Suffix Trees

02-714
Slides by Carl Kingsford
Preprocessing Strings

- Over the next few lectures, we’ll see several methods for preprocessing string data into data structures that make many questions (like searching) easy to answer:
  - Suffix Tries
  - Suffix Trees
  - Suffix Arrays

- Typical setting: A long, known, and fixed text string (like a genome) and many unknown, changing query strings.
  - Allowed to preprocess the text string once in anticipation of the future unknown queries.
- Data structures will be useful in other settings as well.
Suffix Tries

• A trie, pronounced “try”, is a tree that exploits some structure in the keys
  - e.g. if the keys are strings, a binary search tree would compare the entire strings, but a trie would look at their individual characters

- Suffix trie are a space-inefficient data structure to store a string that allows many kinds of queries to be answered quickly.

- Suffix trees are hugely important for searching large sequences like genomes. Eg. the basis for a tool called “MUMMer”.
Suffix Tries

$s = abaaba$

$\text{SufTrie}(s) = \text{suffix trie representing string } s.$

Edges of the suffix trie are labeled with letters from the alphabet $\sum$ (say \{A,C,G,T\}).

Every path from the root to a solid node represents a suffix of $s$.

Every suffix of $s$ is represented by some path from the root to a solid node.

Why are all the solid nodes leaves?
How many leaves will there be?
Processing Strings Using Suffix Tries

Given a suffix trie $T$, and a string $q$, how can we:

• determine whether $q$ is a substring of $T$?
• check whether $q$ is a suffix of $T$?
• count how many times $q$ appears in $T$?
• find the longest repeat in $T$?
• find the longest common substring of $T$ and $q$?

Main idea:

every substring of $s$ is a prefix of some suffix of $s$. 
Searching Suffix Tries

s = abaaba$

Is “baa” a substring of s?

Follow the path given by the query string.

After we’ve built the suffix trees, queries can be answered in time: $O(|query|)$ regardless of the text size.
Searching Suffix Tries

$s = \text{abaaba}\$

Is “baa” a substring of $s$?

Follow the path given by the query string.

After we’ve built the suffix trees, queries can be answered in time:

$O(|\text{query}|)$

regardless of the text size.
Applications of Suffix Tries (1)

Check whether $q$ is a **substring** of $T$:

Check whether $q$ is a **suffix** of $T$:

Count # of occurrences of $q$ in $T$:

Find the longest repeat in $T$:

Find the lexicographically (alphabetically) first suffix:
Applications of Suffix Tries (1)

Check whether q is a substring of T:
   Follow the path for q starting from the root.
   If you exhaust the query string, then q is in T.

Check whether q is a suffix of T:

Count # of occurrences of q in T:

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:
Applications of Suffix Tries (1)

Check whether q is a **substring** of T:
Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:
Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Find the longest repeat in T:

Find the lexicographically (alphabetically) first suffix:
Applications of Suffix Tries (1)

Check whether $q$ is a **substring** of $T$:
- Follow the path for $q$ starting from the root.
- If you exhaust the query string, then $q$ is in $T$.

Check whether $q$ is a **suffix** of $T$:
- Follow the path for $q$ starting from the root.
- If you end at a leaf at the end of $q$, then $q$ is a suffix of $T$.

Count # of occurrences of $q$ in $T$:
- Follow the path for $q$ starting from the root.
- The number of leaves under the node you end up in is the number of occurrences of $q$.

Find the longest repeat in $T$:

Find the lexicographically (alphabetically) first suffix:
Applications of Suffix Tries (1)

Check whether q is a **substring** of T:
Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:
Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T.

Count # of occurrences of q in T:
Follow the path for q starting from the root.
The number of leaves under the node you end up in is the number of occurrences of q.

Find the longest repeat in T:
Find the deepest node that has at least 2 leaves under it.

Find the lexicographically (alphabetically) first suffix:
Applications of Suffix Tries (1)

Check whether \( q \) is a **substring** of \( T \):
- Follow the path for \( q \) starting from the root.
- If you exhaust the query string, then \( q \) is in \( T \).

Check whether \( q \) is a **suffix** of \( T \):
- Follow the path for \( q \) starting from the root.
- If you end at a leaf at the end of \( q \), then \( q \) is a suffix of \( T \).

Count # of occurrences of \( q \) in \( T \):
- Follow the path for \( q \) starting from the root.
- The number of leaves under the node you end up in is the number of occurrences of \( q \).

Find the longest repeat in \( T \):
- Find the deepest node that has at least 2 leaves under it.

Find the lexicographically (alphabetically) first suffix:
- Start at the root, and follow the edge labeled with the lexicographically (alphabetically) smallest letter.
Suffix Links

- Suffix links connect node representing “xα” to a node representing “α”.
- Most important suffix links are the ones connecting suffixes of the full string (shown at right).
- But every node has a suffix link.
  - Why?
  - How do we know a node representing α exists for every node representing xα?

s = abaaba$
A node represents the *prefix* of some suffix:

\[ \text{abaaba$} \]

The node’s suffix link should link to the *prefix* of the suffix $s$ that is 1 character shorter.

Since the suffix trie contains all suffixes, it contains a path representing $s$, and therefore contains a node representing every prefix of $s$. 
s = abaaba$

A node represents the prefix of some suffix:

The node’s suffix link should link to the prefix of the suffix $s$ that is 1 character shorter.

Since the suffix trie contains all suffixes, it contains a path representing $s$, and therefore contains a node representing every prefix of $s$. 
Find the longest common substring of $T$ and $q$:

$T = \text{abaaba}\$

$q = \text{bbaa}$
Applications of Suffix Tries (II)

Find the longest common substring of $T$ and $q$:

Walk down the tree following $q$.
If you hit a dead end, save the current depth, and follow the suffix link from the current node.
When you exhaust $q$, return the longest substring found.

$$T = abaaba$$
$$q = bbba$$
Constructing Suffix Tries
Suppose we want to build suffix trie for string:

\[ s = \text{abbacabaa} \]

We will walk down the string from left to right:

\[ \text{abba} \text{cabaa} \]

building suffix tries for \( s[0] \), \( s[0..1] \), \( s[0..2] \), ..., \( s[0..n] \)

To build suffix trie for \( s[0..i] \), we will use the suffix trie for \( s[0..i-1] \) built in previous step

To convert \( \text{SufTrie}(s[0..i-1]) \rightarrow \text{SufTrie}(s[0..i]) \), add character \( s[i] \) to all the suffixes:

\[ \text{abba} \text{cabaa} \]

\[ i=4 \]

Need to add nodes for the suffixes:

\[ \text{abba} \text{cac} \text{abac} \text{bac} \text{ac} \text{c} \]

Purple are suffixes that will exist in \( \text{SufTrie}(s[0..i-1]) \) Why?

How can we find these suffixes quickly?
Suppose we want to build suffix trie for string:

\[ s = \text{abbacabaa} \]

We will walk down the string from left to right:

\[ \text{abbacabaa} \]

building suffix tries for \( s[0] \), \( s[0..1] \), \( s[0..2] \), ..., \( s[0..n] \)

To build suffix trie for \( s[0..i] \), we will use the suffix trie for \( s[0..i-1] \) built in previous step

To convert \( \text{SufTrie}(s[0..i-1]) \) \( \rightarrow \) \( \text{SufTrie}(s[0..i]) \), add character \( s[i] \) to all the suffixes:

\[ \text{abba} \text{cabaa} \]

\( i=4 \)

Need to add nodes for the suffixes:

Purple are suffixes that will exist in \( \text{SufTrie}(s[0..i-1]) \) Why?

How can we find these suffixes quickly?
Need to add nodes for the suffixes:

Purple are suffixes that will exist in SufTrie(s[0..i-1]) Why?

How can we find these suffixes quickly?

Where is the new deepest node? (aka longest suffix)

How do we add the suffix links for the new nodes?
i=4

Need to add nodes for the suffixes:

Purple are suffixes that will exist in SufTrie(s[0..i-1]) Why?

How can we find these suffixes quickly?

Where is the new deepest node? (aka longest suffix)

How do we add the suffix links for the new nodes?
To build SufTrie(s[0..i]) from SufTrie(s[0..i-1]):

CurrentSuffix = longest (aka deepest suffix)

Repeat:
   Add child labeled s[i] to CurrentSuffix.
   Follow suffix link to set CurrentSuffix to next shortest suffix.

Add suffix links connecting nodes you just added in the order in which you added them.

until you reach the root or the current node already has an edge labeled s[i] leaving it.

Because if you already have a node for suffix αs[i] then you have a node for every smaller suffix.

In practice, you add these links as you go along, rather than at the end.
class SuffixNode:
    def __init__(self, suffix_link = None):
        self.children = {}
        if suffix_link is not None:
            self.suffix_link = suffix_link
        else:
            self.suffix_link = self
    def add_link(self, c, v):
        """link this node to node v via string c""
        self.children[c] = v

def build_suffix_trie(s):
    """Construct a suffix trie.""
    assert len(s) > 0

    # explicitly build the two-node suffix tree
    Root = SuffixNode()
    # the root node
    Longest = SuffixNode(suffix_link = Root)
    Root.add_link(s[0], Longest)

    # for every character left in the string
    for c in s[1:]:
        Current = Longest; Previous = None
        while c not in Current.children:
            # create new node r1 with transition Current -c->r1
            r1 = SuffixNode()
            Current.add_link(c, r1)

            # if we came from some previous node, make that
            # node’s suffix link point here
            if Previous is not None:
                Previous.suffix_link = r1

            # walk down the suffix links
            Previous = r1
            Current = Current.suffix_link

            # make the last suffix link
        if Current is Root:
            Previous.suffix_link = Root
        else:
            Previous.suffix_link = Current.children[c]

        # move to the newly added child of the longest path
        # (which is the new longest path)
        Longest = Longest.children[c]
    return Root
boundary path
Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)
Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)
Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)
Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it)
Note: there's already a path for suffix "a", so we don't change it (we just add a suffix link to it).
How many nodes can a suffix trie have?

s = a^n b^n will have
- 1 root node
- n nodes in a path of “b”s
- n paths of n+1 “b” nodes

Total = n(n+1)+n+1 = O(n^2) nodes.

This is not very efficient.

How could you make it smaller?
So... we have to “trie” again...

Space-Efficient Suffix Trees
A More Compact Representation

- Compress paths where there are no choices.

- Represent sequence along the path using a range $[i,j]$ that refers to the input string $s$. 

$s = \text{abaaba}\$

1234567
Space usage:

- In the compressed representation:
  - \# leaves = \(O(n)\) [one leaf for each position in the string]
  - Every internal node is at least a binary split.
  - Each edge uses \(O(1)\) space.

- Therefore, \# number of internal nodes is about equal to the number of leaves.

- And \# of edges \(\approx\) number of leaves, and space per edge is \(O(1)\).

- Hence, linear space.
• The same idea as with the suffix trie algorithm.

• Main difference: not every trie node is explicitly represented in the tree.

• Solution: represent trie nodes as pairs \((u, \alpha)\), where \(u\) is a real node in the tree and \(\alpha\) is some string leaving it.

• Some additional tricks to get to \(O(n)\) time. (We’ll talk about these later.)
Storing more than one string with **Generalized Suffix Trees**
Constructing Generalized Suffix Trees

**Goal.** Represent a set of strings \( P = \{s_1, s_2, s_3, \ldots, s_m\} \).

**Example.** att, tag, gat

Simple solution:
1. build suffix tree for string \( \text{aat#}_1 \text{tag#}_2 \text{gat#}_3 \)
Constructing Generalized Suffix Trees

**Goal.** Represent a set of strings \( P = \{s_1, s_2, s_3, \ldots, s_m\} \).

**Example.** att, tag, gat

Simple solution:
1. build suffix tree for string \( \text{aat}\text{#}_1\text{tag}\text{#}_2\text{gat}\text{#}_3 \)

(2) For every leaf node, remove any text after the first # symbol.
Applications of Generalized Suffix Trees

Longest common substring of $S$ and $T$:

Determine the strings in a database $\{S_1, S_2, S_3, \ldots, S_m\}$ that contain query string $q$: 
Applications of Generalized Suffix Trees

Longest common substring of $S$ and $T$:

- Build generalized suffix tree for $\{S, T\}$
- Find the deepest node that has descendants from both strings (containing both $#1$ and $#2$)

Determine the strings in a database $\{S_1, S_2, S_3, \ldots, S_m\}$ that contain query string $q$: 
Applications of Generalized Suffix Trees

Longest common substring of S and T:

Build generalized suffix tree for \{S, T\}
Find the deepest node that has descendants from both strings (containing both \#1 and \#2)

Determine the strings in a database \{S_1, S_2, S_3, ..., S_m\} that contain query string q:

Build generalized suffix tree for \{S_1, S_2, S_3, ..., S_m\}
Follow the path for q in the suffix tree.
Suppose you end at node u: traverse the tree below u, and output i if you find a string containing \#i.
Longest Common Extension

Longest common extension: We are given strings S and T. In the future, many pairs (i,j) will be provided as queries, and we want to quickly find:

the longest substring of S starting at i that matches a substring of T starting at j.

Build generalized suffix tree for S and T.

Preprocess tree so that lowest common ancestors (LCA) can be found in constant time.

Create an array mapping suffix numbers to leaf nodes.

Given query (i,j):
Find the leaf nodes for i and j
Return string of LCA for i and j
Longest Common Extension

Longest common extension: We are given strings S and T. In the future, many pairs \((i,j)\) will be provided as queries, and we want to quickly find:

the longest substring of S starting at \(i\) that matches a substring of T starting at \(j\).

Build generalized suffix tree for S and T. \(O(|S| + |T|)\)

Preprocess tree so that lowest common ancestors (LCA) can be found in constant time. \(O(|S| + |T|)\)

Create an array mapping suffix numbers to leaf nodes. \(O(|S| + |T|)\)

Given query \((i,j)\):
Find the leaf nodes for \(i\) and \(j\) \(O(1)\)
Return string of LCA for \(i\) and \(j\) \(O(1)\)
Using LCE to Find Palindromes

Maximal even palindrome at position $i$: the longest string to the left and right so that the left half is equal to the reverse of the right half.

Goal: find all maximal palindromes in $S$. 

Diagram: 

- $S$ 
- $x$ 
- $y$ 
- $i$ 
- $x \neq y$ 
- Red = the reverse of Green
Using LCE to Find Palindromes

Maximal even palindrome at position \( i \): the longest string to the left and right so that the left half is equal to the reverse of the right half.

\[
\text{Goal: find all maximal palindromes in } S.
\]

Construct \( S^r \), the reverse of \( S \).

Preprocess \( S \) and \( S^r \) so that LCE queries can be solved in constant time (previous slide).

LCE\((i, n-i)\) is the length of the longest palindrome centered at \( i \).

For every position \( i \):
- Compute LCE\((i, n-i)\)
Using LCE to Find Palindromes

Maximal even palindrome at position \(i\): the longest string to the left and right so that the left half is equal to the reverse of the right half.

**Goal:** find all maximal palindromes in \(S\).

Construct \(S^r\), the reverse of \(S\). \(O(|S|)\)

Preprocess \(S\) and \(S^r\) so that LCE queries can be solved in constant time (previous slide). \(O(|S|)\)

LCE\((i, n-i)\) is the length of the longest palindrome centered at \(i\).

For every position \(i\):

- Compute LCE\((i, n-i)\) \(O(1)\)

Total time = \(O(|S|)\)
**Match Statistics**

**Def.** $ms_{XY}(i) :=$ the longest substring of $X$ that starts at $i$ and matches *someplace* in $Y$.

**Algorithm sketch** to compute $ms_{XY}(i)$ in $O(|X| + |Y|)$ time and $O(|X| + |Y|)$ space:

1. Build suffix tree $T_Y$ for $Y$. \( O(|Y|) \) time
2. Compute $ms_{XY}(1)$ by querying for $X$ in $T_Y$. The depth of the node where you stop is $ms_{XY}(1)$.
3. For $i = 2 \ldots |X|$:
   - Follow the suffix link from where you stopped.
   - Continue searching for $X$ where you left off.
   - $ms_{XY}(i) = \text{the depth where the search gets stuck}$

Can also compute $p_{XY}(i) :=$ one of the location in $Y$ where a matching substring of length $ms_{XY}(i)$ occurs.
To find $LCE(i,j)$:

1. Get $p_{XY}(i)$.
2. Compute $LCE(p(i), j)$ using the old LCE algorithm.
3. Return $\min\{ms_{XY}(i), LCE(p(i), j)\}$.

**Idea.** The string in $Y$ starting at $p(i)$ is a proxy for the string in $X$ starting at $i$.

1. Get $p_{XY}(i)$.
2. Compute $LCE(p(i), j)$ using the old LCE algorithm.
3. Return $\min\{ms_{XY}(i), LCE(p(i), j)\}$.

Note that you need the suffix tree only for the *smaller* of the two strings.
Space-efficient LCE

To find $LCE(i,j)$:

1. Get $p_{XY}(i)$.
2. Compute $LCE(p(i), j)$ using the old LCE algorithm.
3. Return $\min \{ms_{XY}(i), LCE(p(i), j)\}$.

Idea. The string in $Y$ starting at $p(i)$ is a proxy for the string in $X$ starting at $i$.

1. Get $p_{XY}(i)$.
2. Compute $LCE(p(i), j)$ using the old LCE algorithm.
3. Return $\min \{ms_{XY}(i), LCE(p(i), j)\}$.

Note that you need the suffix tree only for the smaller of the two strings.
Space-efficient LCE

To find $LCE(i, j)$:

1. Get $p_{XY}(i)$.
2. Compute $LCE(p(i), j)$ using the old LCE algorithm.
3. Return $\min\{ms_{XY}(i), LCE(p(i), j)\}$.

Idea. The string in $Y$ starting at $p(i)$ is a proxy for the string in $X$ starting at $i$.

1. Get $p_{XY}(i)$.
2. Compute $LCE(p(i), j)$ using the old LCE algorithm.
3. Return $\min\{ms_{XY}(i), LCE(p(i), j)\}$.

Note that you need the suffix tree only for the smaller of the two strings.
k-mismatch using LCE

Checking whether there is a k-mismatch of $P$ starting at position $i$ of $T$:

$j = 1$  // position in $P$
$c = 0$  // number of mismatches found so far

repeat until $c > k$:
  $j += \text{LCE}(i, j)+1$  // O(1)-time longest match in $P$ and $T$ @ $(i,j)$
  $i += \text{LCE}(i, j)+1$
  If $j \geq |P|+1$: return True  // we’ve matched all of $P$
  $c++$

return False

Finding all k-mismatches of $P$ in $T$ therefore takes $O(k|T|)$-time.
Some implementation tricks and variants
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>$</th>
</tr>
</thead>
</table>
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

Ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ "end of string" character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ "end of string" character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ "end of string" character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ "end of string" character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ "end of string" character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ "end of string" character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added,
cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Alphabet = ACDE, special $ “end of string” character

ace, add, added, cede, dad, deed
Generalized Tries Can Be Compress Too:

A lot of nodes were *non-discriminatory*: they didn’t discriminate between two keys:

Only store the discriminatory nodes
Generalized Tries Can Be Compress Too:

A lot of nodes were *non-discriminatory*: they didn’t discriminate between two keys:

Only store the discriminatory nodes
Patricia Tries

Same tree, but only storing the discriminatory nodes.

**BUT:** now have to store the index of the character position the node is testing

before, a node at depth d tested position d, but now that isn’t true: we can skip over positions

Also: now you must CHECK whether a leaf you reach matches the query.

E.g. what if we searched for *cedar.*
Saving Space #2

- Store nodes in a 2-d table.
- table size = $| \sum | = \text{size of the alphabet by number of nodes } n$
- Each entry contains the index of the node it "points to"
- Uses $O(\log n)$ space instead of the size of the pointer (e.g. 32 or 64 bits)
- General trick: you ensure nodes are contained in some region of memory of size $M$. 

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>$$</th>
</tr>
</thead>
</table>
Saving Space #3

- Replace

```
A  C  D  E  $
```

- with

```
A -> D -> $ -> NULL
```

- at each node

Array at each node becomes a linked list.

Saves space when the branching factor is low
don’t need to store an entry for each character in the alphabet)

Also imagine a hybrid method, using arrays at nodes with high branching factors

dela Brandais tree
• Suffix tries natural way to store a string -- search, count occurrences, and many other queries answerable easily.

• But they are not space efficient: $O(n^2)$ space.

• Suffix trees are space optimal: $O(n)$, but require a little more subtle algorithm to construct.

• Suffix trees can be constructed in $O(n)$ time using Ukkonen’s algorithm.

• Similar ideas can be used to store sets of strings.