    else:
      h = m
  return l-1
```

Summary

m - length of text
d - max length of pattern
x - number of patterns

Method	Storage Cost	Single Pattern Search Time	Multiple Pattern Search Time
Brute Force	$O(m)$	$O(dm)$	$O(xdm)$
Keyword Tries	$O(xd)$	$O(dm)$	$O(dm)$
Suffix Trees	$O(m)$ [20m bytes]	$O(d)$	$O(xd)$
Suffix Arrays	$O(m*\log(m))$ [4m bytes]	$O(d*\log(m))$	$O(xd*\log(m))$

Next class: Burrows-Wheeler Transform

m - length of text
d - max length of pattern
x - number of patterns

Method	Storage Cost	Single Pattern Search Time	Multiple Pattern Search Time
Brute Force	$O(m)$	$O(dm)$	$O(xdm)$
Keyword Tries	$O(xd)$	$O(dm)$	$O(dm)$
Suffix Trees	$O(m)$ [20m bytes]	$O(d)$	$O(xd)$
Suffix Arrays	$O(m*\log(m))$ [4m bytes]	$O(d*\log(m))$	$O(xd*\log(m))$
BWT	**$O(m)$ [often m bits]**	**$O(d)$**	**$O(xd)$**