String Algorithms and Data Structures

Suffix Tries (and Trees)

CS 199-225
Brad Solomon

October 10, 2022
Exact pattern matching with indexing

Preprocess (index) $\approx O(|T|)$

Search Index $\approx O(|P|)$

Find instances of $P$ in $T$
String indexing with Tries

**Trie**: A rooted tree storing a collection of (key, value) pairs

Keys:    Values:
  instant   1  
  internal  2  
  internet  3  

Each edge is labeled with a character $c \in \Sigma$

For given node, at most one child edge has label $c$, for any $c \in \Sigma$

Each key is “spelled out” along some path starting at root and each value is stored at the leaf
NaryTree build_trie(std::string T)

$T: C \, G \, T \, G \, C$

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We are storing $\frac{|T|(|T| + 1)}{2}$ values

We had to do $\frac{|T|(|T| + 1)}{2}$ insertions

root: $G$
std::vector<int> searchPattern(P)

\(T: C G T G C\)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We can do exact pattern matching in \(O(P)\) time!
Exact pattern matching with indexing

How can we be more efficient in our preprocessing?

1) Perform fewer insertions to store T

2) Store fewer values in index

What if we just stored the suffixes?
Exact pattern matching w/ indexing

There are many data structures built on suffixes

Modern methods still use these today
Suffix Trie

Build a trie containing all suffixes of a text $T$

$T$: C G T G C

C G T G C
G T G C
T G C
G C
C

$T$ suffixes
Suffix Trie

Build a trie containing all suffixes of a text $T$

$T$: C G T G C
   C G T G C
   G T G C
   T G C
   G C
   C

Inserting just $T$ suffixes gets us the same tree*!
Suffix Trie

To prevent loss of information, add a **terminal character**

$T$: C G T G C $

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGTGC$</td>
<td>0</td>
</tr>
<tr>
<td>GTGC$</td>
<td>1</td>
</tr>
<tr>
<td>TGC$</td>
<td>2</td>
</tr>
<tr>
<td>GC$</td>
<td>3</td>
</tr>
<tr>
<td>C$</td>
<td>4</td>
</tr>
<tr>
<td>$</td>
<td>5</td>
</tr>
</tbody>
</table>

A terminal character cannot be a part of the alphabet!
Suffix Trie

To prevent loss of information, add a **terminal character**

\[ T: C G T G C \]$\]

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGTGC$</td>
<td>0</td>
</tr>
<tr>
<td>GTGC$</td>
<td>1</td>
</tr>
<tr>
<td>TGC$</td>
<td>2</td>
</tr>
<tr>
<td>GC$</td>
<td>3</td>
</tr>
<tr>
<td>C$</td>
<td>4</td>
</tr>
<tr>
<td>$</td>
<td>5</td>
</tr>
</tbody>
</table>
Suffix Trie Construction

$T: a \ b \ a \ \$$

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>aba$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

Where do I put value?
Suffix Trie Construction

\( T: a \ b \ a \$ \)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>aba$</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

What are my other keys?
Suffix Trie Construction

$T$: \texttt{a b a}$

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>aba$</td>
<td>0</td>
</tr>
<tr>
<td>ba$</td>
<td>1</td>
</tr>
<tr>
<td>a$</td>
<td>2</td>
</tr>
</tbody>
</table>

What edges do I add?
Suffix Trie Construction

\[ T: \text{a b a }\$ \]

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>aba$</td>
<td>0</td>
</tr>
<tr>
<td>ba$</td>
<td>1</td>
</tr>
<tr>
<td>a$</td>
<td>2</td>
</tr>
</tbody>
</table>

Are we done?
Suffix Trie Construction

$T$: \text{a b a $}$

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>aba$</td>
<td>0</td>
</tr>
<tr>
<td>ba$</td>
<td>1</td>
</tr>
<tr>
<td>a$</td>
<td>2</td>
</tr>
<tr>
<td>$</td>
<td>3</td>
</tr>
</tbody>
</table>
class NaryTree {

public:
    struct Node {
        int index;
        std::map<std::string, Node*> children;

        Node(std::string s, int i)
        {
            if(s.length() > 0 ){
                std::string f = s.substr(0,1);
                children[f] = new Node(s.substr(1), i );
            } else {
                index = i;
            }
        }
    }

protected:
    Node* root;
    ...
};
NaryTree build_strie(std::string T)

\[ T: C G T G C $ \]

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGTGC$</td>
<td>0</td>
</tr>
<tr>
<td>GTGC$</td>
<td>1</td>
</tr>
<tr>
<td>TGC$</td>
<td>2</td>
</tr>
<tr>
<td>GC$</td>
<td>3</td>
</tr>
<tr>
<td>C$</td>
<td>4</td>
</tr>
<tr>
<td>$</td>
<td>5</td>
</tr>
</tbody>
</table>

Store \(|T| + 1\) values

Perform \(|T| + 1\) insertions
Assignment 6: a_stree

Learning Objective:

Use an existing implementation of a suffix trie as a N-ary Tree

Implement exact pattern matching using a suffix trie

Construct a suffix tree from a suffix trie
Suffix Trie Search

Each of $T$'s substrings is spelled out along a path from the root.

Every substring is a prefix of some suffix

$P = baa$

$Search(P) =$

$P = ab$

$Search(P) =$
Suffix Trie Search

Each of $T$'s substrings is spelled out along a path from the root.

Every **substring** is a **prefix** of some **suffix**

Starting at root:

0) If $P$ empty, return values of all leaves.
1) Try to match front character
2) If match, move to appropriate child
   2.5) Set pattern equal to remainder
   2.5) Go back to (0)
3) If mismatch, $P$ is not a substring!

$T = \text{abaaba}$

$P = \text{abaa}$

Yes, it's a substring

Return \{0\}
Suffix Trie Search

Each of $T$'s substrings is spelled out along a path from the root.

Every substring is a prefix of some suffix

Starting at root:

(0) If $P$ empty, return values of all leaves.
(1) Try to match front character
(2) If match, move to appropriate child
   (2.5) Set pattern equal to remainder
   (2.5) Go back to (0)
(3) If mismatch, $P$ is not a substring!

$T = \text{abaaba}$
$P = \text{baabb}$
Suffix Trie Search

Each of $T$'s substrings is spelled out along a path from the root.

Every *substring* is a *prefix* of some *suffix*

Starting at root:

(0) If $P$ empty, return values of all leaves.
(1) Try to match front character
(2) If match, move to appropriate child
   (2.5) Set pattern equal to remainder
   (2.5) Go back to (0)
(3) If mismatch, $P$ is not a substring!

$T = \text{abaaba}$
$P = \text{baabb}$

Yes, it's a substring
Return $\{0,2,3,5\}$
Suffix Trie Search

Each of $T$'s substrings is spelled out along a path from the root.

Every substring is a prefix of some suffix.

Starting at root:

(0) If $P$ empty, return values of all leaves.

Trie Search Big O:

Suffix Trie Search Big O:
Suffix Trie

How does the suffix trie grow with $|T| = m$?

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

Yes: a string of $m$ a’s in a row ($a^m$)
Suffix Trie

How does the suffix trie grow with $|T| = m$?

Is there a class of string where the number of suffix trie nodes grows with $m^2$?

Yes: $a^n b^n$ where $2n = m$

- 1 root
- $n$ nodes along “b chain,” right
- $n$ nodes along “a chain,” middle
- $n$ chains of $n$ “b” nodes hanging off “a chain” ($n^2$ total)
- $2n + 1$ $\$$ leaves (not shown)

$n^2 + 4n + 2$ nodes, where $m = 2n$
Suffix trie: actual growth

Built suffix tries for the first 500 prefixes of a virus genome

Black curve shows how # nodes increases with prefix length

Figure & example by Ben Langmead
Suffix trie: actual growth

Built suffix tries for the first 500 prefixes of a virus genome

Black curve shows how # nodes increases with prefix length

Actual growth much closer to worst case than to best!

Figure & example by Ben Langmead
Assignment 6: a_stree

Learning Objective:

Use an existing implementation of a suffix trie as a N-ary Tree

Implement exact pattern matching using a suffix trie

Construct a suffix tree from a suffix trie
Suffix Trie: Making it smaller

$I = \text{abaaba}$

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

Reduces # nodes, edges, guarantees non-leaf nodes have >1 child
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 non-leaf nodes have >1 child
Coalescing edges

We want to coalesce paths that don’t branch.
Coalescing edges

We want to coalesce paths that don’t branch.

‘Current root’ in blue

Coalesce ‘$’?  No, nothing to merge

Coalesce ‘a’?  No, child has a branch

Coalesce ‘b’?  Yes, b->a is the only path.
Coalescing edges

We want to coalesce paths that don’t branch.

‘Current root’ in blue

Coalesce ‘$’?  **No**, nothing to merge

Coalesce ‘a’?  **No**, child has a branch

Coalesce ‘b’?  **Yes**, b->a is the only path.
Coalescing edges

We want to coalesce paths that don’t branch.

‘Current root’ in blue

Coalesce ‘$’? **No**, nothing to merge

Coalesce ‘a’? **No**, child has a branch

Coalesce ‘b’? **Yes**, b->a is the only path.

Are we done?
Coalescing edges

We want to coalesce paths that don’t branch.

‘Current root’ in blue

We added a new edge ‘ba’!

We might need to coalesce again!
Coalescing edges

We want to coalesce paths that don’t branch.

‘Current root’ in blue

We added a new edge ‘ba’!

We might need to coalesce again!

Repeat until **all** edges have been checked
Coalescing Edges

Coalesce all paths that don’t branch.

A suffix tree of $|T| = m$ should have:

- $m$ leaves* (including ‘$’ in T)
- $\leq m - 1$ internal nodes
- Each internal node $\geq 2$ children

How to do this is up to you (and for you to work out)!
Iterator on Dynamic Data

```cpp
map<string,Node*>::iterator it = myMap.begin();

while(it != myMap.end()){
    Node* myChild = it->second;
    if < LOGIC STATEMENT >{
        Node* temp = < myChild's child >;
        myMap["NewEdge"] = temp;
        delete myChild;
        myMap.erase(it++);
    }
}
```

```cpp
b
```

```cpp
myChild
```

```cpp
myChild's child
```
map<string,Node*>::iterator it = myMap.begin();
while(it != myMap.end()){
    Node* myChild = it->second;
    if < LOGIC STATEMENT >{
        Node* temp = < myChild's child >;
        myMap["NewEdge"] = temp;
        delete myChild;
        myMap.erase(it++);
    }
}
map<string, Node*>::iterator it = myMap.begin();
while(it != myMap.end()){
    Node* myChild = it->second;
    if < LOGIC STATEMENT >{
        Node* temp = < myChild's child >;
        myMap["NewEdge"] = temp;
        delete myChild;
        myMap.erase(it++);
    }
}
Iterator on Dynamic Data

```cpp
map<string, Node*>::iterator it = myMap.begin();
while (it != myMap.end()) {
    Node* myChild = it->second;
    if (< LOGIC STATEMENT >) {
        Node* temp = < myChild's child >;
        myMap["NewEdge"] = temp;
        delete myChild;
        myMap.erase(it++);
    }
}
```

iterator points at 'next'

myChild's child
map<string,Node*>::iterator it = myMap.begin();
while(it != myMap.end()){
    Node* myChild = it->second;
    if < LOGIC STATEMENT >{
        Node* temp = < myChild's child >;
        myMap[“NewEdge”] = temp;
        delete myChild;
        it = myMap.erase(it);
    }
}
Assignment 6: a_stree

Learning Objective:

Use an existing implementation of a suffix trie as a N-ary Tree

Implement exact pattern matching using a suffix trie

Construct a suffix tree from a suffix trie

Consider: The modified NaryTree code works for both suffix tries and trees. Can you write a search that works for both trees and tries?
Suffix tree: building

**Method 1:** build suffix trie, coalesce non-branching paths, relabel edges

\[ O(m^2) \text{ time, } O(m^2) \text{ space} \]

**Method 2:** build single-edge tree representing longest suffix, augment to include the 2\textsuperscript{nd}-longest, augment to include 3\textsuperscript{rd}-longest, etc (Gusfield 5.4)

\[ O(m^2) \text{ time, } O(m) \text{ space} \]
On-Line Construction of Suffix Trees

E. Ukkonen

Abstract. An on-line algorithm is presented for constructing the suffix tree for a given string in time linear in the length of the string. The new algorithm has the desirable property of processing the string symbol by symbol from left to right. It always has the suffix tree for the scanned part of the string ready. The method is developed as a linear-time version of a very simple algorithm for (quadratic size) suffix tries. Regardless of its quadratic worst case this latter algorithm can be a good practical method when the string is not too long. Another variation of this method is shown to give, in a natural way, the well-known algorithms for constructing suffix automata (DAWGs).

Key Words. Linear-time algorithm, Suffix tree, Suffix trie, Suffix automaton, DAWG.

Canonical algorithm for $O(m)$ time & space suffix tree construction

Won’t cover it in class; see Gusfield Ch. 6 for details
Suffix trie

>100K nodes

Length prefix over which suffix trie was built

# suffix trie nodes

m(m+1)/2
actual
2m+2

Suffix tree

<1K nodes

Length prefix over which suffix tree was built

# suffix tree nodes

2m
actual
m
Suffix Tree

A rooted tree storing a collection of suffixes as (key, value) pairs

- Each key is “spelled out” along some path starting at root
- Each edge is labeled with a string \( s \)
- For given node, at most one child edge starts with character \( c \), for any \( c \in \Sigma \)
- Each internal node contains >1 children
- Each key’s value is stored at a leaf
Bonus Slides
Suffix Tree: Size Redux

\[ T = abaaba$ \quad |T| = m \]

# leaves?

# non-leaf nodes (upper-bound)?
Trie or tree: we contain all suffixes of a text $|T| = m$

$T$: 

\[
\begin{array}{l}
GTTATAGCTGATCGCGGCGTAGCGG$

GTTATAGCTGATCGCGGCGTAGCGG$

TTATAGCTGATCGCGGCGTAGCGG$

TATAGCTGATCGCGGCGTAGCGG$

ATAGCTGATCGCGGCGTAGCGG$

TAGCTGATCGCGGCGTAGCGG$

AGCTGATCGCGGCGTAGCGG$

GCTGATCGCGGCGTAGCGG$

CTGATCGCGGCGTAGCGG$

TGATCGCGGCGTAGCGG$

GATCGCGGCGTAGCGG$

ATCGCGGCGTAGCGG$

TCGCGGCGTAGCGG$

CGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG$

GGCGGCGTAGCGG

\end{array}
\]

$m(m+1)/2$ chars
Suffix Tree: Size Redux

Store $T$ itself in addition to the tree. Convert tree’s edge labels to (index, length) pairs with respect to $T$.

Space is now $O(m)$  Suffix trie was $O(m^2)$!
Suffix Tree: Size Redux

$T = \text{abaaba}$

$T(0,1) = \text{a}$

$T(3,4) = \text{aba}$

Label = “aaba$”

$T(0,1) =$ “a”

$T(3,4) =$ “aba$”

Label = “aaba$”
Suffix Tree: Size Redux

$T = \text{abaaba}$

$T = \text{abaaba}$

Index 2

Label = “abaaba$”