Introduction to Suffix Trees

Suffix tree is an index structure that gives efficient solutions to a wide range of complex string problems.

For example, the substring problem:

For a text $S$ of length $m$, after $O(m)$ time preprocessing, given any string $P$ either find an occurrence of $P$ in $S$, or determine that one does not exist, in time $O(|P|)$.

How to solve it?

Definition of a Suffix Tree

A suffix tree $T$ for $S[1 \ldots m]$ is a rooted tree with:

- $m$ leaves numbered $1, \ldots, m$.
- at least two children for each internal node (with the root as a possible exception).
- each edge labeled by a nonempty substring of $S$.
- no two edges out of a node beginning with the same character.

Again, $L(v)$ denotes the label of a node $v$, i.e., the concatenation of edge labels on the path from the root to $v$.

Key feature:

- $L(i) = S[i \ldots m]$ for each leaf $i = 1, \ldots, m$ of $T$.

Ensuring the Existence of a Suffix Tree

Not necessarily:

If some suffix of $w$ appears as a prefix of some other one, the path labeled by $w$ does not lead to a leaf.

To avoid this, assume that $S$ has a "termination character" $\$" that occurs only at the end of $S$ (or insert one, if needed)

- no suffix appears as a prefix of any other
- suffixes label complete paths leading to the leaves.

Solving the Substring Problem

First idea: Build a keyword tree of all substrings of $S$.

No: takes too much time ($O(n^2)$).

Second idea: It is easy to find prefixes of strings in a keyword tree. Each substring $S[i \ldots j]$ is a prefix of the suffix $S[i \ldots m]$ of $S$.

- create a keyword tree of the $m$ non-empty suffixes of $S$.

The rest is just refinement . . . (yet quite challenging to get the construction time to linear!)

Example of a Suffix Tree

The suffix tree for string $x a b a x a c$: $\$

Does a suffix tree always exist?

Applying Suffix Trees to Matching

Suffix trees can be used to solve exact matching:

1. Construct the suffix tree $T$ for text $T[1 \ldots m]$; (We’ll discuss later how to do this in $O(m)$ time)

2. Match characters of $P$ along a path from the root.

(a) If $P$ can be fully matched, let $z$ be the number of leaves below the path labeled by $P$. Each of these is a start of an occurrence of $P$, and they can be collected in time $O(z)$ (Exercise).

(b) If it doesn’t match completely, $P$ doesn’t occur in $T$.

Total time: $O(|T| + |P| + z)$.

(1) $P$ may end at the middle of an edge.
Naive Construction of Suffix Trees

Start with a root and a leaf numbered 1, connected by an edge labeled $S$.

Enter suffixes $S[2 \ldots m]$, $S[3 \ldots m]$, $S[4 \ldots m]$, and $S[m]$ into the tree as follows:

To insert $K_i = S[i \ldots m]$, follow the path from the root matching characters of $K_i$, until the first mismatch at character $K_i[j]$ (which is bound to happen)

(a) If the matching cannot continue from a node, denote that node by $w$

(b) Otherwise the mismatch occurs at the middle of an edge, which has to be split

Example of the Naive Construction

Consider building the suffix tree for the string $S$:

```
1 2 3 4 5 6
x a b x a c
```

Start:

```
  x a b x a c
```

After inserting the second and the third suffix:

```
x a b x a c
```

```
  a b x c
     \  /  \
    x a c
```

Example of the Naive Construction (2)

Entering $S[4 \ldots 6] = x a c$ causes the first edge to split:

```
x a c
```

```
  
```

Same happens for the second edge when $S[5 \ldots 6] = a c$ is entered

Example of the Naive Construction (3)

After entering suffixes $S[5 \ldots 6] = a c$ and $S[6] = c$ the suffix tree is complete:

```
  x a
     |  \
    a c
```

Compact Representation

Each edge is labeled by a non-empty substring $S[i \ldots j]$.

Compression: Represent label $S[i \ldots j]$ by two indices $i$ and $j$ to the string $S$

--- each edge takes only constant space, and thus $O(m)$ space suffices for the entire suffix tree of $S[1 \ldots m]$

Example of Compact Representation

The suffix tree for a string with compacted edge labels

```
1 2 3 4 5 6
x a b x a c
```

```
  1
   1
```

```
  2
    2
```

```
  3
    3
```

```
  4
```

```
  5
```

Complexity of the Naive Construction

Each suffix $S[i \ldots m]$ is entered in the tree in time $\Theta(|S[i \ldots m]|)$ so total time is $\Theta(n^2)$

Observations: Number of edges in a suffix tree $T$ is at most $2n - 1$ (Exercise)

On the other hand, the total length of edge labels can be $O(n^2)$ (Exercise)

As a simple example consider the suffix tree of $a b \ldots x y z$, whose total length of edge labels is $\sum_{i=1}^{n} l_i = 26 \times 27/2$

For linear time we need a compact representation of edge labels

Notes on the previous page:

- (Exercise) if the mismatch occurs at the middle of an edge $e = (u, v)$, let the label of that edge be $a_1 \ldots a_k$

- (Exercise) if the mismatch occurred at character $a_k$, then create a new node $w$, and replace $e$ by edges $(u, w)$ and $(w, v)$ labeled by $a_1 \ldots a_{k-1}$ and $a_k \ldots a_l$

- (Exercise) finally, in both cases (a) and (b), create a new leaf numbered $i$, and connect $w$ to it by an edge labeled with $K_i[\ldots |K_i|]$
Denote the implicit suffix tree of the prefix \(j = 1\)

Let's call such labels of (partial) paths (string) paths

That is, a string path of \(I_i\) is

- a string that can be matched along the edges, starting from the root
- a prefix of any node label
- any substring of \(S[1 \ldots i]\)

Start with \(T \leftarrow I_1\)

Update \(T\) to trees \(I_2, \ldots, I_{m+1}\) in \(m\) phases (vaihe)

Let \(S[m+1]\) be \(\$\) -- the final value of \(T\) is a true suffix tree, which contains all suffixes of \(S\) (extended with \(\$\))

- Phase \(i + 1\) updates \(T\) from \(I_i\) (with all suffixes of \(S[1 \ldots i]\)) to \(I_{i+1}\) (with all suffixes of \(S[1 \ldots i + 1]\))

Each phase \(i + 1\) consists of extensions (lisäysaskel)

\(j = 1, \ldots, i + 1\)

Extension \(j\) ensures that suffix \(S[j \ldots i + 1]\) is in \(I_{i+1}\)

Ukkonen’s Method

Ukkonen’s method constructs a suffix tree for \(S[1 \ldots n]\) in time \(O(n)\)

We begin with a high-level description, and then refine the method to run in linear time

The method builds, as intermediate results, for each prefix \(S[1 \ldots 1], S[1 \ldots 2], \ldots, S[1 \ldots m]\), an implicit suffix tree

The implicit suffix tree of a string is what results by applying suffix tree construction to the string without an added end marker \(\$\)

~ all suffixes are included, but not necessarily as labels of complete root-to-leaf paths

Example of Implicit Suffix Trees (1)

Example of Implicit Suffix Trees (2)

Implicit Suffix Trees of Prefixes

Denote the implicit suffix tree of the prefix \(S[1 \ldots i]\) by \(I_i\)

\(I_i\) is just a single edge labeled by \(S[1]\) leading to leaf 1

Example:

Implicit suffix trees for the first three prefixes of \(axaba\):

- \(I_1\):
  - a
  - 1
- \(I_2\):
  - ax
  - x
  - 2
- \(I_3\):
  - axa
  - xa
  - 1
  - 2

String Paths of \(I_i\)

\(I_i\) contains each suffix \(S[1 \ldots i], S[2 \ldots i], \ldots, S[i]\) of \(S[1 \ldots i]\) as a label of some path (possibly ending at the middle of an edge)

Let’s call such labels of (partial) paths (string) paths

That is, a string path of \(I_i\) is

- a string that can be matched along the edges, starting from the root
- a prefix of any node label
- any substring of \(S[1 \ldots i]\)

BSA Lecture 6: Intro to suffix trees – p.17/46

BSA Lecture 6: Intro to suffix trees – p.18/46

BSA Lecture 6: Intro to suffix trees – p.19/46

BSA Lecture 6: Intro to suffix trees – p.20/46

BSA Lecture 6: Intro to suffix trees – p.21/46

BSA Lecture 6: Intro to suffix trees – p.22/46

BSA Lecture 6: Intro to suffix trees – p.23/46

BSA Lecture 6: Intro to suffix trees – p.24/46
### Suffix Extension Rules

**Extension rules** for the three possible cases:

**Rule 1** If $S[j + i]$ leads to a leaf $(j)$, concatenate $S[i + 1]$ to its edge label.

**Rule 2** If path $S[j + i]$ ends before a leaf, and doesn’t continue by $S[i + 1]$: Connect the end of the path to a new leaf of edge labeled by char $S[i + 1]$. (If the path ended at the middle of an edge, split the edge and insert a new node as the parent of the leaf.)

**Rule 3** If the path continues by $S[i + 1]$, do nothing.

(Suffix $S[j + i + 1]$ is already in the tree)

Let’s call an extension that applies Rule 3 **void** (and applications of Rules 1 and 2 **non-void**).

### Complexity of a Naive Implementation

Consider first a single phase $i + 1$.

Each of the **extension rules** can be applied in constant time $\sim$ for extensions $j = 1, \ldots, i + 1$ they take total time $\Theta(i)$

Traversing the paths $S[1], \ldots, S[i + 1]$ explicitly takes time $\Theta(\sum_{i=0}^{m} t_i) = \Theta(i^2)$

$\sim$ Total time for all phases $i = 2, \ldots, m + 1$ is $\Theta(\sum_{i=2}^{n+i} t_i) = \Theta(m^2)$

Q: How to improve this?

### Locating Ends of Paths

The extensions of phase $i + 1$ need to locate the ends of all the $i + 1$ suffixes of $S[1 \ldots i]$.

How to do this efficiently?

For each internal node $v$ of $T$, labeled by $v = x_0$, where $x \in \Sigma$ and $x \in \Sigma^*$, define $s(v)$ to be the node labeled by $x$.

We’ll show that these exist, in a moment.

Then a pointer from $v$ to $s(v)$ is the **suffix link** of $v$.

**NB:** If node $v$ is labeled by a single char $(x)$, then $\alpha = \varepsilon$ and $s(v)$ is the root.

### Intuitive Motivation for Suffix Links

Extension $j$ of phase $i + 1$ finds the end of the path $S[j + i]$ in the tree (and extends it with char $S[i + 1]$).

Extension $j + 1$ finds the end of the path $S[j + 1 + i]$.

Assume that $v$ is an internal node labeled by $S[j]a$ on the path $S[j + i]$. Then we can avoid traversing path $a$ when locating the end of $S[j + 1 + i]$, by starting from node $s(v)$.

Q: Do suffix links always exist?

A: Yes, and each suffix link $(v, s(v))$ is easy to set.

### Example of Extensions

Consider phase 6 with string $S = axabxbx$.

Now $T = I_6$ contains all suffixes of $S[1 \ldots 5]$ = $axabx$.

```
| xabx | 1 |
| xabx | 2 |
| bxbx | 4 |
```


Extension 5: $S[6 \ldots 6] = x$ enters Rule 2, creating leaf 5 and its parent.

Extension 6: $S[6 \ldots 6] = b$ entered by Rule 3

```
| xabx | 1 |
| axabxb | 3 |
| bxbx | 4 |
```

### Example of Suffix Links

```
| axabxb | 1 |
| bxac   | 2 |
| bxac   | 3 |
| bxac   | 4 |
```

What are suffix links good for?

### Computation of Suffix Links

**Observation:** If an internal node $v$ is created during extension $j$ of phase $i + 1$, then the next extension $j + 1$ will find out the node $s(v)$.

Why?

Let $v$ be labeled by $x_0$.

Now $x_0 = S[j \ldots i]$, and node $v$ is created by Rule 2.

That is, $v$ is inserted at the end of path $S[j \ldots i]$, which continued by some character $c \neq S[i + 1]$

$\Rightarrow$ paths $x_0 c$ and $x c$ were entered earlier.

$\Rightarrow$ in extension $j + 1$, node $s(v)$ is either found or created at the end of path $\alpha = S[j + 1 \ldots i]$. 

**Example:** Consider phase 6 with string $S = axabxbx$.

Now $T = I_6$ contains all suffixes of $S[1 \ldots 5] = axabx$.

```
| xabx | 1 |
| xabx | 2 |
| bxbx | 4 |
```


Extension 5: $S[6 \ldots 6] = x$ enters Rule 2, creating leaf 5 and its parent.

Extension 6: $S[6 \ldots 6] = b$ entered by Rule 3

```
| xabx | 1 |
| axabxb | 3 |
| bxbx | 4 |
```

**Example of Extensions**

Consider phase 6 with string $S = axabxbx$.

Now $T = I_6$ contains all suffixes of $S[1 \ldots 5] = axabx$.

```
| xabx | 1 |
| xabx | 2 |
| bxbx | 4 |
```


Extension 5: $S[6 \ldots 6] = x$ enters Rule 2, creating leaf 5 and its parent.

Extension 6: $S[6 \ldots 6] = b$ entered by Rule 3

```
| xabx | 1 |
| axabxb | 3 |
| bxbx | 4 |
```

### Intuitive Motivation for Suffix Links

Extension $j$ (of phase $i + 1$) finds the end of the path $S[j + i]$ in the tree (and extends it with char $S[i + 1]$).

Extension $j + 1$ finds the end of the path $S[j + 1 + i]$.

Assume that $v$ is an internal node labeled by $S[j]a$ on the path $S[j + i]$. Then we can avoid traversing path $a$ when locating the end of $S[j + 1 + i]$, by starting from node $s(v)$.

Q: Do suffix links always exist?

A: Yes, and each suffix link $(v, s(v))$ is easy to set.

### Computation of Suffix Links

**Observation:** If an internal node $v$ is created during extension $j$ (of phase $i + 1$), then the next extension $j + 1$ will find out the node $s(v)$.

Why?

Let $v$ be labeled by $x_0$.

Now $x_0 = S[j \ldots i]$, and node $v$ is created by Rule 2.

That is, $v$ is inserted at the end of path $S[j \ldots i]$, which continued by some character $c \neq S[i + 1]$

$\Rightarrow$ paths $x_0 c$ and $x c$ were entered earlier.

$\Rightarrow$ in extension $j + 1$, node $s(v)$ is either found or created at the end of path $\alpha = S[j + 1 \ldots i]$. 

**Example:** Consider phase 6 with string $S = axabxbx$.

Now $T = I_6$ contains all suffixes of $S[1 \ldots 5] = axabx$.

```
| xabx | 1 |
| xabx | 2 |
| bxbx | 4 |
```


Extension 5: $S[6 \ldots 6] = x$ enters Rule 2, creating leaf 5 and its parent.

Extension 6: $S[6 \ldots 6] = b$ entered by Rule 3

```
| xabx | 1 |
| axabxb | 3 |
| bxbx | 4 |
```
**Speeding up Path Traversals**

Consider extensions of phase $i + 1$

Extension 1 extends path $S[1 \ldots i]$ with char $S[i + 1]$
Path $S[1 \ldots i]$ always ends at leaf 1, and is thus extended by Rule 1

$\leadsto$ Extension 1 can be performed in constant time, by maintaining a pointer to the incoming edge of leaf 1

What about subsequent extensions $j + 1$ for $j = 1, \ldots, i$?

**Short-cutting Traversals**

In case (b), let $x\alpha\beta$ be the label of $v$
$\Rightarrow$ $S[j \ldots i] = x\alpha\beta$ for some $\beta \in \Sigma^*$

(Show a diagram of the path)

Then follow the suffix link of $v$, and continue by matching $\beta$ down-wards from node $s(v)$ (which is now labeled by $\alpha$)

Having found the end of path $\alpha\beta = S[j+1 \ldots i]$, apply extension rules to ensure that it continues with $S[j+1]$

Finally, if a new internal node $w$ was created in extension $j$, set its suffix link to point to the end node of path $S[j+1 \ldots i]$

**Bounding the Time of Tree Traversals**

**Lemma 6.1.2** For any node $v$ with a suffix link to $s(v)$,

$$\text{depth}(s(v)) \geq \text{depth}(v) - 1$$

That is, following a suffix link leads at most one level closer to the root

**Proof.** (Idea) The suffix links for any ancestor of $v$ lead to distinct ancestors of $s(v)$

Now we can argue a linear time bound for any phase by considering the current node depth can change
- $\geq$ the depth of the most recently visited node

**Locating Subsequent Paths**

Extension $j$ has located the end of the path $S[j \ldots i]$

Starting from there, walk up at most one node either
- (a) to the root, or
- (b) to a node $v$ with a suffix link

In case (a), traverse path $S[j+1 \ldots i]$ explicitly down-wards from the root

**Speeding up Explicit Traversals**

(Trick 1 (skip/count) in Gusfield)
The path $S[j \ldots i]$ followed in extension $j$ is in the tree
$\Rightarrow$ it is a suffix of $S[1 \ldots i]$

$\Rightarrow$ no need to check all of its characters; it suffices to choose the correct edges
Let $S[k]$ be the next char to be matched on path $S[j \ldots i]$

An edge labeled by $S[p \ldots q]$ can be traversed simply by checking that $S[p] = S[k]$, and skipping the next $q - p$ chars of $S[j \ldots i]$

$\Rightarrow$ time to traverse a path is proportional to the node-length on the path (instead of its string-length)

**Bounding the Edge Traversals**

The possible up movement and suffix link traversal decrement current node depth at most twice

$\Rightarrow$ the current node depth is decremented at most $2m$ times during the entire phase

On the other hand, the current node depth cannot exceed $m$ $\Rightarrow$ it is incremented (by following downward edges) at most $3m$ times

$\Rightarrow$ total time of a phase is $O(m)$

**Final Improvements**

Some extensions can be found unnecessary to compute explicitly

**Obs 1:** Rule 3 is a “show-stopper”:
- If path $S[j \ldots i+1]$ is already in the tree, so are paths $S[j+1 \ldots i+1], \ldots, S[i+1]$, too

$\Rightarrow$ Phase $i + 1$ can be finished at the first extension $j$ that applies Rule 3; all the rest are void, too

**Locating Subsequent Paths**

**Linear Bound for any Single Phase**

**Theorem 6.1.1** Using suffix links and the skip/count trick, a single phase takes time $O(m)$

**Proof**

There are $i + 1 \leq m + 1$ extensions in phase $i + 1$

In any extension, other work except tree-traversals takes constant time only

How to bound the work for traversing the tree?
To find the end of the next path, an extension first moves at most one level up. Then a suffix link may be followed, which is followed by a down traversal to match the rest of the path

**Speeding up Explicit Traversals**

(Trick 1 (skip/count) in Gusfield)
The path $S[j \ldots i]$ followed in extension $j$ is in the tree
$\Rightarrow$ it is a suffix of $S[1 \ldots i]$

$\Rightarrow$ no need to check all of its characters; it suffices to choose the correct edges
Let $S[k]$ be the next char to be matched on path $S[j \ldots i]$

An edge labeled by $S[p \ldots q]$ can be traversed simply by checking that $S[p] = S[k]$, and skipping the next $q - p$ chars of $S[j \ldots i]$

$\Rightarrow$ time to traverse a path is proportional to the node-length on the path (instead of its string-length)

**Locating Subsequent Paths**

**Final Improvements**

Some extensions can be found unnecessary to compute explicitly

**Obs 1:** Rule 3 is a “show-stopper”:
- If path $S[j \ldots i+1]$ is already in the tree, so are paths $S[j+1 \ldots i+1], \ldots, S[i+1]$, too

$\Rightarrow$ Phase $i + 1$ can be finished at the first extension $j$ that applies Rule 3; all the rest are void, too

**Bounding the Time of Tree Traversals**

**Lemma 6.1.2** For any node $v$ with a suffix link to $s(v)$,

$$\text{depth}(s(v)) \geq \text{depth}(v) - 1$$

That is, following a suffix link leads at most one level closer to the root

**Proof.** (Idea) The suffix links for any ancestor of $v$ lead to distinct ancestors of $s(v)$

Now we can argue a linear time bound for any phase by considering the current node depth can change
- $\geq$ the depth of the most recently visited node

**Locating Subsequent Paths**

Extension $j$ has located the end of the path $S[j \ldots i]$

Starting from there, walk up at most one node either
- (a) to the root, or
- (b) to a node $v$ with a suffix link

In case (a), traverse path $S[j+1 \ldots i]$ explicitly down-wards from the root

**Bounding the Edge Traversals**

The possible up movement and suffix link traversal decrement current node depth at most twice

$\Rightarrow$ the current node depth is decremented at most $2m$ times during the entire phase

On the other hand, the current node depth cannot exceed $m$ $\Rightarrow$ it is incremented (by following downward edges) at most $3m$ times

$\Rightarrow$ total time of a phase is $O(m)$

**Final Improvements**

Some extensions can be found unnecessary to compute explicitly

**Obs 1:** Rule 3 is a “show-stopper”:
- If path $S[j \ldots i+1]$ is already in the tree, so are paths $S[j+1 \ldots i+1], \ldots, S[i+1]$, too

$\Rightarrow$ Phase $i + 1$ can be finished at the first extension $j$ that applies Rule 3; all the rest are void, too
**Final Improvements (2)**

**Obs 2**: A node created as a leaf stays as a leaf because no extension rule adds children to a leaf.

If extension \( j \) created a leaf (numbered \( j \)), extension \( j \) of any later phase \( i + 1 \) applies Rule 1 (appending the next char \( S[i+1] \) to the edge label of \( j \)).

Explicit applications of Rule 1 can be eliminated as follows:

- Use compressed edge representation (i.e., indices \( p \) and \( q \) instead of substring \( S[p..q] \)), and
- Represent the end position of each terminal edge by a global value \( e \) for “current end position”.

**Eliminating Extensions**

Denote by \( j_i \) the last non-void extension of phase \( i \) (that is, application of Rule 1 or 2); \( j_1 = 1 \).

**Observations**:

- Extensions \( j_1;..j_i \) of phase \( i \) are non-void leaves \( 1;..j_i \) have been created at the end of phase \( i \).
- Extensions \( j_i;..j \) of any subsequent phase all apply Rule 1; they are non-void in phase \( i + 1 \), too.

Sufficient to execute explicitly in phase \( i + 1 \), only extensions \( j_i;..j \).

**Single Phase Algorithm**

Algorithm for phase \( i + 1 \) with unnecessary extensions eliminated:

1. Set current end-position: \( e := i + 1 \);
   (to implement extensions \( j_1;..j_i \) implicitly)
2. Compute extensions \( j_i+1;..j \) until \( j > i + 1 \) or Rule 3 was applied in extension \( j \);
3. Set \( j_{i+1} := j_i - 1 \); (for the next phase)

All these tricks together can be shown to lead to linear time.

**Analysis of the Tuned Implementation**

Let \( j = 1;..m+1 \) denote the index of the current extension.

Over all phases \( 2;..m+1 \) index \( j \) never decreases, but it can remain the same at the start of phases \( 3;..m+1 \) at most \( 2m \) extensions are computed explicitly.

Similarly to the proof of Th. 6.1.1, the current node depth can be decremented at most \( 4m \) times, and thus the total length of all downward traversals is bounded by \( 5mn \).

**A Final Touch**

Finally, \( T_m \) can be converted to the true suffix tree of \( S[1..m] \) as follows:

All occurrences of the “current end position” marker \( e \) on edge labels can be replaced by \( m + 1 \) (with a simple tree traversal, in time \( O(n) \)).

**Remark**: Ukkonen’s method is “on-line”, by processing \( S \) left-to-right and having a suffix tree ready for the scanned part.