# Read Mapping

## Burrows Wheeler Transform and Reference Based Assembly

Peter N. Robinson

Institut für Medizinische Genetik und Humangenetik
Charité Universitätsmedizin Berlin

Genomics: Lecture #3 WS 2014/2015

## Today

- Reference based assembly: what's the goal?
- Exploiting the data for reference based assembly: from naive algorithms to suffix trees/arrays
- Discussion of suffix based string index algorithms
- Goal is to review background needed to understand the Burrows Wheeler Transform and bwa for reference based genome alignment (next time)

## Outline

## Reference-Based Assembly

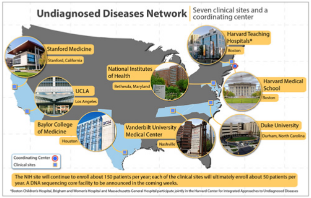
Referenced based assembly follows the goal of finding the **differences** between an individual's genome and the reference genome for the corresponding species, rather than characterizing the genome of that species in the first place.

- A major application is in medical diagnostics
- Other applications include the characterization of variation in model organisms such as mice and plant and animal breeding programs

## Genomic Diagnostics: A Paradigm Shift in Medicine?

Genomics england

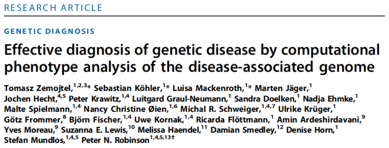
- UK 100,000 Genomes Project
- sequence 100,000 whole genomes from NHS patients by 2017.


Undiagnosed Diseases Network
Seven clinical sites and a coordinating center
Stanford Medicine
Stanford, California
UCLA
Los Angeles
National Institutes of Health
Bethesda, Maryland
Harvard Teaching Hospitals*
Boston
Harvard Medical School
Boston
Baylor College of Medicine
Houston
Vanderbilt University Medical Center
Nashville
Duke University
Durham, North Carolina
Coordinating Center
Clinical sites
The NIH site will continue to enroll about 150 patients per year; each of the clinical sites will ultimately enroll about 50 patients per year. A DNA sequencing core facility to be announced in the coming weeks.
*Boston Children's Hospital, Brigham and Women's Hospital and Massachusetts General Hospital participate jointly in the Harvard Center for Integrated Approaches to Undiagnosed Diseases

- NIH Undiagnosed Diseases Network

Global Alliance for Genomics & Health

- effective and responsible sharing of genomic and clinical data


RESEARCH ARTICLE

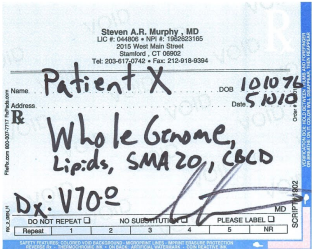
GENETIC DIAGNOSIS

**Effective diagnosis of genetic disease by computational phenotype analysis of the disease-associated genome**

Tomasz Zemojtel,[1,2,3]* Sebastian Köhler,[1]* Luisa Mackenroth,[1]* Marten Jäger,[1] Jochen Hecht,[4,5] Peter Krawitz,[1,4] Luitgard Graul-Neumann,[1] Sandra Doelken,[1] Nadja Ehmke,[1] Malte Spielmann,[1,4] Nancy Christine Øien,[1,6] Michal R. Schweiger,[1,4,7] Ulrike Krüger,[1] Götz Frommer,[8] Björn Fischer,[1,4] Uwe Kornak,[1,4] Ricarda Flöttmann,[1] Amin Ardeshirdavani,[9] Yves Moreau,[9] Suzanna E. Lewis,[10] Melissa Haendel,[11] Damian Smedley,[12] Denise Horn,[1] Stefan Mundlos,[1,4,5] Peter N. Robinson[1,4,5,13]†

- *Sci Transl Med* **6**:252ra123 (2014)

## A standard clinical test?



Steven A.R. Murphy , MD
LIC #: 044806 • NPI #: 1962623165
2015 West Main Street
Stamford , CT 06902
Tel: 203-617-0742 • Fax: 212-918-9394

Name. Patient X DOB 101076
Address. Date 51010

℞ Whole Genome,
Lipids, SMA 20, CBCD

Dx: V70⁰

[illegible] MD

DO NOT REPEAT ☐ NO SUBSTITUTION ☐ PLEASE LABEL ☐

| | | | | | | |
|---|---|---|---|---|---|---|
| Repeat | 1 | 2 | 3 | 4 | 5 | NR |

SAFETY FEATURES: COLORED VOID BACKGROUND - MICROPRINT LINES - IMPRINT ERASURE PROTECTION
REVERSE Rx - THERMOCHROMIC INK • ON BACK: ARTIFICIAL WATERMARK - COIN REACTIVE INK

FileRx.com 800-307-7717 RxPads.com

RX_2_GEN_H

SCRIPT#1902

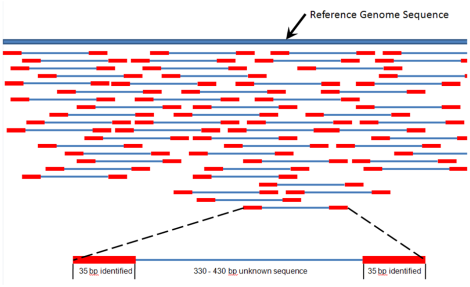
VERIFICATION BOX: HOLD BETWEEN THUMB AND FOREFINGER OR BREATHE ON IT. COLOR WILL DISAPPEAR, THEN REAPPEAR

- Genome (or exome) sequencing can be extremely useful for patients with rare diseases and cancer
- It is still not very useful for common disease

  (read the delightfully cogent blog of Dr Murphy: http://thegenesherpa.blogspot.de/, May 10, 2010)
- But: Clinical NGS resequencing is *rapidly* gaining in importance in many areas

## So why not use de novo assembly?

- de novo assembly algorithms with de Bruijn graphs are computational demanding and error-prone
- We would like to use our knowledge about the reference human genome to guide alignment of NGS reads of individual humans



## Outline

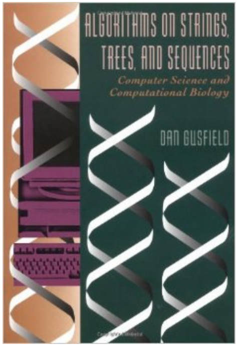
## So why not use de novo assembly?

- Faster search algorithms are based on preprocessing of the text (genome) to build a substring index
- Using the resulting data structure, occurrences of a pattern can be found quickly.
- a **substring index** is a data structure which gives substring search in a text or text collection in sublinear time.
  1. Suffix tree
  2. Suffix array
  3. FM index
  4. . . .

## String searches

String matching algorithms identify the positions where one or multiple strings are found as substrings of a larger string

- Classic book by Dan Gusfield
- Today: review of tries, suffix tree/array
- Next time: BWT, bwa


ALGORITHMS ON STRINGS, TREES, AND SEQUENCES
Computer Science and Computational Biology
DAN GUSFIELD

## Naive string search

Our "genome": bananasavannah ($n$ characters)
Our "read" (pattern): nas ($m$ characters)

- The simplest string algorithm would simply slide the pattern across the genome , extending it letter for letter as long as there is a match
- Run time $\mathcal{O}(nm)$
- Perfectly fine if we only have one read...

```
Bananasavannah
nas
Bananasavannah
 nas
Bananasavannah
  nas
Bananasavannah
   nas
Bananasavannah
    nas
```

## Naive string search

- If we do not have just one read we want to map, but say $\ell$ reads with a total length of $|\text{reads}|$ , then the runtime becomes $\mathcal{O}(n \cdot |\text{reads}|)$
- The combined length of the reads, $|\text{reads}|$ is hugh, often much bigger than the genome size
- (it is obvious that if we sequence a genome to 50x coverage then $|\text{reads}| \approx 50n$)
- In practice, this is extremely slow!

Let us now look at some slightly more intricate, but still naive ways of performing reference based genome alignment

## Tries

A trie (from re**trie**val), is a multi-way tree structure useful for storing strings over an alphabet.

- Usually pronounced like "try"
- A data structure for representing a collection of strings
- Fast pattern matching within this collection

```
banana$
bandana$
nasa$
anna$
Annex$
```

Let's make a trie for this collection of "reads"

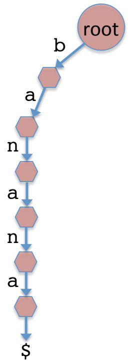
## Trie (1)

A trie is defined formally as the smallest tree over an alphabet $\Sigma$ such that

- Each edge of the trie is labeled with one character $c \in \Sigma$
- A node has at most one outgoing edge labeled with any given character $c$ for any $c \in \Sigma$
- Each key (string contained in the trie) is "spelled out" along some path starting at the root

Tries can be constructed to have all of the suffixes of some larger string be the keys

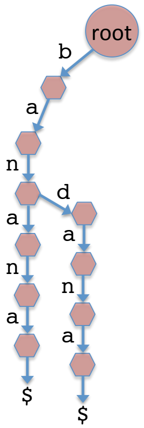
## Trie (1)



```
banana$
bandana$
nasa$
anna$
Annex$
```

- Add each word to the trie one at a time.
- The letters of the word label the edges, with one node/edge for each letter
- The dollar sign $ is a termination character

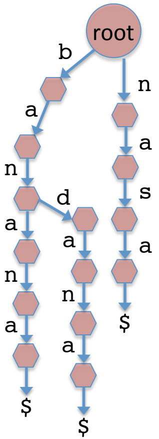
## Trie (2)



banana$
bandana$
nasa$
anna$
annex$

- Adding the second word bandana

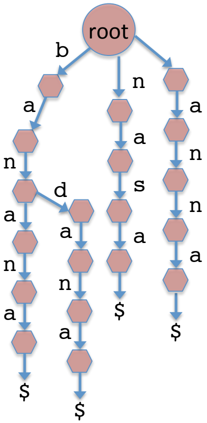
## Trie (3)



banana\$
bandana\$
nasa\$
anna\$
annex\$

- Adding the third word nasa

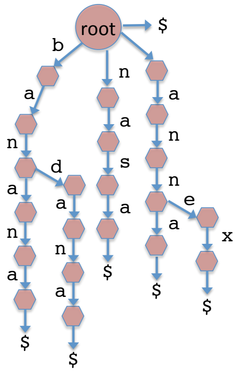
## Trie (4)



banana\$
bandana\$
nasa\$
anna\$
annex\$

- Adding the fourth word anna

## Trie (5)



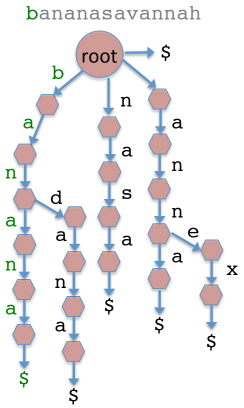
banana$
bandana$
nasa$
anna$
annex$

- Adding the fifth word annex
- Adding the termination character $

## Trie (5)

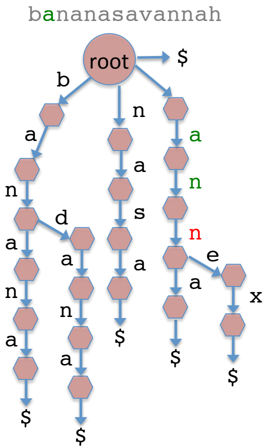
- Let us now use the trie to map reads to the genome
- instead of sliding individual reads down the genome one by one, we just slide the trie down the genome once

## Trie (6)



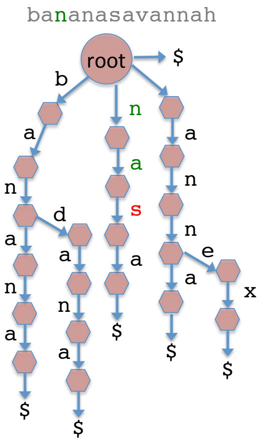
- Search for a pattern match starting at position 1 of genome
- Found banana

## Trie (7)



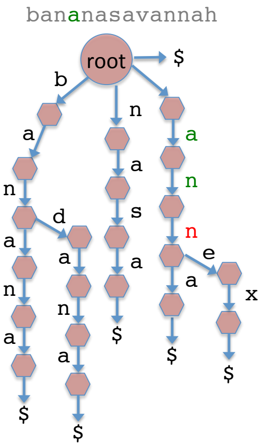
- Search for a pattern mach starting at position 2 of genome
- No match (only matched first two letters of anna$)

# Trie (8)



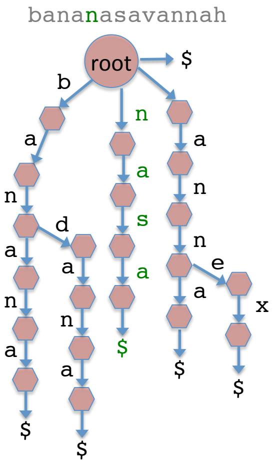
- Search for a pattern mach starting at position 3 of genome
- No match (only matched first two letters of `nasa$`)

## Trie (9)



- Search for a pattern mach starting at position 4 of genome
- No match (only matched first two letters of anna$)

# Trie (10)



- Search for a pattern mach starting at position 5 of genome
- Found nasa$

## Trie: That pesky Dollar sign

\$ is a symbol that does not appear anywhere else in our template $T$.

- We define it to be "less" than our other characters lexicographically
- For instance, for genomics we would have $\$ < A < C < G < T$
- The \$ enforces a lexicographic rule that we know from dictionaries: For instance, "over" comes before "overture".
- For instance $AC$ comes before $ACG$ because we are actually comparing $AC\$$ with $ACG\$$ and by definition \$ comes before $G$
- \$ also ensures that no suffix is a prefix of any other suffix

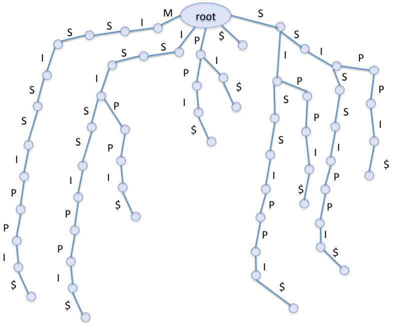
## Trie

- So what have we gained?
- Recall the running time of the simple naive algorithm was $\mathcal{O}(m \cdot |\mathrm{reads}|)$ with $n$=genome length and $|\mathrm{reads}|$ combined length of reads
- If $m'$ is the maximum length of any read, then the runtime of the trie algorithm is the $\mathcal{O}(m'n)$ for matching and $\mathcal{O}(|\mathrm{reads}|)$ for trie construction.

But ... The amount of memory we need for the trie is in the worst case proportional to the total length of the reads, which can be enormous: $\mathcal{O}(|reads|)$. Can we flip the paradigm and preprocess the genome?

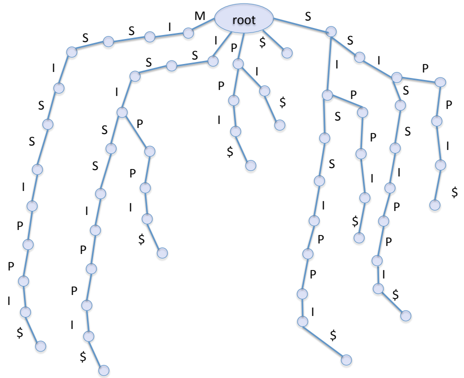
# Trie

In the previous example, we made a trie out of the "reads" and slid this trie across our "genome" to search for matches. Let us now examine another strategy that will take us to suffix trees and arrays.



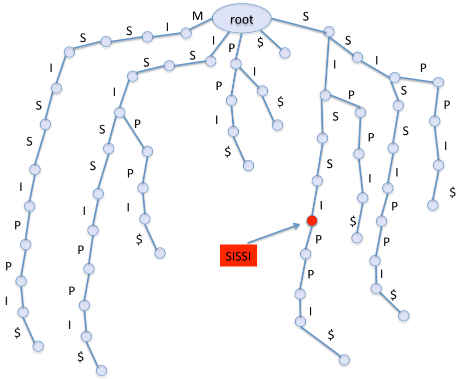
## Trie

T:MISSISSIPI T$: MISSISSIPI$



Each path from the root to a leaf represents a suffix, and each suffix is represented by a path from the root to a leaf.

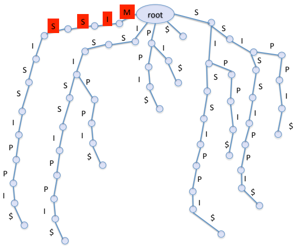
## Trie



- The nodes have implicit **labels** that reflect the string of characters on the path from the root to the node.

## Trie

- Each substring of T is represented by a path from the root, i.e., every T substring is **a prefix of some suffix of T**
- Thus to search for a substring, start at the root and follow the edges labeled with the characters of $S$
- If at some point there is no outgoing edge for the next character of $S$, then $S$ is *not* a substring of $T$

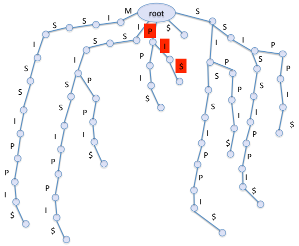


MISS is a substring of $T$

MIST is *not* a substring

## Trie

- A string $S$ is a **suffix** of $T$ if it is a substring and the final node on the walk has an outgoing edge labeled \$

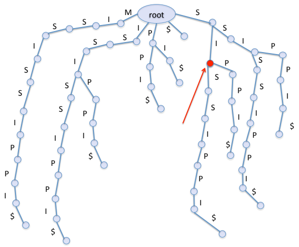


PI is a substring of $T$

PI is also a suffix of $T$

## Trie

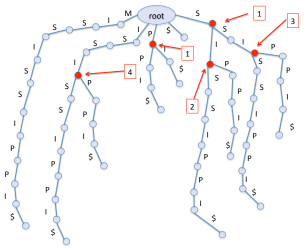
- How many times does some string $S$ occur in $T$?
- Follow the path for $S$
- If we finish at some node $n$, then $S$ occurs the same number of times as the number of leaf nodes in the subtree rooted at $n$



The substring SI occurs twice in $T$

## Trie

- What is the longest repeated substring $S$ of $T$?
- This is the deepest node with 2 or more children



ISSI is the longest repeated substring

## Constructing a Suffix Trie

The naive algorithm is pretty simple to implement

**Algorithm 1** Suffix Trie($T$)

```
 1: $T += \$$
 2: root = {}
 3: for i=1 to i=length($T$) do
 4:     n = root # n is the current node
 5:     for c in $T[i:]$ # for each char in the i-th suffix do
 6:         if $c \notin$ n then
 7:             n[c] = {} # add outgoing edge to n if needed
 8:         end if
 9:         n = n[c] # switch current node to child node
10:     end for
11: end for
12: return root
```

## Searching a Suffix Trie

`followPath` returns the node at the end of the path or NULL if there is no path.

**Algorithm 2** followPath($T$,$S$)

```
root = SuffixTrie($T$)
n = root  # n is the current node
for i=1 to i=length($S$) do
    c = S[i] # i-th char of S
    if $c \notin$ n then
        return NULL # not found
    end if
    n = n[c] # switch current node to child node
end for
return n
```

## Searching a Suffix Trie

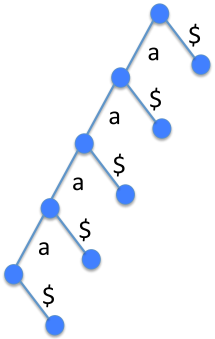
**Algorithm 3** HasSubstring($T$,$S$)

```
1: n = FollowPath($T$,$S$)
2: if $n \neq$ NULL then
3:     return TRUE
4: else
5:     return FALSE
6: end if
```

- `hasSubstring` basically checks if FollowPath does not "fall off" the tree and return NULL
- One could write a similar function `hasSuffix` that would check if the node returned by `followPath` is not NULL and is equal to $

## Suffix Trie: Size complexity

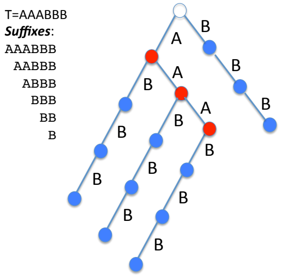
- We would like to know the limits for the size of a suffix trie
- How many nodes does a suffix trie have if the string it is based on has $m$ characters?
- Consider the string $T = aaaa\$$, i.e., $m$ a's in a row
- There is one root
- there are $m$ nodes with an incoming "a" edge
- there are $m+1$ nodes with an incoming "\$" edge
- Total 2m+2 nodes, i.e., $\mathcal{O}(m)$ nodes



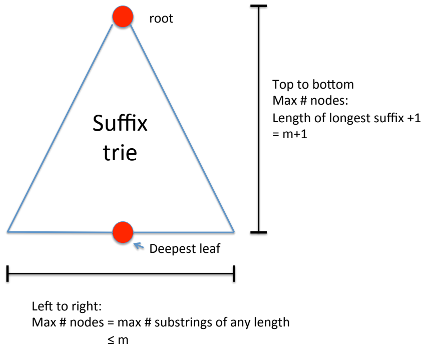
## Suffix Trie: Size complexity

- We would like to know the limits for the size of a suffix trie

- How many nodes does a suffix trie have if the string it is based on has $m$ characters?
- Consider the string $T = AAABBB\$$ with $n = 3$ A's, $n = 3$ B's and $m = 2n$
- There is one root
- there are $n$ nodes on the "b" chain (right)
- there are $n$ nodes on the "a" chain (middle)
- there are $n$ chains of $n$ "b" nodes (hanging from each "a" node)
- there are $2n + 1$ "\$" nodes (not shown here)
- Total $n^2 + 4n + 2$ nodes, i.e., $\mathcal{O}(n^2)$ nodes



## Suffix Trie: Size complexity



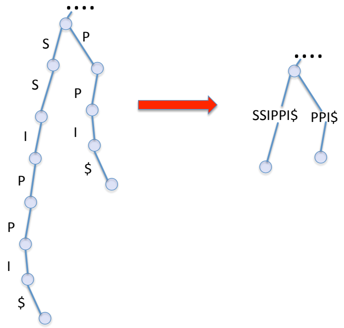
- Thus, we have seen two example string classes with size complexity (nmber of nodes) that grow at $\mathcal{O}(m)$ and $\mathcal{O}(m^2)$
- The figure shows that the worst case is $\mathcal{O}(m^2)$
- On real life data, the number of nodes grows more than linearly but less than quadratically – still usually too much to be practical

## Suffix Trie: Size complexity

The challenge of algorithmic development for NGS read aligners is basically to make string indices smaller and faster. We will go through various ideas that take us from the suffix trie to the suffix tree

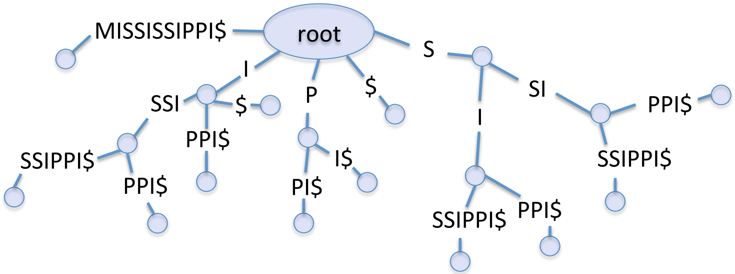
- Combine non-branching paths into a single edge with a string label
- Replace the string label with $\mathcal{O}(1)$ references to the original "genome" string
- $\mathcal{O}(n)$ "online" method for constructing suffix tree (Ukkonen)

## Suffix Trie to Suffix Tree



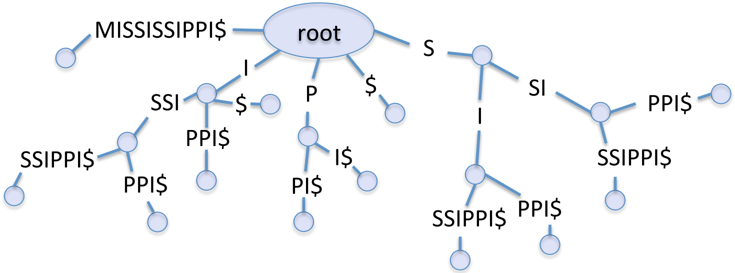
- Combine non-branching paths into a single edge with a string label
- This clearly reduces the number of nodes and edges
- As a side effect, it ensures that all internal nodes have more than one child node.

## Suffix Trie to Suffix Tree



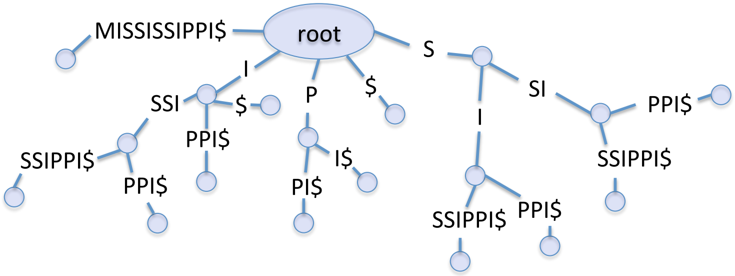
- Once non-branching paths are combined into single edges, *what is the effect on the number of leaves and internal nodes?*
- $T$=MISSISSIPPI\$
- $m = \text{length}(T) = 12$

## Suffix Trie to Suffix Tree



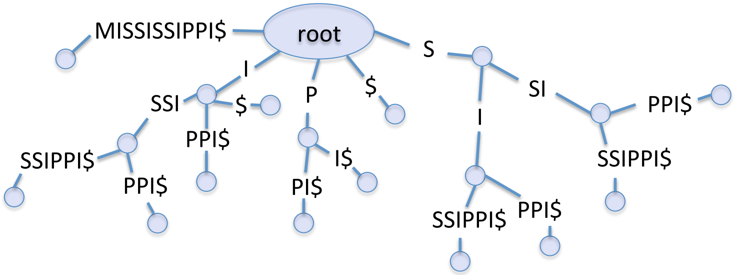
- There are $m$ leaf nodes (obvious, since we have $m$ suffixes)
- Recall that if a full binary tree has $m$ leaf nodes, it has exactly $m - 1$ internal nodes
- Our tree has at most as many internal nodes as a full binary tree (because an internal node of a suffix tree can have $> 2$ children)
- Thus, there are $\leq 2m - 1$ total nodes, i.e., $\mathcal{O}(m)$ nodes

## Suffix Trie to Suffix Tree



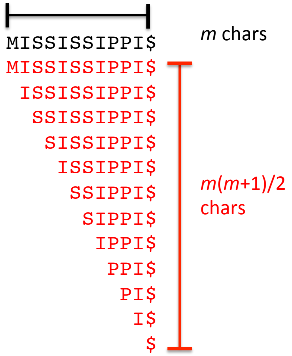
- Thus, the number of nodes is now linear in the size of the input
- BUT the total length of the edge labels is still $\mathcal{O}(m^2)$.
- To reduce the size complexity of the edges, we will simply store the offset and length of the original labels for each edge (two ints or two longs depending on the implementation, i.e., $\mathcal{O}(1)$).

## Suffix Trie to Suffix Tree



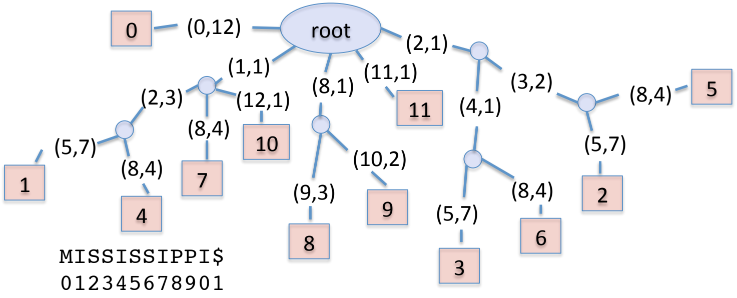
- Thus, the number of nodes is now linear in the size of the input
- BUT the total length of the edge labels is still $\mathcal{O}(m^2)$.

## Suffix Trie to Suffix Tree



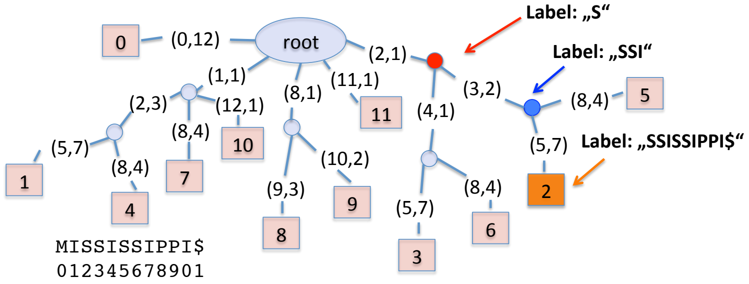
- If we store all of the suffixes in the edges, this results in $\mathcal{O}(m^2)$ space complexity.
- To reduce the size complexity of the edges, we will simply store the offset and length of the original labels for each edge (two ints or two longs depending on the implementation, i.e., $\mathcal{O}(1)$).

# Suffix Trie to Suffix Tree



- We can store the offsets in the leaves
- For example, the longest suffix has offset zero

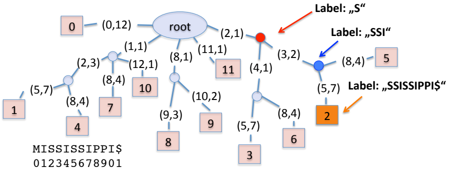
## Suffix Trie to Suffix Tree



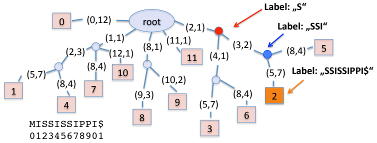
- The node label is the concatenated edge labels from the root to the node
- As mentioned, the labels are not stored explicitly

## Suffix Trie to Suffix Tree

- **Node depth**: Number of edges from the root to a given node
- **Label depth**: Total length of edge labels (characters) on a path from the root to a given node



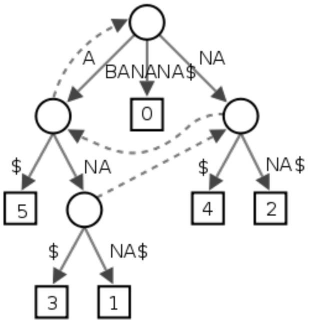
## How to build a Suffix Tree



- Naive method 1: First build a suffix trie and then convert it to a suffix trie by combining non-branching paths and relabeling the edges
- Naive method 2: Build a suffix tree one suffix at a time (add entire string, then the suffix starting at position 1,2,3,...)
- These methods that $\mathcal{O}(m^2)$ time
- Is there a difference in the space complexity of the two methods?

## How to build a Suffix Tree

One of the most elegant algorithms around is Ukkonen's linear time online suffix tree consruction algorithm. It is well described in Gusfield's book

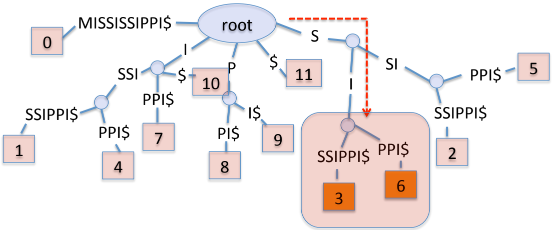


Ukkonen, E. (1995), On-line construction of suffix trees, Algorithmica 14 (3): 249260,

## How to build a Suffix Tree

- We will not go into the details of Ukkonen's algorithm here
- Memory and time for construction are linear, a substantial improvement over the suffix trie
- The basic search algorithms presented for the suffix trie work with corresponding modifications for the suffix tree

## Suffix Tree: Find all matches of P to T



```
MISSISSIPPI
   SI SI
```

*P*=SI    *T*=MISSISSIPPI$

```
Walk down path corresponding to P
Visit all leaf nodes in subtree (DFS)
```

- Let $k = \#$ of matches and $n$ be the length of the pattern
- The search is then $\mathcal{O}(n + k)$
- Note the subtree where we stop has $\mathcal{O}(k)$ nodes and DFS to enumerate these nodes is linear time

## Suffix Tree: Back to the Real World

- Although a linear algorithm, i.e., $\mathcal{O}(n)$ is desirable, the big-O notation tells us nothing about the constant factor
- The constant factor is relatively high for suffix trees
- Up to over 20 bytes per node for naive implementations
- Practical implementations reach about 12.5 bytes per node
- Can be relatively impractical for indexing say the human genome

## Outline

## Suffix Arrays

The suffix array, at least in its simplest incarnation, requires only 4 bytes per character of the input sequence. We will discuss some of the algorithms surrounding suffix arrays here in preparation for our treatment of BWT algorithms next time.

Notation

- We will refer to our string "MISSISSIPPI" as $S[1 \dots N]$ with $N = 12$
- a naive implementation of the suffix array basically manipulates an array of pointers to the suffixes $S[1 \dots N]$, $S[2 \dots N]$, ..., $S[N \dots N]$

## How to Build a Suffix Array (Naive)

```
T:   MISSISSIPPI$
0:   MISSISSIPPI$
1:   ISSISSIPPI$
2:   SSISSIPPI$
3:   SISSIPPI$
4:   ISSIPPI$
5:   SSIPPI$
6:   SIPPI$
7:   IPPI$
8:   PPI$
9:   PI$
10:  I$
11:  $
```

- Form all possible suffixes from the input string $T$=MISSISSIPPI$

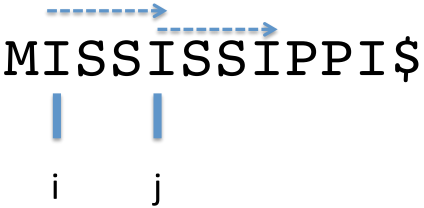
## How to Build a Suffix Array (Naive)

```
0:   MISSISSIPPI$                      11:  $
1:   ISSISSIPPI$                       10:  I$
2:   SSISSIPPI$                        7:   IPPI$
3:   SISSIPPI$                         4:   ISSIPPI$
4:   ISSIPPI$                          1:   ISSISSIPPI$
5:   SSIPPI$           sort            0:   MISSISSIPPI$
6:   SIPPI$        ──────────→         9:   PI$
7:   IPPI$                             8:   PPI$
8:   PPI$                              6:   SIPPI$
9:   PI$                               3:   SISSIPPI$
10:  I$                                5:   SSIPPI$
11:  $                                 2:   SSISSIPPI$
```

- Sort lexicographically (e.g., radix sort)
- This has the effect of bringing repeated substrings together
- This suggests a search algorithm to find all occurences
  1. suffix sort the text
  2. binary search for the query and scan until mismatch

## Suffix Array

Longest repeated substring problem



Consider the naive approach

- Try all indices $i$ and $j$ of a string with $m$ characters
- Compute the longest common prefix for each pair
- complexity $\mathcal{O}(Dm^2)$ where $D$ is the length of the longest match.

## Suffix Array

Longest repeated substring problem

```
0:  MISSISSIPPI$                      11: $
1:  ISSISSIPPI$                       10: I$
2:  SSISSIPPI$                        7:  IPPI$
3:  SISSIPPI$                         4:  ISSIPPI$
4:  ISSIPPI$             sort         1:  ISSISSIPPI$
5:  SSIPPI$           ---------->     0:  MISSISSIPPI$
6:  SIPPI$                            9:  PI$
7:  IPPI$                             8:  PPI$
8:  PPI$                              6:  SIPPI$
9:  PI$                               3:  SISSIPPI$
10: I$                                5:  SSIPPI$
11: $                                 2:  SSISSIPPI$
```

- Easy if we have a suffix array of the input string
- Scan through list to find neighbors with longest common prefix

## Suffix Array

- We have examined a naive method for constructing the suffix array until now
- There are a number of linear time suffix array construction algorithms
- We will instead present a simpler $\mathcal{O}(n \log n)$ algorithm due to Manber and Myers

## Suffix Array

### Manber Myers Algorithm

- Initialize: Sort on first character (using key-indexed counting sort)
- Phase $i$: Given an array of suffixes sorted on the first $2^{i-1}$ characters, create an array of suffixes sorted on the first $s^i$ characters
- Running time $\mathcal{O}(n \log n)$
- We can perform a single phase in linear time

# How to Build a Suffix Array (Manber/Myers)

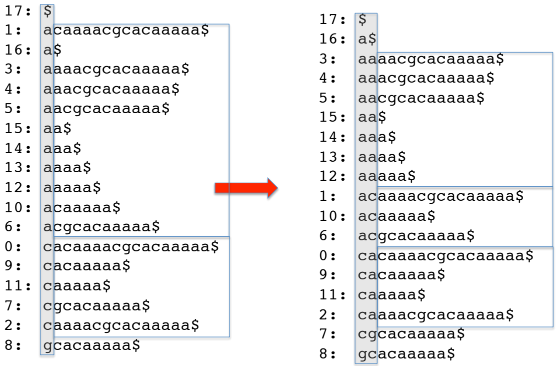
```
0:   cacaaaacgcacaaaaa$
1:   acaaaacgcacaaaaa$
2:   caaaacgcacaaaaa$
3:   aaaacgcacaaaaa$
4:   aaacgcacaaaaa$
5:   aacgcacaaaaa$
6:   acgcacaaaaa$
7:   cgcacaaaaa$
8:   gcacaaaaa$
9:   cacaaaaa$
10:  acaaaaa$
11:  caaaaa$
12:  aaaaa$
13:  aaaa$
14:  aaa$
15:  aa$
16:  a$
17:  $
```

$\rightarrow$

```
17:  $
1:   acaaaacgcacaaaaa$
16:  a$
3:   aaaacgcacaaaaa$
4:   aaacgcacaaaaa$
5:   aacgcacaaaaa$
15:  aa$
14:  aaa$
13:  aaaa$
12:  aaaaa$
10:  acaaaaa$
6:   acgcacaaaaa$
0:   cacaaaacgcacaaaaa$
9:   cacaaaaa$
11:  caaaaa$
7:   cgcacaaaaa$
2:   caaaacgcacaaaaa$
8:   gcacaaaaa$
```

Initialization: radix sort on first character

## How to Build a Suffix Array (Manber/Myers)

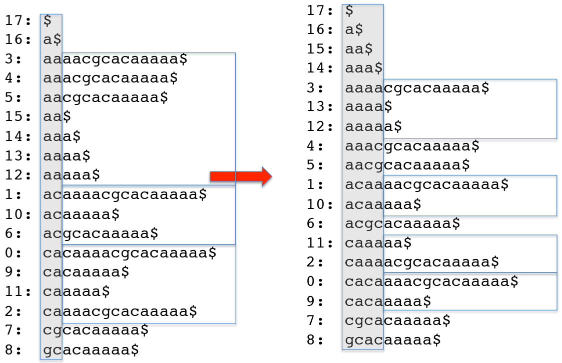


```
17: $
1:  acaaaacgcacaaaaa$
16: a$
3:  aaaacgcacaaaaa$
4:  aaacgcacaaaaa$
5:  aacgcacaaaaa$
15: aa$
14: aaa$
13: aaaa$
12: aaaaa$
10: acaaaaa$
6:  acgcacaaaaa$
0:  cacaaaacgcacaaaaa$
9:  cacaaaaa$
11: caaaaa$
7:  cgcacaaaaa$
2:  caaaacgcacaaaaa$
8:  gcacaaaaa$
```

```
17: $
16: a$
3:  aaaacgcacaaaaa$
4:  aaacgcacaaaaa$
5:  aacgcacaaaaa$
15: aa$
14: aaa$
13: aaaa$
12: aaaaa$
1:  acaaaacgcacaaaaa$
10: acaaaaa$
6:  acgcacaaaaa$
0:  cacaaaacgcacaaaaa$
9:  cacaaaaa$
11: caaaaa$
2:  caaaacgcacaaaaa$
7:  cgcacaaaaa$
8:  gcacaaaaa$
```

Step 1: radix sort on first two characters

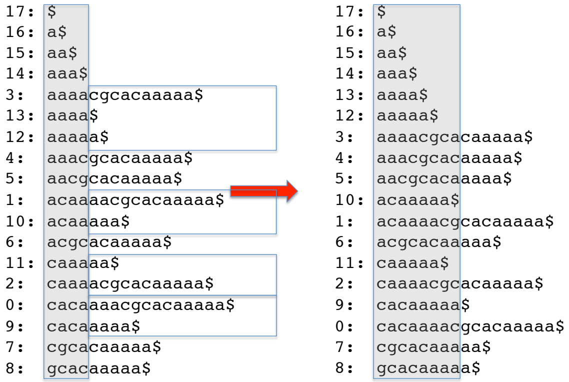
## How to Build a Suffix Array (Manber/Myers)



```
17: $                        17: $
16: a$                       16: a$
3:  aaaacgcacaaaaa$          15: aa$
4:  aaacgcacaaaaa$           14: aaa$
5:  aacgcacaaaaa$            3:  aaaacgcacaaaaa$
15: aa$                      13: aaaa$
14: aaa$                     12: aaaaa$
13: aaaa$                    4:  aaacgcacaaaaa$
12: aaaaa$                   5:  aacgcacaaaaa$
1:  acaaaacgcacaaaaa$        1:  acaaaacgcacaaaaa$
10: acaaaaa$                 10: acaaaaa$
6:  acgcacaaaaa$             6:  acgcacaaaaa$
0:  cacaaaacgcacaaaaa$       11: caaaaa$
9:  cacaaaaa$                2:  caaaacgcacaaaaa$
11: caaaaa$                  0:  cacaaaacgcacaaaaa$
2:  caaaacgcacaaaaa$         9:  cacaaaaa$
7:  cgcacaaaaa$              7:  cgcacaaaaa$
8:  gcacaaaaa$               8:  gcacaaaaa$
```

Step 2: radix sort on first four characters

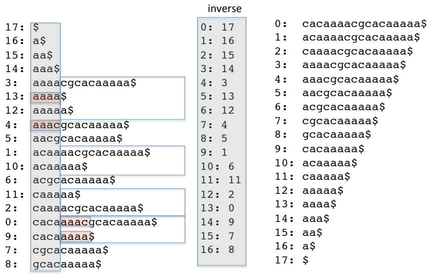
# How to Build a Suffix Array (Manber/Myers)



```
17: $                       17: $
16: a$                      16: a$
15: aa$                     15: aa$
14: aaa$                    14: aaa$
3:  aaaacgcacaaaaa$         13: aaaa$
13: aaaa$                   12: aaaaa$
12: aaaaa$                  3:  aaaacgcacaaaaa$
4:  aaacgcacaaaaa$          4:  aaacgcacaaaaa$
5:  aacgcacaaaaa$           5:  aacgcacaaaaa$
1:  acaaaacgcacaaaaa$       10: acaaaaa$
10: acaaaaa$                1:  acaaaacgcacaaaaa$
6:  acgcacaaaaa$            6:  acgcacaaaaa$
11: caaaaa$                 11: caaaaa$
2:  caaaacgcacaaaaa$        2:  caaaacgcacaaaaa$
0:  cacaaaacgcacaaaaa$      9:  cacaaaaa$
9:  cacaaaaa$               0:  cacaaaacgcacaaaaa$
7:  cgcacaaaaa$             7:  cgcacaaaaa$
8:  gcacaaaaa$              8:  gcacaaaaa$
```

Step 3: radix sort on first eight characters

# How to Build a Suffix Array (Manber/Myers)



```
17: $
16: a$
15: aa$
14: aaa$
3:  aaaacgcacaaaaa$
13: aaaa$
12: aaaaa$
4:  aaacgcacaaaaa$
5:  aacgcacaaaaa$
1:  acaaaacgcacaaaaa$
10: acaaaaa$
6:  acgcacaaaaa$
11: caaaaa$
2:  caaaacgcacaaaaa$
0:  cacaaaacgcacaaaaa$
9:  cacaaaaa$
7:  cgcacaaaaa$
8:  gcacaaaaa$
```

inverse

```
0: 17
1: 16
2: 15
3: 14
4: 3
5: 13
6: 12
7: 4
8: 5
9: 1
10: 6
11: 11
12: 2
13: 0
14: 9
15: 7
16: 8
```

```
0:  cacaaaacgcacaaaaa$
1:  acaaaacgcacaaaaa$
2:  caaaacgcacaaaaa$
3:  aaaacgcacaaaaa$
4:  aaacgcacaaaaa$
5:  aacgcacaaaaa$
6:  acgcacaaaaa$
7:  cgcacaaaaa$
8:  gcacaaaaa$
9:  cacaaaaa$
10: acaaaaa$
11: caaaaa$
12: aaaaa$
13: aaaa$
14: aaa$
15: aa$
16: a$
17: $
```

- To sort by 8-mers we can reuse information we have from sorting two mers
- To sort the suffixes 0 and 9 (cacaaaac and cacaaaaa), we can reuse information.
- We now that the first four chars are sorted and only need to care about the last four chars. But these were sorted previously!
- To get the index of the second four chars of suffix 0, we look at the suffix at 0+4=4 (rank 7 in sorted list), and for suffix 9, we look at 9+4=13 (rank 5 in sorted list)

## Finally

- Email: peter.robinson@charite.de
- Office hours by appointment

### Further reading

- Shrestha AM et al (2014)A bioinformatician's guide to the forefront of suffix array construction algorithms. *Brief Bioinform* 15(2):138-54.