Suffix trees and applications

String Algorithms
Tries

... a trie is a data structure for storing and retrieval of strings ....
Tries

... a trie is a data structure for storing and retrieval of strings ....

\[ x_1 = a \ b \]
\[ x_2 = a \ b \ c \]
Tries

... a trie is a data structure for storing and retrieval of strings ....

\[ x_1 = a \ b \]

\[ x_2 = a \ b \ c \]
... a trie is a data structure for storing and retrieval of strings ....

\[
x_1 = a \ b
\]
\[
x_2 = a \ b \ c
\]
Tries

... a trie is a data structure for storing and retrieval of strings ....

\[ x_1 = \{a, b\} \]

\[ x_2 = \{a, b, c\} \]
Tries

... a trie is a data structure for storing and retrieval of strings ....

\[ x_1 = a \ b \]
\[ x_2 = a \ b \ c \]
Tries

... a trie is a data structure for storing and retrieval of strings ....

\[ x_1 = a \ b \]
\[ x_2 = a \ b \ c \]
Tries

... a trie is a data structure for storing and retrieval of strings ....

\[ x_1 = \text{a b} \]
\[ x_2 = \text{a b c} \]

Observations: shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?
... a trie is a data structure for storing and retrieval of strings ....

Observations: shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?
Tries

... a trie is a data structure for storing and retrieval of strings ...

\[
x_1 = a \ b \\
x_2 = a \ b \ c
\]

\[
x_1 = a \ b \ $ \\
x_2 = a \ b \ c \ $ \\
x_3 = $
\]

Observations: shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?
Tries

... a trie is a data structure for storing and retrieval of strings ....

\[ x_1 = a \ b \]
\[ x_2 = a \ b \ c \]

Observations: shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?
Tries

... a trie is a data structure for storing and retrieval of strings ....

\[ x_1 = a\ b \]
\[ x_2 = a\ b\ c \]
\[ x_3 = \$ \]

Observations: shared prefixes implies shared initial paths ...

Often we want each string to correspond to a unique root-to-leaf path, i.e. make sure that no input-string is a prefix of another. How?
Tries

... a trie is a data structure for storing and retrieval of strings ....

Observations: shared prefixes implies shared initial paths ...

Application: Given a query-string $y[1..m]$, we can determine if $y$ equals one of the input-strings (or a prefix of one) in time $O(m)$ ...
What is the space complexity?
Compacted tries

Saving space: Eliminate all internal nodes of degree 2 ...

If we have $n$ input-strings, then the trie has $n+1$ leaves and at most $n$ internal nodes, i.e. space $O(n)$ for the tree. What about the labels?
Compacted tries

Saving space: Eliminate all internal nodes of degree 2 ...

If we have $n$ input-strings, then the trie has $n+1$ leaves and at most $n$ internal nodes, i.e. space $O(n)$ for the tree. What about the labels?

Labels can be represented in space $O(1)$, i.e. “ab” $\Rightarrow (1, 1, 2)$
Suffix tree

The suffix tree $T(x)$ of string $x[1..n]$ is the compacted trie of all suffixes $x[i..n]$ for $i = 1,.., n+1$, i.e. including the empty suffix.
The suffix tree $T(x)$ of string $x[1..n]$ is the compacted trie of all suffixes $x[i..n]$ for $i = 1,...,n+1$, i.e. including the empty suffix.

Example for $x = tatat$
The **suffix tree** $T(x)$ of string $x[1..n]$ is the **compacted trie** of all suffixes $x[i..n]$ for $i = 1,..,n+1$, i.e. including the empty suffix

**Example for** $x = tatat$

![Suffix Tree Example]
A larger example

\[ S = \text{Mississippi$} \]

\[
\begin{array}{ccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\end{array}
\]
A larger example

Node has **path-label** `sssi` and is at **depth 3** ...

$S = \text{Mississippi}$

1 2 3 4 5 6 7 8 9 10 11 12
A larger example

Path-label of leaf $i$ is suffix $i$, i.e. $x[i..n]$$ \ldots$
A larger example

Path-label of lowest common ancestor of leaf $i$ and $j$, is longest common prefix of suffix $i$ and $j$ of $x$ ....

Node has path-label $ssi$ and is at depth 3 ...

Path-label of leaf $i$ is suffix $i$, i.e. $x[i..n]\$ $...$
What is the space complexity?
Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$. 

$S = \text{Mississippi}$

1 2 3 4 5 6 7 8 9 10 11 12
Observation: $T(S)$ requires $O(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching \(\Rightarrow\) at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ \(\Rightarrow\) pair of pointers $(i, j)$ into $S$. 

$S = Mississippis$

\[ 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 \]
Space consumption

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:
1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S$ $\Rightarrow$ pair of pointers $(i, j)$ into $S$.

$S =$ Mississippi $\quad S$
1 2 3 4 5 6 7 8 9 10 11 12

(1,12) (2,2)
Space consumption

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most $n$ leaves.
2. Each internal node is branching $\Rightarrow$ at most $n - 1$ internal nodes.
3. A tree with at most $2n - 1$ nodes has at most $2n - 2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S \Rightarrow$ pair of pointers $(i, j)$ into $S$.

$S = Mississipi$
Constructing suffix trees

Constructing $T(x)$ by inserting each suffix one by one takes time $O(n^2)$

Can we do better?
Constructing suffix trees

Constructing $T(x)$ by inserting each suffix one by one takes time $O(n^2)$

[Weiner 1973]: $T(x)$ can be constructed in time $O(n)$ ...

Can we do better?

There are two practical algorithms that construct the suffix tree in linear time: McCreight (1976) and Ukkonen (1993) ...
What about applications?

... exact matching, finding repeats, longest common substring ...
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$

If $y$ occurs in $x$ at position $i$, then $y$ is a prefix of suffix $i$ of $x$

$y$ is spelled by an initial part of the path from the root to leaf $i$ in $T(x)$
Exact matching

Given string \( x \) and pattern \( y \), report where \( y \) occurs in \( x \)
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$

Pattern $ata$ occurs at position 2 in $tatat$

Time: $O(|P|)$ using the suffix tree $T(S)$
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$.
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$
Exact matching

Given string $x$ and pattern $y$, report where $y$ occurs in $x$

Pattern $tatt$ does not occur in $tatat$

Time: $O(|P|)$ using the suffix tree $T(S)$
Repeats

A pair of substrings $R=(S[i_1..j_1], S[i_2..j_2])$ is a ...

→ exact repeat if $S[i_1, j_1] = S[i_2, j_2]$

![Diagram of exact repeat](image)
Repeats

A pair of substrings $R = (S[i_1..j_1], S[i_2..j_2])$ is a...

→ exact repeat if $S[i_1, j_1] = S[i_2, j_2]$

→ $k$-mismatch repeat if there are $k$ mismatches between $S[i_1, j_1]$ and $S[i_2, j_2]$
Repeats

A pair of substrings $R= (S[i_1..j_1], S[i_2..j_2])$ is a ...

→ **exact repeat** if $S[i_1, j_1] = S[i_2, j_2]$

→ **k-mismatch repeat** if there are $k$ mismatches between $S[i_1, j_1]$ and $S[i_2, j_2]$

→ **k-differences repeat** if there are $k$ differences (mismatches, insertions, deletions) between $S[i_1, j_1]$ and $S[i_2, j_2]$. 
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in $S$ in linear time.

Idea:

- Consider string $S$ and its suffix tree $T(S)$.
- Repeated substrings of $S$ correspond to internal locations in $T(S)$.
- Leaf numbers tell us positions where substrings occur.
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)
- It is possible to find all pairs of repeated substrings (repeats) in $S$ in linear time.

Idea:
- consider string $S$ and its suffix tree $T(S)$.
- repeated substrings of $S$ correspond to internal locations in $T(S)$.
- leaf numbers tell us positions where substrings occur.

Diagram: $S = T A T A T S$
- AT: (2, 4)
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in $S$ in linear time.

Idea:

- consider string $S$ and its suffix tree $T(S)$.
- repeated substrings of $S$ correspond to internal locations in $T(S)$.
- leaf numbers tell us positions where substrings occur.
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in $S$ in linear time.

Idea:

- consider string $S$ and its suffix tree $T(S)$.
- repeated substrings of $S$ correspond to internal locations in $T(S)$.
- leaf numbers tell us positions where substrings occur.
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in $S$ in linear time.

Idea:

- consider string $S$ and its suffix tree $T(S)$.
- repeated substrings of $S$ correspond to internal locations in $T(S)$.
- leaf numbers tell us positions where substrings occur.
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in $S$ in linear time.

Idea:

- consider string $S$ and its suffix tree $T(S)$.
- repeated substrings of $S$ correspond to internal locations in $T(S)$.
- leaf numbers tell us positions where substrings occur.
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in $S$ in linear time.

Idea:

- consider string $S$ and its suffix tree $T(S)$.
- repeated substrings of $S$ correspond to internal locations in $T(S)$.
- leaf numbers tell us positions where substrings occur.

Analysis: $O(n + z)$ time with $z = |output|$, $O(n)$ space
A larger example

$S = \text{Mississippi}$

1 2 3 4 5 6 7 8 9 10 11 12

\[
\begin{array}{cccc}
\text{i:} & (8,5) & \text{is:} & (5,2) \\
\text{(8,2)} & \text{p:} & (10,9) \\
\text{(8,11)} & \text{s:} & (7,4) \\
\text{(5,2)} & \text{si:} & (7,4) \\
\text{(5,11)} & \text{iss:} & (5,2) \\
\text{(2,11)} & \text{ss:} & (6,3) \\
\text{issi:} & (5,2) & \text{ssi:} & (6,3) \\
\text{(4,6)} & \text{(4,3)} & \text{(6,3)}
\end{array}
\]
Finding *maximal* exact repeats
Finding *maximal* exact repeats
Finding *maximal* exact repeats
Finding **maximal** exact repeats

**Idea:**

- For right-maximality ($X \neq Y$)
  - consider only **internal nodes** of $T(S)$
  - report only pairs of leaves from different subtrees (or from different **leaf-lists**)
Finding maximal exact repeats

Idea:

- For right-maximality ($X \neq Y$)
  - consider only internal nodes of $T(S)$
  - report only pairs of leaves from different subtrees (or from different leaf-lists)

- For left-maximality ($A \neq B$)
  - keep lists for the different left-characters
  - report only pairs from different lists

Analysis: $\mathcal{O}(n + z)$ time with $z = |\text{output}|$, $\mathcal{O}(n)$ space
Other repeats

Maximal repeats with bounded gap in time $O(n \log n + z)$

Tandem repeats in time $O(n \log n + z)$

Palindromic repeats in $O(n + z)$

... all using suffix trees ...
More strings

The *longest common substring* of $x[1..n]$ and $y[1..m]$ is the longest string $z$ which occurs in both $x$ and $y$ ...

Can this be found efficiently using a suffix tree?
The *longest common substring* of $x[1..n]$ and $y[1..m]$ is the longest string $z$ which occurs in both $x$ and $y$ ...

Can this be found efficiently using a suffix tree?

$z$ is the longest common prefix of any pair of suffixes $x[i..n]$ and $y[j..m]$.
More strings

\( z \) is the longest common prefix of any pair of suffixes \( x[i..n] \) and \( y[j..m] \).

**Idea:** Build a compacted trie of all suffixes of \( x \) and \( y \), such that each suffix of \( x \) and \( y \) corresponds to unique root-to-leaf paths ...
More strings

\( z \) is the longest common prefix of any pair of suffixes \( x[i..n] \) and \( y[j..m] \)

**Idea:** Build a compacted trie of all suffixes of \( x \) and \( y \), such that each suffix of \( x \) and \( y \) corresponds to unique root-to-leaf paths ...
More strings

$z$ is the longest common prefix of any pair of suffixes $x[i..n]$ and $y[j..m]$

Idea: Build a compacted trie of all suffixes of $x$ and $y$, such that each suffix of $x$ and $y$ corresponds to unique root-to-leaf paths ...
More strings

$z$ is the longest common prefix of any pair of suffixes $x[i..n]$ and $y[j..m]$

**Idea:** Build a compacted trie of all suffixes of $x$ and $y$, such that each suffix of $x$ and $y$ corresponds to unique root-to-leaf paths ...
More strings

\( z \) is the longest common prefix of any pair of suffixes \( x[i..n] \) and \( y[j..m] \)

**Idea:** Build a compacted trie of all suffixes of \( x \) and \( y \), such that each suffix of \( x \) and \( y \) corresponds to unique root-to-leaf paths ...
More strings

$z$ is the longest common prefix of any pair of suffixes $x[i..n]$ and $y[j..m]$

Idea: Build a compacted trie of all suffixes of $x$ and $y$, such that each suffix of $x$ and $y$ corresponds to unique root-to-leaf paths ...
More strings

\( z \) is the longest common prefix of any pair of suffixes \( x[i..n] \) and \( y[j..m] \)

**Idea:** Build a compacted trie of all suffixes of \( x \) and \( y \), such that each suffix of \( x \) and \( y \) corresponds to unique root-to-leaf paths ...

\[
\begin{align*}
tatat$ & \quad aataa# \\
atat$ & \quad ataa# \\
tat$ & \quad taa# \\
at$ & \quad aa# \\
t & \quad a# \\
\varepsilon & \quad \varepsilon#
\end{align*}
\]
More strings

\( z \) is the longest common prefix of any pair of suffixes \( x[i..n] \) and \( y[j..m] \)

\[
\begin{align*}
tatat$ & \quad aataa# \\
atat$ & \quad ata$a# \\
tat$ & \quad taa# \\
at$ & \quad aa# \\
t & \quad a# \\
\varepsilon & \quad \varepsilon#
\end{align*}
\]

Observe: \( z \) is the path-label of the deepest node with suffixes from both \( x \) and \( y \) as leaves in its sub-tree ...
More strings

$z$ is the longest common prefix of any pair of suffixes $x[i..n]$ and $y[j..m]$

Observe: $z$ is the path-label of the deepest node with suffixes from both $x$ and $y$ as leaves in its sub-tree ... Time: $O(n+m)$
Generalized suffix tree

This is the **generalized suffix tree** of `tatat` and `aataa`

```
tatat$
atat$
tat$
at$
$ ε$

aataa#
ataa#
taa#
aa#
a#
$ ε#
```

Can be constructed by constructing the suffix tree of ...

```
tatat$aataa#
```
Generalized suffix tree

... we must argue that we get the same branching structure ...

$1 \ldots n+2 \ldots n+m+1 \ldots 1 \ldots m$
Generalized suffix tree

Case 1:

$n+2$  $n+m+1$

1  $n$  $n+2$  $n+m+1$

Case 1:

1  $n$  $n+2$  $n+m+1$

1  $n$  $n+2$  $n+m+1$

1  $n$  $n+2$  $n+m+1$
Generalized suffix tree

Case 1:

\[
1 \quad n \quad n+2 \quad n+m+1
\]

\[
j \quad i \quad 1 \quad m \quad #
\]
Generalized suffix tree

Case 1:

```
1  n  n+2
\_\_\_\_\_
  1   n   1
  \_\_\_\_
```

```
1  m
\_\_\_\_
  1   m
  \_\_\_\_
```

```
(1, i)
```

```
(1, j)
```

```
(1, i)
```
Generalized suffix tree

Case 2:

$1 \quad n \quad 1 \quad m \quad \#$

(i, #) and (i' + (n + 1), #)

(2, j') and (2, i')
Generalized suffix tree

Case 3:

$$i \quad \#$$

$$j \quad \#$$

$$j' + (n+1)$$

$$1 \quad n \quad 1 \quad m \quad n+2 \quad n+m+1$$

$$(1, i) \quad (2, j') \quad \$$
Is everything great?
Space consumption

Fact: $T(x)$ requires $O(n)$ space, where $n = |x|$

... but how much space does it consume in “practice”?
Representation of suffix trees

Standard representation of trees:
- Store nodes as records with child and sibling pointer.
- An edge label $(i, j)$ is stored at node below the edge.
  $\Rightarrow$ about $32n$ bytes in the worst case
  $2n$ nodes $\times$ (2 integers + 2 pointers)

Ideas for more efficient representation:
- Do not represent leaves explicitly.
- Avoid sibling pointers by storing all children of the same node in a row.
- Do not represent the right pointer of an edge label.
  $\Rightarrow$ below $12n$ bytes in the worst case, $8.5n$ on average
Space consumption

Fact: $T(x)$ requires $O(n)$ space, where $n=|x|$, but ...

... in practice somewhere between 10 and 40 bytes per letter in $x$ ...

Is this a problem? Depends on $n$, if $\approx 500.000.000$ then yes...
Alphabet size

How much time does it take to find the proper edge out from a node when searching in a suffix tree?
Alphabet size

How much time does it take to find the proper edge out from a node when searching in a suffix tree?

Time proportional to the out-degree of the node \( \leq |A| \) ...

... search time in “pratice” is \( O(|A| \cdot |P|) \) ...

If \(|A|\) is large, e.g. 256, this matters!!
Alphabet size

How much time does it take to find the proper edge out from a node when searching in a suffix tree?

Idea 1: Organising children in a search-tree, reduces search time from $|A|$ to $O(\log |A|)$ ... (requires an ordered alphabet)
Alphabet size

How much time does it take to find the proper edge out from a node when searching in a suffix tree?

**Idea 2:** Organising children in a vector of size $|A|$ indexed by letters, reduces search time from $|A|$ to $O(1)$ ... (requires a finite alphabet)
Alphabet size

How much time does it take to find the proper edge out from a node when searching in a suffix tree?

Idea 3: Use some other dictionary for mapping letters to children ...  

... the alphabet size matters in practice ...