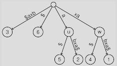


Datenstrukturen und Effiziente Algorithmen

Vorlesung *Datenstrukturen und Effiziente Algorithmen* im WS 18/19

Marc Hellmuth
Institut für Mathematik und Informatik
Universität Greifswald

Introduction

Suffix Tree

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems
- first linear-time algorithm for suffix tree construction found in 1973 (Wiener)
- simpler algorithm by Ukkonen (1995), that we will cover

Introduction

Suffix Tree

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems
- first linear-time algorithm for suffix tree construction found in 1973 (Wiener)
- simpler algorithm by Ukkonen (1995), that we will cover

Introduction

Suffix Tree

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n + m)$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n + k)$, where k is the number of occurences of P in T
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems
- first linear-time algorithm for suffix tree construction found in 1973 (Wiener)
- simpler algorithm by Ukkonen (1995), that we will cover

Introduction

Suffix Tree

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems
- first linear-time algorithm for suffix tree construction found in 1973 (Wiener)
- simpler algorithm by Ukkonen (1995), that we will cover

Introduction

Suffix Tree

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems
- first linear-time algorithm for suffix tree construction found in 1973 (Wiener)
- simpler algorithm by Ukkonen (1995), that we will cover

Introduction

Suffix Tree

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n + m)$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n + k)$, where k is the number of occurences of P in T
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems
- first linear-time algorithm for suffix tree construction found in 1973 (Wiener)
- simpler algorithm by Ukkonen (1995), that we will cover

Introduction

Suffix Tree

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n + m)$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n + k)$, where k is the number of occurences of P in T
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems
- first linear-time algorithm for suffix tree construction found in 1973 (Wiener)
- simpler algorithm by Ukkonen (1995), that we will cover

Introduction

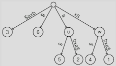
Suffix Tree

- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems
- first linear-time algorithm for suffix tree construction found in 1973 (Wiener)
- simpler algorithm by Ukkonen (1995), that we will cover

Introduction

Suffix Tree

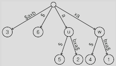
- data structure build from a string
- will assume that the alphabet size is a constant
- also allows to solve the exact matching problem in time $O(n+m)$
- but here: preprocessing of text T in $O(m)$ and then searching of P in T in time $O(n+k)$, where k is the number of occurences of P in T
- Z-Algorithm (and also Boyer-Moore) requires time $\Omega(m)$ for searching
- suffix trees much more efficient than Z-Algorithm or Boyer-Moore, when $m \gg n$ and many patterns are searched in fixed text
- suffix trees flexible data structure to solve many more string problems
- first linear-time algorithm for suffix tree construction found in 1973 (Wiener)
- simpler algorithm by Ukkonen (1995), that we will cover



Definition: Suffix Tree

Definition 1 (suffix tree)

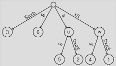
- A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m.
- Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S.
- No two edges out of a node can have edge-labels beginning with the same character.
- For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix $S[i..m]$.



Definition: Suffix Tree

Definition 1 (suffix tree)

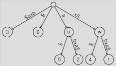
- A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m.
- Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S.
- No two edges out of a node can have edge-labels beginning with the same character.
- For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix $S[i..m]$.



Definition: Suffix Tree

Definition 1 (suffix tree)

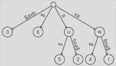
- A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m.
- Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S.
- No two edges out of a node can have edge-labels beginning with the same character.
- For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix $S[i..m]$.



Definition: Suffix Tree

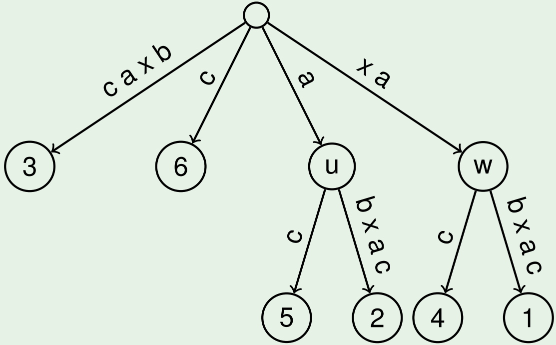
Definition 1 (suffix tree)

- A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m.
- Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S.
- No two edges out of a node can have edge-labels beginning with the same character.
- For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix $S[i..m]$.

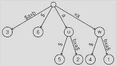


Example: Suffix Tree

Example 2 (Suffix tree for $S = xabxac$)



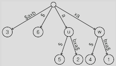
(letters on the edges to be read top-down)



Suffix Tree

Existence

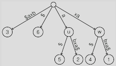
- if a suffix of S is also as proper substring of S then no suffix tree according to above definition exists
- if the last character of S does not appear elsewhere, then a suffix tree always exists
- therefore will append a unique character \$ to S and assume that \$ does not appear in S
- will build suffix tree of S\$ but sometimes not explicitly mention that \$ has been added



Suffix Tree

Existence

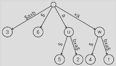
- if a suffix of S is also as proper substring of S then no suffix tree according to above definition exists
- if the last character of S does not appear elsewhere, then a suffix tree always exists
- therefore will append a unique character \$ to S and assume that \$ does not appear in S
- will build suffix tree of S\$ but sometimes not explicitly mention that \$ has been added



Suffix Tree

Existence

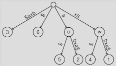
- if a suffix of S is also as proper substring of S then no suffix tree according to above definition exists
- if the last character of S does not appear elsewhere, then a suffix tree always exists
- therefore will append a unique character \$ to S and assume that \$ does not appear in S
- will build suffix tree of S\$ but sometimes not explicitly mention that \$ has been added



Suffix Tree

Existence

- if a suffix of S is also as proper substring of S then no suffix tree according to above definition exists
- if the last character of S does not appear elsewhere, then a suffix tree always exists
- therefore will append a unique character \$ to S and assume that \$ does not appear in S
- will build suffix tree of S\$ but sometimes not explicitly mention that \$ has been added



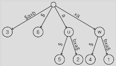
Suffix Tree, Definitions

Definition 3 (label of a path)

The label of a path in a suffix tree $\mathcal{T}$ is the concatenation, in order, of the substrings labeling the edges of that path. The path-label of a node is the label of the path from the root of $\mathcal{T}$ to that node.

Definition 4 (string-depth)

For any node v in a suffix tree, the string-depth of v is the number of characters in v's label.



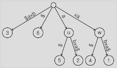
Suffix Tree, Definitions

Definition 3 (label of a path)

The label of a path in a suffix tree $\mathcal{T}$ is the concatenation, in order, of the substrings labeling the edges of that path. The path-label of a node is the label of the path from the root of $\mathcal{T}$ to that node.

Definition 4 (string-depth)

For any node v in a suffix tree, the string-depth of v is the number of characters in v's label.



Suffix Tree, Definitions

Definition 5 (path splits an edge)

Let (u, v) be an edge of a suffix tree and the string α be its label. For a proper prefix α' of α we then say the path from root to u and α' specify together a path that splits the edge (u, v). This path has as label the concatenation of the label of the path from root to u with α'.

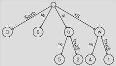
Definition 6 (string in the tree)

We say that string α is in the tree if there is a path in the tree, starting from the root, that has label α.

Notation

When a string uniquely determines a path from the root with given path-label (as is the case in suffix trees), we will identify strings and paths.
For example, we will say *string α ends at vertex u*.

Marc Hellmuth



Suffix Tree, Definitions

Definition 5 (path splits an edge)

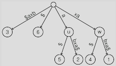
Let (u, v) be an edge of a suffix tree and the string α be its label. For a proper prefix α' of α we then say the path from root to u and α' specify together a path that splits the edge (u, v). This path has as label the concatenation of the label of the path from root to u with α'.

Definition 6 (string in the tree)

We say that string α is in the tree if there is a path in the tree, starting from the root, that has label α.

Notation

When a string uniquely determines a path from the root with given path-label (as is the case in suffix trees), we will identify strings and paths.
For example, we will say *string* α *ends at vertex* u.



Suffix Trees to Solve the Exact Substring Matching Problem

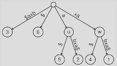
Exact Substring Matching with Suffix Trees

Input: strings P and T of lengths n, m, respectively
buid suffix tree for $T\$$ in $O(m)$ // explained later
follow the unique path from the root to find the path π with label P
if no such path exists **then**
    report that P does not occur as substring in T
else
    report the number of every leaf below the end of π as starting position of P in T

Running time

The running time of above algorithm is $O(n + m)$.
More specifically, after $O(m)$ preprocessing of T, it finds all occurences of P in T in time $O(n + k)$ where k is the number of times P occurs in T.

(Exercise: Argue, that the subtree below the end of π has $O(k)$ vertices and edges and show how its leaf set can be determined in $O(k)$ time.)



Suffix Trees to Solve the Exact Substring Matching Problem

Exact Substring Matching with Suffix Trees

Input: strings P and T of lengths n, m, respectively
buid suffix tree for $T\$$ in $O(m)$ // explained later
follow the unique path from the root to find the path π with label P
if no such path exists **then**
    report that P does not occur as substring in T
else
    report the number of every leaf below the end of π as starting position of P in T

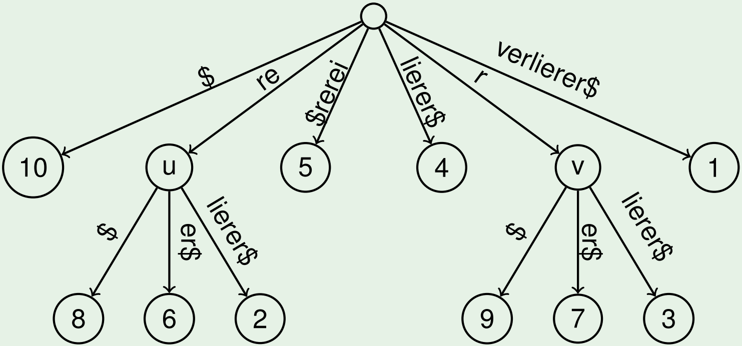
Running time

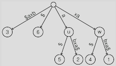
The running time of above algorithm is $O(n + m)$.
More specifically, after $O(m)$ preprocessing of T, it finds all occurences of P in T in time $O(n + k)$ where k is the number of times P occurs in T.

(Exercise: Argue, that the subtree below the end of π has $O(k)$ vertices and edges and show how its leaf set can be determined in $O(k)$ time.)

Suffix Trees to Solve the Exact Substring Matching Problem

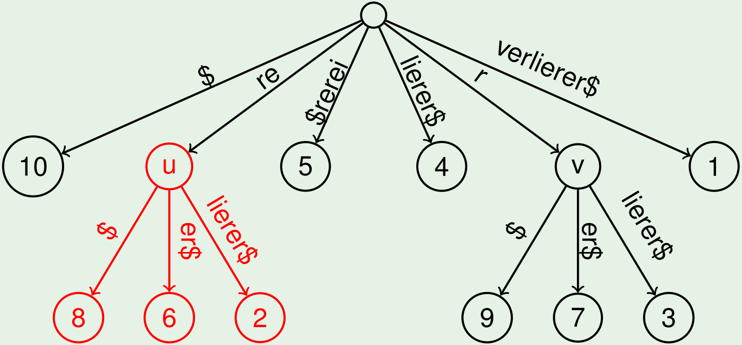
Example 7 ($T =$ `verlierer$`, $P =$ `er`)





Suffix Trees to Solve the Exact Substring Matching Problem

Example 7 (*T* = `verlierer$`, *P* = `er`)



The pattern `er` occurs at positions 2, 6 and 8 in the text. These are the labels of all leaves in the subtree below the end of the path with label `er`.

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$
- start with $\mathcal{T}_1$ which has a single edge for S from root to leaf labeled 1

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$
- start with $\mathcal{T}_1$ which has a single edge for S from root to leaf labeled 1
- when $\mathcal{T}_i$ has already been build, insert suffix $S[i+1..n]$ into $\mathcal{T}_i$:

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$
- start with $\mathcal{T}_1$ which has a single edge for S from root to leaf labeled 1
- when $\mathcal{T}_i$ has already been build, insert suffix $S[i+1..n]$ into $\mathcal{T}_i$:
    - from the root find a path with label that matches the longest possible prefix of $S[i+1..m]$

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$
- start with $\mathcal{T}_1$ which has a single edge for S from root to leaf labeled 1
- when $\mathcal{T}_i$ has already been build, insert suffix $S[i+1..n]$ into $\mathcal{T}_i$:
    - from the root find a path with label that matches the longest possible prefix of $S[i+1..m]$
    - there is a unique such path π, since the labels of all edges leaving a node start with different characters

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$
- start with $\mathcal{T}_1$ which has a single edge for S from root to leaf labeled 1
- when $\mathcal{T}_i$ has already been build, insert suffix $S[i+1..n]$ into $\mathcal{T}_i$:
    - from the root find a path with label that matches the longest possible prefix of $S[i+1..m]$
    - there is a unique such path π, since the labels of all edges leaving a node start with different characters
    - π cannot have label $S[i+1..m]$, since \$ appears at the end of each suffix

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$
- start with $\mathcal{T}_1$ which has a single edge for S from root to leaf labeled 1
- when $\mathcal{T}_i$ has already been build, insert suffix $S[i+1..n]$ into $\mathcal{T}_i$:
    - from the root find a path with label that matches the longest possible prefix of $S[i+1..m]$
    - there is a unique such path π, since the labels of all edges leaving a node start with different characters
    - π cannot have label $S[i+1..m]$, since \$ appears at the end of each suffix
    - two cases: 1) π ends at internal node w or 2) π splits an edge u and v

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$
- start with $\mathcal{T}_1$ which has a single edge for S from root to leaf labeled 1
- when $\mathcal{T}_i$ has already been build, insert suffix $S[i+1..n]$ into $\mathcal{T}_i$:
    - from the root find a path with label that matches the longest possible prefix of $S[i+1..m]$
    - there is a unique such path π, since the labels of all edges leaving a node start with different characters
    - π cannot have label $S[i+1..m]$, since \$ appears at the end of each suffix
    - two cases: 1) π ends at internal node w or 2) π splits an edge u and v
    - in case 2) insert a new node w between u and v, remove edge (u, v) and insert edges (u, w) and (w, v), label them with the prefix of (u, v)'s label that still matched and the rest of that string, respectively

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

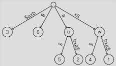
- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$
- start with $\mathcal{T}_1$ which has a single edge for S from root to leaf labeled 1
- when $\mathcal{T}_i$ has already been build, insert suffix $S[i+1..n]$ into $\mathcal{T}_i$:
  - from the root find a path with label that matches the longest possible prefix of $S[i+1..m]$
  - there is a unique such path π, since the labels of all edges leaving a node start with different characters
  - π cannot have label $S[i+1..m]$, since \$ appears at the end of each suffix
  - two cases: 1) π ends at internal node w or 2) π splits an edge u and v
  - in case 2) insert a new node w between u and v, remove edge (u, v) and insert edges (u, w) and (w, v), label them with the prefix of (u, v)'s label that still matched and the rest of that string, respectively
  - in both cases 1) and 2) insert a new edge $(w, i+1)$ labeled with the suffix of $S[i+1..m]$ that is not matched by the label of the path from root to w

Naive Algorithm to Build Suffix Tree

Naive Algorithm to Build Suffix Tree $\mathcal{T}$ *(chalk board)*

- let S be a string of length m ending in \$
- iteratively build suffix trees $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m = \mathcal{T}$, where $\mathcal{T}_i$ contains all suffixes $S[1..m], \ldots, S[i..m]$
- start with $\mathcal{T}_1$ which has a single edge for S from root to leaf labeled 1
- when $\mathcal{T}_i$ has already been build, insert suffix $S[i+1..n]$ into $\mathcal{T}_i$:
  - from the root find a path with label that matches the longest possible prefix of $S[i+1..m]$
  - there is a unique such path π, since the labels of all edges leaving a node start with different characters
  - π cannot have label $S[i+1..m]$, since \$ appears at the end of each suffix
  - two cases: 1) π ends at internal node w or 2) π splits an edge u and v
  - in case 2) insert a new node w between u and v, remove edge (u, v) and insert edges (u, w) and (w, v), label them with the prefix of (u, v)'s label that still matched and the rest of that string, respectively
  - in both cases 1) and 2) insert a new edge $(w, i+1)$ labeled with the suffix of $S[i+1..m]$ that is not matched by the label of the path from root to w
- call the new tree $\mathcal{T}_{i+1}$

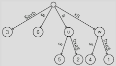
Marc Hellmuth



Naive Algorithm to Build Suffix Tree

Running Time

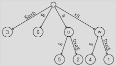
Above algorithm takes time $O(m^2)$ to build a suffix tree of a string of length m.



Ukkonen's Algorithm

Ukkonen's Algorithm

- constructs a suffix tree in linear time (Esko Ukkonen, 1995)
- will introduce the algorithm by step-wise improvements, starting from a simple, inefficient version, introducing ideas for the speedup from $O(m^3)$ to $O(m^2)$ to $O(m)$



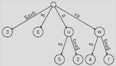
Ukkonen's Algorithm

Ukkonen's Algorithm

- constructs a suffix tree in linear time (Esko Ukkonen, 1995)
- will introduce the algorithm by step-wise improvements, starting from a simple, inefficient version, introducing ideas for the speedup from $O(m^3)$ to $O(m^2)$ to $O(m)$

Definition 8 (implicit suffix tree)

An implicit suffix tree for string S is a tree obtained from the suffix tree for $S\$$ by removing every copy of the terminal symbol \$ from the edge labels of the tree, then removing any edge that has no label, and then removing any internal node that does not have at least two children.



Ukkonen's Algorithm

Ukkonen's Algorithm

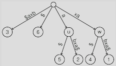
- constructs a suffix tree in linear time (Esko Ukkonen, 1995)
- will introduce the algorithm by step-wise improvements, starting from a simple, inefficient version, introducing ideas for the speedup from $O(m^3)$ to $O(m^2)$ to $O(m)$

Definition 8 (implicit suffix tree)

An implicit suffix tree for string S is a tree obtained from the suffix tree for $S\$$ by removing every copy of the terminal symbol \$ from the edge labels of the tree, then removing any edge that has no label, and then removing any internal node that does not have at least two children.

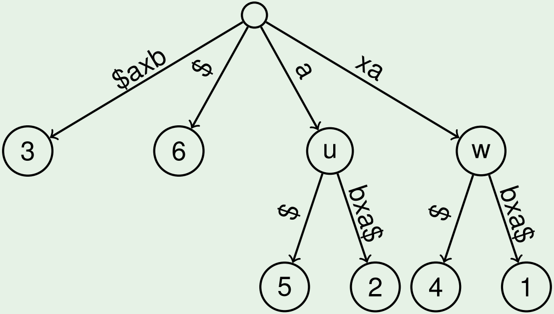
implicit suffix tree

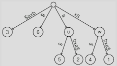
- an implicit suffix tree also contains all the suffixes of S
- but not necessarily all suffixes are the path-labels of leafs



Implicit Suffix Tree

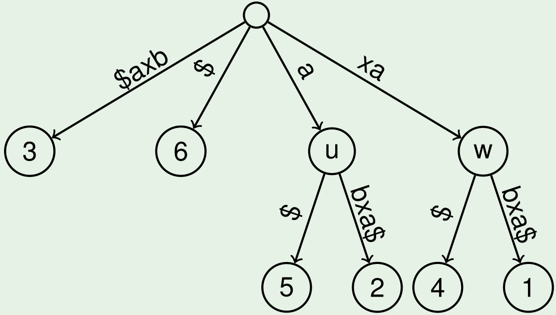
Example 9 (Suffix tree for $S = xabxa\$$)



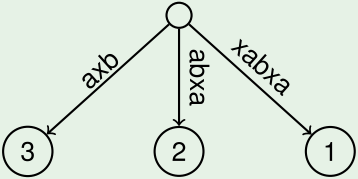


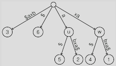
Implicit Suffix Tree

Example 9 (Suffix tree for $S = xabxa\$$)



Example 10 (Implicit suffix tree for $S = xabxa$)





Ukkonen's algorithm on a high-level

- constructs an implicit suffix tree $\mathcal{I}_i$ for prefix $S[1..i]$ for $i = 1, \ldots, m$ in that order
- the suffix tree for S is build from $\mathcal{I}_m$
- algorithm divided into m phases
- in phase $i + 1$, $\mathcal{I}_{i+1}$ is build from $\mathcal{I}_i$
  - each phase is subdivided into $i + 1$ (suffix) extensions
  - in extension j of phase $i + 1$ the algorithm finds the end of the path from the root labeled with $S[j..i]$
  - it then extends the substring by adding $S[i + 1]$ to its end if it is not already in the tree
  - extension $i + 1$ of phase $i + 1$ just puts the single-letter string $S[i + 1]$ into the tree, if it is not already there

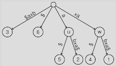
Ukkonen's Algorithm on a High-Level

Ukkonen's algorithm on a high-level

construct tree $\mathcal{I}_1$
for $i = 1$ to $m - 1$ **do**
    // phase $i + 1$ begins
    for $j = 1$ to $i + 1$ **do**
        // extension j begins
        find the end of the path π with label $S[j..i]$ from the root in the current tree
        if $S[j..i + 1]$ is not already in the tree **then**
            extend π by adding character $S[i + 1]$

Order of suffix insertions

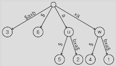
$S[1]$, (phase 1)
$S[1..2]$, $S[2..2]$, (phase 2)
$S[1..3]$, $S[2..3]$, $S[3..3]$, (phase 3)
. . .
$S[1..i]$, $S[2..i]$, ..., $S[i..i]$, (phase i)
. . .
$S[1..m]$, $S[2..m]$, ..., $S[m..m]$, (phase m)



Suffix Extension Rules

Suffix Extension Rules

In extension j of phase $i+1$ the algorithm first finds the end of $\beta := S[j..i]$ in the tree.
It then extends β and ensures that $\beta S[i+1] = S[j..i+1]$ is in the tree, according to one of the following rules:

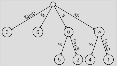


Suffix Extension Rules

Suffix Extension Rules

In extension j of phase $i+1$ the algorithm first finds the end of $\beta := S[j..i]$ in the tree.
It then extends β and ensures that $\beta S[i+1] = S[j..i+1]$ is in the tree, according to one of the following rules:

case 1 If β ends at a leaf, character $S[i+1]$ is appended to the end of the leaf edge.



Suffix Extension Rules

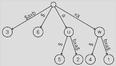
Suffix Extension Rules

In extension j of phase $i+1$ the algorithm first finds the end of $\beta := S[j..i]$ in the tree.
It then extends β and ensures that $\beta S[i+1] = S[j..i+1]$ is in the tree, according to one of the following rules:

case 1 If β ends at a leaf, character $S[i+1]$ is appended to the end of the leaf edge.

case 2 If no path from the end of β starts with $S[i+1]$, but β does not end at a leaf, then:

- If β ends inside an edge (u, v), then create a new node w between u and v, such that β ends in w.
- If β ends at a node, then let w denote that node.
- Create a new leaf labeled j and an edge from w to that leaf labeled $S[i+1]$.



Suffix Extension Rules

Suffix Extension Rules

In extension j of phase $i+1$ the algorithm first finds the end of $\beta := S[j..i]$ in the tree.
It then extends β and ensures that $\beta S[i+1] = S[j..i+1]$ is in the tree, according to one of the following rules:

case 1 If β ends at a leaf, character $S[i+1]$ is appended to the end of the leaf edge.

case 2 If no path from the end of β starts with $S[i+1]$, but β does not end at a leaf, then:

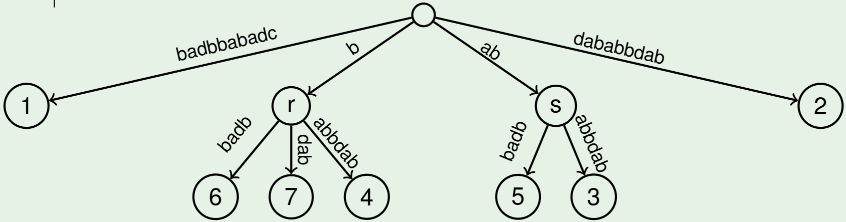
- If β ends inside an edge (u, v), then create a new node w between u and v, such that β ends in w.
- If β ends at a node, then let w denote that node.
- Create a new leaf labeled j and an edge from w to that leaf labeled $S[i+1]$.

case 3 If some path from the end of β starts with character $S[i+1]$, then do nothing.

Suffix Extension

Example 11 (Implicit suffix tree of `cdababbdab`)

	1	2	3	4	5	6	7	8	9	10
$S =$	c	d	a	b	a	b	b	d	a	b

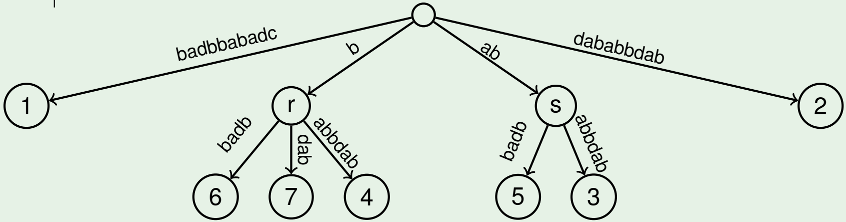


goto compressed version

Suffix Extension

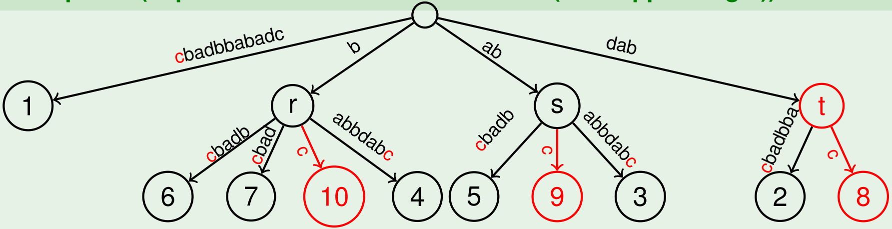
Example 11 (Implicit suffix tree of `cdababbdab`)

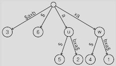
$S =$	1	2	3	4	5	6	7	8	9	10
	c	d	a	b	a	b	b	d	a	b



goto compressed version

Example 12 (Implicit suffix tree of `cdababbdabc` (after appending `c`))



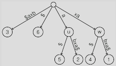


Direct Implementation of High-Level Algorithm

Direct implementation

Suppose in extension j of phase i the end of $S[j..i]$ were determined by a search of the path from the root.

- extension j of phase i would then take time $O(i-j)$

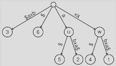


Direct Implementation of High-Level Algorithm

Direct implementation

Suppose in extension j of phase i the end of $S[j..i]$ were determined by a search of the path from the root.

- extension j of phase i would then take time $O(i-j)$
- phase i would then take time $O(i^2)$

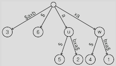


Direct Implementation of High-Level Algorithm

Direct implementation

Suppose in extension j of phase i the end of $S[j..i]$ were determined by a search of the path from the root.

- extension j of phase i would then take time $O(i - j)$
- phase i would then take time $O(i^2)$
- the whole algorithm would take time $O(m^3)$



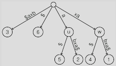
Direct Implementation of High-Level Algorithm

Direct implementation

Suppose in extension j of phase i the end of $S[j..i]$ were determined by a search of the path from the root.

- extension j of phase i would then take time $O(i - j)$
- phase i would then take time $O(i^2)$
- the whole algorithm would take time $O(m^3)$

Will reduce this to $O(m)$ with several tricks



Direct Implementation of High-Level Algorithm

Direct implementation

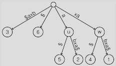
Suppose in extension j of phase i the end of $S[j..i]$ were determined by a search of the path from the root.

- extension j of phase i would then take time $O(i-j)$
- phase i would then take time $O(i^2)$
- the whole algorithm would take time $O(m^3)$

Will reduce this to $O(m)$ with several tricks

Observation

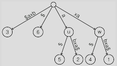
If the end of $S[j..i]$ in the tree is known then the extension of $S[i+1]$ to it can be done in constant time.
We will therefore try to speed up the search for the $S[j..i]$'s.



Suffix Links

Definition 13 (Suffix link)

Let v be an internal node with path-label $x\alpha$, where x is an arbitrary character and α an arbitrary, possibly empty, string. If there is another node $s(v)$ with path-label α, then a pointer from v to $s(v)$ is called a suffix link.

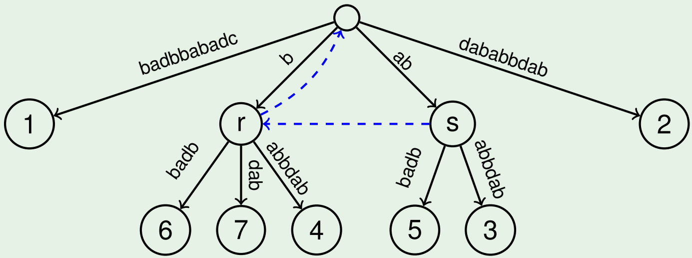


Suffix Links

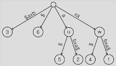
Definition 13 (Suffix link)

Let v be an internal node with path-label $x\alpha$, where x is an arbitrary character and α an arbitrary, possibly empty, string. If there is another node $s(v)$ with path-label α, then a pointer from v to $s(v)$ is called a suffix link.

Example 14 (Suffix links)



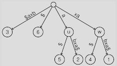
(suffix links dashed and in blue)



Suffix Links

Lemma 15 (about existence of suffix links)

If a new internal node v with path-label $x\alpha$ is added to the current tree in extension j of phase $i+1$, then after extension $j+1$ of that phase, α will end at an internal node also.



Suffix Links

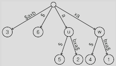
Lemma 15 (about existence of suffix links)

If a new internal node v with path-label $x\alpha$ is added to the current tree in extension j of phase $i+1$, then after extension $j+1$ of that phase, α will end at an internal node also.

Proof.

(chalk board)

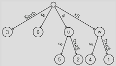
Let v be a new internal node with path-label $x\alpha$ that was added in extension j (of phase $i+1$). New internal nodes are only added in case 2 of the extension rules and we have $\alpha = S[j+1..i]$. After extension j there are at least two edges leaving v, the one to the newly created leaf and at least one other edge. Let c be the first character on that other edge. We must have $c \neq S[i+1]$ because of the suffix tree property. After extension $j+1$ the string $S[j+1..i+1] = \alpha S[i+1]$ and the string αc are both in the tree. The latter because $x\alpha c$ must have first been inserted in an earlier phase $i' \leq i$. After that phase i' also αc was in the tree. As the first disagreement between $\alpha S[i+1]$ and αc is right after α, there must be an internal node at α. □



Suffix Links

Corollary 16

In Ukkonen's algorithm, any newly created internal node will have a suffix link from it by the end of the next extension.



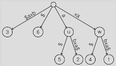
Suffix Links

Corollary 16

In Ukkonen's algorithm, any newly created internal node will have a suffix link from it by the end of the next extension.

Proof.

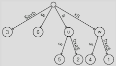
This follows from above lemma by observing that the only extension that is not followed by another extension of the same phase is the last extension of a phase, which inserts a single character and does not create an internal node. □



First Extension is a Special Case

The first extension of a phase

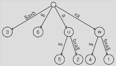
- consider extension 1 of phase $i + 1$



First Extension is a Special Case

The first extension of a phase

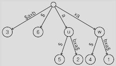
- consider extension 1 of phase $i + 1$
- inserts $S[1..i + 1]$ into the tree



First Extension is a Special Case

The first extension of a phase

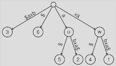
- consider extension 1 of phase $i + 1$
- inserts $S[1..i + 1]$ into the tree
- $S[1..i]$ is previously the longest string in the tree



First Extension is a Special Case

The first extension of a phase

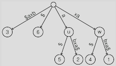
- consider extension 1 of phase $i + 1$
- inserts $S[1..i + 1]$ into the tree
- $S[1..i]$ is previously the longest string in the tree
- $\beta = S[1..i]$ therefore ends at a leaf labeled 1



First Extension is a Special Case

The first extension of a phase

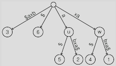
- consider extension 1 of phase $i + 1$
- inserts $S[1..i + 1]$ into the tree
- $S[1..i]$ is previously the longest string in the tree
- $\beta = S[1..i]$ therefore ends at a leaf labeled 1
- $\Rightarrow$ first extension is of suffix extension case 1



First Extension is a Special Case

The first extension of a phase

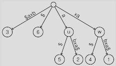
- consider extension 1 of phase $i + 1$
- inserts $S[1..i + 1]$ into the tree
- $S[1..i]$ is previously the longest string in the tree
- $\beta = S[1..i]$ therefore ends at a leaf labeled 1
- $\Rightarrow$ first extension is of suffix extension case 1
- $S[1..i + 1]$ will be a leaf again



First Extension is a Special Case

The first extension of a phase

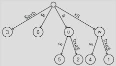
- consider extension 1 of phase $i + 1$
- inserts $S[1..i + 1]$ into the tree
- $S[1..i]$ is previously the longest string in the tree
- $\beta = S[1..i]$ therefore ends at a leaf labeled 1
- $\Rightarrow$ first extension is of suffix extension case 1
- $S[1..i + 1]$ will be a leaf again
- can be done in constant time, when a pointer to the leaf labeled 1 is maintained



Suffix Links

Using suffix links

Idea of suffix links: a shortcut so we don't have to start walking all the way from the root.



Suffix Links

Using suffix links

Idea of suffix links: a shortcut so we don't have to start walking all the way from the root.

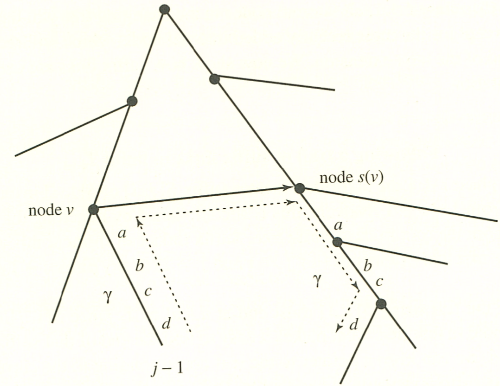


Figure 6.5: Extension $j > 1$ in phase $i+1$. Walk up atmost one edge (labeled γ) from the end of the path labeled $S[j-1..i]$ to node v; then follow the suffix link to $s(v)$; then walk down the path specifying substring γ; then apply the appropriate extension rule to insert suffix $S[j..i+1]$.

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$
3. start at end of $S[j - 1..i]$ (known from previous extension)

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$
3. start at end of $S[j - 1..i]$ (known from previous extension)
4. if not at a node which has a suffix link, walk *up* to the next node: internal or root

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$
3. start at end of $S[j - 1..i]$ (known from previous extension)
4. if not at a node which has a suffix link, walk *up* to the next node: internal or root
5. let v be the current node (walked up or not)

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$
3. start at end of $S[j - 1..i]$ (known from previous extension)
4. if not at a node which has a suffix link, walk *up* to the next node: internal or root
5. let v be the current node (walked up or not)
6. if v is the root, then find $S[j..i]$ as in the naive algorithm, continue at 11

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$
3. start at end of $S[j - 1..i]$ (known from previous extension)
4. if not at a node which has a suffix link, walk *up* to the next node: internal or root
5. let v be the current node (walked up or not)
6. if v is the root, then find $S[j..i]$ as in the naive algorithm, continue at 11
7. assume now, v is internal, with path-label $x\alpha$ and $S[j - 1..i] = x\alpha\gamma$

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$
3. start at end of $S[j - 1..i]$ (known from previous extension)
4. if not at a node which has a suffix link, walk *up* to the next node: internal or root
5. let v be the current node (walked up or not)
6. if v is the root, then find $S[j..i]$ as in the naive algorithm, continue at 11
7. assume now, v is internal, with path-label $x\alpha$ and $S[j - 1..i] = x\alpha\gamma$
8. v has a suffix link by the lemma about the existence of suffix links

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$
3. start at end of $S[j - 1..i]$ (known from previous extension)
4. if not at a node which has a suffix link, walk *up* to the next node: internal or root
5. let v be the current node (walked up or not)
6. if v is the root, then find $S[j..i]$ as in the naive algorithm, continue at 11
7. assume now, v is internal, with path-label $x\alpha$ and $S[j - 1..i] = x\alpha\gamma$
8. v has a suffix link by the lemma about the existence of suffix links
9. follow suffix link to $s(v)$, which has path-label α

Using Suffix Links

Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$
3. start at end of $S[j - 1..i]$ (known from previous extension)
4. if not at a node which has a suffix link, walk *up* to the next node: internal or root
5. let v be the current node (walked up or not)
6. if v is the root, then find $S[j..i]$ as in the naive algorithm, continue at 11
7. assume now, v is internal, with path-label $x\alpha$ and $S[j - 1..i] = x\alpha\gamma$
8. v has a suffix link by the lemma about the existence of suffix links
9. follow suffix link to $s(v)$, which has path-label α
10. follow from $s(v)$ the path labeled γ to the end of $S[j..i]$

Using Suffix Links

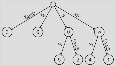
Single Extension Algorithm

1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i+1]$
3. start at end of $S[j-1..i]$ (known from previous extension)
4. if not at a node which has a suffix link, walk *up* to the next node: internal or root
5. let v be the current node (walked up or not)
6. if v is the root, then find $S[j..i]$ as in the naive algorithm, continue at 11
7. assume now, v is internal, with path-label $x\alpha$ and $S[j-1..i] = x\alpha\gamma$
8. v has a suffix link by the lemma about the existence of suffix links
9. follow suffix link to $s(v)$, which has path-label α
10. follow from $s(v)$ the path labeled γ to the end of $S[j..i]$
11. extend character $S[i+1]$ using the suffix extension rules

Using Suffix Links

Single Extension Algorithm

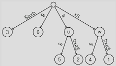
1. consider extension $j > 1$ of some phase $i + 1$
2. need to extend $S[j..i]$ to $S[j..i + 1]$
3. start at end of $S[j - 1..i]$ (known from previous extension)
4. if not at a node which has a suffix link, walk *up* to the next node: internal or root
5. let v be the current node (walked up or not)
6. if v is the root, then find $S[j..i]$ as in the naive algorithm, continue at 11
7. assume now, v is internal, with path-label $x\alpha$ and $S[j - 1..i] = x\alpha\gamma$
8. v has a suffix link by the lemma about the existence of suffix links
9. follow suffix link to $s(v)$, which has path-label α
10. follow from $s(v)$ the path labeled γ to the end of $S[j..i]$
11. extend character $S[i + 1]$ using the suffix extension rules
12. if in extension $j - 1$ a new internal node w was created (at the end of $S[j - 1..i]$, then create the suffix link from w to the end of $\alpha\gamma = S[j..i]$



Trick 1: Count Edge Label Lengths

Improving performance of path searching

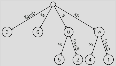
- following a path γ as in the Single Extension Algorithm (SEA) the naive way takes time proportional to $|\gamma|$



Trick 1: Count Edge Label Lengths

Improving performance of path searching

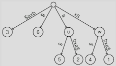
- following a path γ as in the Single Extension Algorithm (SEA) the naive way takes time proportional to $|\gamma|$
- want to improve to time proportional to the number of nodes on γ



Trick 1: Count Edge Label Lengths

Improving performance of path searching

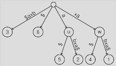
- following a path γ as in the Single Extension Algorithm (SEA) the naive way takes time proportional to $|\gamma|$
- want to improve to time proportional to the number of nodes on γ
- better, since the edge-labels can become arbitrarily long



Trick 1: Count Edge Label Lengths

Improving performance of path searching

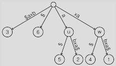
- following a path γ as in the Single Extension Algorithm (SEA) the naive way takes time proportional to $|\gamma|$
- want to improve to time proportional to the number of nodes on γ
- better, since the edge-labels can become arbitrarily long
- in the SEA we know that γ must be in the tree



Trick 1: Count Edge Label Lengths

Improving performance of path searching

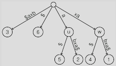
- following a path γ as in the Single Extension Algorithm (SEA) the naive way takes time proportional to $|\gamma|$
- want to improve to time proportional to the number of nodes on γ
- better, since the edge-labels can become arbitrarily long
- in the SEA we know that γ must be in the tree
- therefore: no need to compare all characters on an edge



Trick 1: Count Edge Label Lengths

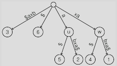
Improving performance of path searching

- following a path γ as in the Single Extension Algorithm (SEA) the naive way takes time proportional to $|\gamma|$
- want to improve to time proportional to the number of nodes on γ
- better, since the edge-labels can become arbitrarily long
- in the SEA we know that γ must be in the tree
- therefore: no need to compare all characters on an edge
- correct edge can be chosen by comparing the first character only



Count Edge Label Length Algorithm for Downwalk

input: starting node v, string γ guaranteed to be in subtree rooted at v
$h \leftarrow 0$ // the number of characters of γ matched so far
repeat
    let (v, w) be the edge for which the first character of its label β matches character $h + 1$ of γ
    $h \rightarrow h + |\beta|$
    $v \leftarrow w$
until $h \geq |\gamma|$
if $h = |\gamma|$ **then**
    γ ends at node w
else
    γ ends on the edge from w's parent to w after character $|\beta| - h + |\gamma|$ on that edge

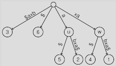


Count Edge Label Length Algorithm for Downwalk

```
input: starting node v, string γ guaranteed to be in subtree rooted at v
h ← 0 // the number of characters of γ matched so far
repeat
    let (v, w) be the edge for which the first character of its label β matches character h + 1 of γ
    h → h + |β|
    v ← w
until h ≥ |γ|
if h = |γ| then
    γ ends at node w
else
    γ ends on the edge from w's parent to w after character |β| − h + |γ| on that edge
```

Runing time

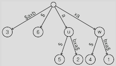
Above algorithm finds the end of γ in time proportional to the number of nodes on its path.



Suffix Links

Definition 17 (depth of a node)

The (node)-depth of a node *u* is the number of edges on the path from the root to *u*. The depth of the root is 0.
We will use the term “current depth” referring to the depth of the node last visited.



Suffix Links

Definition 17 (depth of a node)

The (node)-depth of a node u is the number of edges on the path from the root to u. The depth of the root is 0.
We will use the term "current depth" referring to the depth of the node last visited.

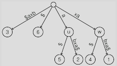
Lemma 18 (depth and suffix link)

Let $(v, s(v))$ be any suffix link traversed during Ukkonen's algorithm. At that moment, the depth of v is at most one greater than the depth of $s(v)$.

Proof.

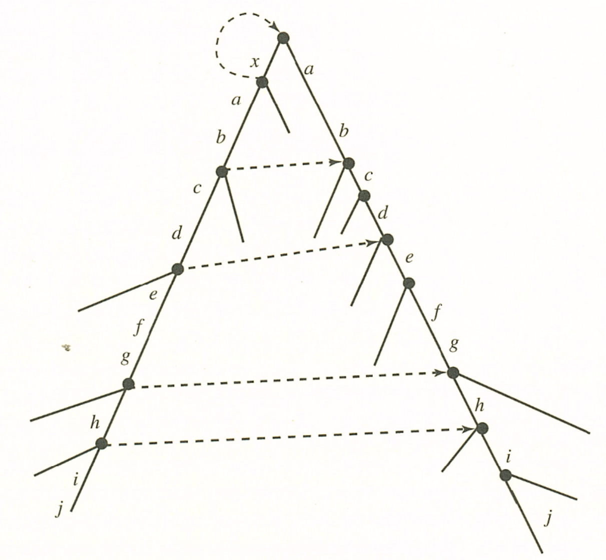
(chalk board)

□

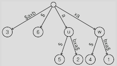


Depth and Suffix Links

Example 19



Gusfield, "Algorithms on Strings, Trees and Sequences"



Reduction to Quadratic Running Time

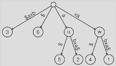
Theorem 20

Using the Count Edge Label Length Trick, any phase of Ukkonen's algorithm takes $O(m)$ time and the complete algorithm takes time $O(m^2)$.

Proof.

(chalk board)

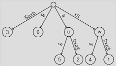
□



Storing Edge Labels

Space requirements

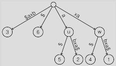
- storing all edge labels explicitly can take more than $O(m)$ space:



Storing Edge Labels

Space requirements

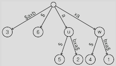
- storing all edge labels explicitly can take more than $O(m)$ space:



Storing Edge Labels

Space requirements

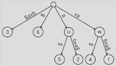
- storing all edge labels explicitly can take more than $O(m)$ space:
  consider the string `abcdefghijklmnopqrstuvwxyz` with 26 characters



Storing Edge Labels

Space requirements

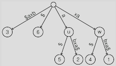
- storing all edge labels explicitly can take more than $O(m)$ space:
  consider the string `abcdefghijklmnopqrstuvwxyz` with 26 characters
  the suffix tree has 26 edges with length 1,2,…, 26 each, totalling $26 \cdot 27/2$ characters
- since an algorithm takes at least as much time as the output size, a linear-time suffix tree construction algorithm cannot use explicit edge label



Storing Edge Labels

Space requirements

- storing all edge labels explicitly can take more than $O(m)$ space:
  consider the string `abcdefghijklmnopqrstuvwxyz` with 26 characters
  the suffix tree has 26 edges with length 1,2,. . . , 26 each, totalling $26 \cdot 27/2$ characters
- since an algorithm takes at least as much time as the output size, a linear-time suffix tree construction algorithm cannot use explicit edge label
- use availability of input string S and only implicitly store edge labels=substrings of S



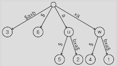
Storing Edge Labels

Space requirements

- storing all edge labels explicitly can take more than $O(m)$ space:
  consider the string `abcdefghijklmnopqrstuvwxyz` with 26 characters
  the suffix tree has 26 edges with length 1,2,. . . , 26 each, totalling $26 \cdot 27/2$ characters
- since an algorithm takes at least as much time as the output size, a linear-time suffix tree construction algorithm cannot use explicit edge label
- use availability of input string S and only implicitly store edge labels=substrings of S

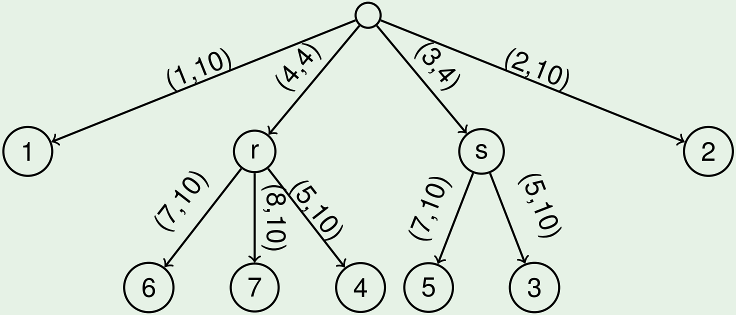
Edge-label compression

When implementing an (implicit) suffix tree of string S, store at each edge only a pair of indices:
the start and end position of a location of the edge label as substring in S.

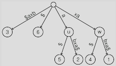


Edge label compression

Example 21 (implicit suffix tree for `cdababbdab` with edge-label compression)



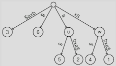
goto uncompressed version



Suffix Extension Case 3 Ends Phase

Suffix Extension Case 3

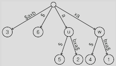
- suppose j is an extension of some phase i in which case 3 aplies, i.e. $S[j..i]$ was already in the tree before extension j



Suffix Extension Case 3 Ends Phase

Suffix Extension Case 3

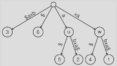
- suppose j is an extension of some phase i in which case 3 aplies, i.e. $S[j..i]$ was already in the tree before extension j
- then also $S[j+1..i]$ must already be in the tree: it was inserted to the latest in the extension after $S[j..i]$ was initially inserted



Suffix Extension Case 3 Ends Phase

Suffix Extension Case 3

- suppose j is an extension of some phase i in which case 3 aplies, i.e. $S[j..i]$ was already in the tree before extension j
- then also $S[j+1..i]$ must already be in the tree: it was inserted to the latest in the extension after $S[j..i]$ was initially inserted
- inductively, in all extensions after j also the case 3 applies



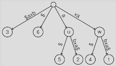
Suffix Extension Case 3 Ends Phase

Suffix Extension Case 3

- suppose j is an extension of some phase i in which case 3 aplies, i.e. $S[j..i]$ was already in the tree before extension j
- then also $S[j+1..i]$ must already be in the tree: it was inserted to the latest in the extension after $S[j..i]$ was initially inserted
- inductively, in all extensions after j also the case 3 applies

Trick 2: End phase after case 3

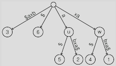
- if in extension j case 3 aplies, then end that phase
- nothing more would be done in that phase anyways as case 3 applies to all further extensions of that phase



Once a Leaf, Always a Leaf

Observation

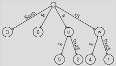
- suppose at some point during Ukkonen's algorithm a leaf labeled j is created



Once a Leaf, Always a Leaf

Observation

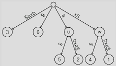
- suppose at some point during Ukkonen's algorithm a leaf labeled j is created
- the suffix extension rules never create an edge out of a leaf



Once a Leaf, Always a Leaf

Observation

- suppose at some point during Ukkonen's algorithm a leaf labeled j is created
- the suffix extension rules never create an edge out of a leaf
- therefore, after leaf j is created all extensions j of future phases will be case 1 extensions, only increasing the end position of the leaf edge label

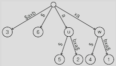


Succession of Suffix Extension Cases

Suffix extensions

Consider the cases for the suffix extension in phase $i + 1$

- some (possibly empty) rest of the extensions in any phase is of case 3, the other cases are 1 or 2: {1,2}*3*

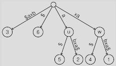


Succession of Suffix Extension Cases

Suffix extensions

Consider the cases for the suffix extension in phase $i + 1$

- some (possibly empty) rest of the extensions in any phase is of case 3, the other cases are 1 or 2: {1,2}*3*
- let j_i be the number of cases 1 or 2 from phase i ($j_1 = 1$)

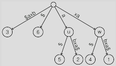


Succession of Suffix Extension Cases

Suffix extensions

Consider the cases for the suffix extension in phase $i + 1$

- some (possibly empty) rest of the extensions in any phase is of case 3, the other cases are 1 or 2: {1,2}*3*
- let j_i be the number of cases 1 or 2 from phase i ($j_1 = 1$)
- extensions $1, 2, \ldots, j_i$ of phase $i + 1$ must then be case 1:

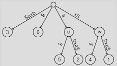


Succession of Suffix Extension Cases

Suffix extensions

Consider the cases for the suffix extension in phase $i+1$

- some (possibly empty) rest of the extensions in any phase is of case 3, the other cases are 1 or 2: {1,2}*3*
- let j_i be the number of cases 1 or 2 from phase i ($j_1 = 1$)
- extensions $1, 2, \ldots, j_i$ of phase $i+1$ must then be case 1:
    - if extension j was of case 1 in phase i then it is of case 1 in phase $i+1$ again because of the "once a leaf, always a leaf" observation

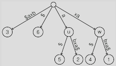


Succession of Suffix Extension Cases

Suffix extensions

Consider the cases for the suffix extension in phase $i + 1$

- some (possibly empty) rest of the extensions in any phase is of case 3, the other cases are 1 or 2: {1,2}*3*
- let j_i be the number of cases 1 or 2 from phase i ($j_1 = 1$)
- extensions $1, 2, \ldots, j_i$ of phase $i + 1$ must then be case 1:
  - if extension j was of case 1 in phase i then it is of case 1 in phase $i + 1$ again because of the "once a leaf, always a leaf" observation
  - if extension j was of case 2, then a new leaf labeled j was created in phase $i + 1$, therefore extension j is of case 1 in phase $i + 1$

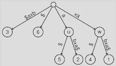


Succession of Suffix Extension Cases

Suffix extensions

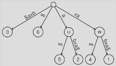
Consider the cases for the suffix extension in phase $i + 1$

- some (possibly empty) rest of the extensions in any phase is of case 3, the other cases are 1 or 2: {1,2}*3*
- let j_i be the number of cases 1 or 2 from phase i ($j_1 = 1$)
- extensions $1, 2, \ldots, j_i$ of phase $i + 1$ must then be case 1:
  - if extension j was of case 1 in phase i then it is of case 1 in phase $i + 1$ again because of the "once a leaf, always a leaf" observation
  - if extension j was of case 2, then a new leaf labeled j was created in phase $i + 1$, therefore extension j is of case 1 in phase $i + 1$
- in phase $i + 1$ the pattern of cases is therefore : 1[j_i]{1,2}*3*



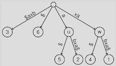
Trick 3: Store End Position Only Globally

- after phase $i + 1$ all leaf edges end at position $i + 1$



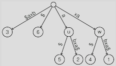
Trick 3: Store End Position Only Globally

- after phase $i + 1$ all leaf edges end at position $i + 1$
- do not update each leaf edge individually, instead consider this fact in the implementation, making all leaf edge updates in constant time,
  e.g. by storing a reference to a global variable that holds the leaf label end positions



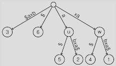
Trick 3: Store End Position Only Globally

- after phase $i + 1$ all leaf edges end at position $i + 1$
- do not update each leaf edge individually, instead consider this fact in the implementation, making all leaf edge updates in constant time,
  e.g. by storing a reference to a global variable that holds the leaf label end positions
- after entering phase $i + 1$ start extensions only at extension $j_i + 1$



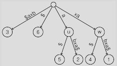
Trick 3: Store End Position Only Globally

- after phase $i+1$ all leaf edges end at position $i+1$
- do not update each leaf edge individually, instead consider this fact in the implementation, making all leaf edge updates in constant time,
  e.g. by storing a reference to a global variable that holds the leaf label end positions
- after entering phase $i+1$ start extensions only at extension $j_i + 1$
- do extensions only until the first case 3 aplies (at extension $j_{i+1} + 1$)



Single Phase Algorithm for Phase $i > 1$

1. increment global variable that holds the leaf label end positions (or similar)
2. explicitly compute extensions using the Single Extension Algorithm starting with extension $j_i + 1$ until the first case extension j^* where case 3 aplies or until all extensions of this phase are done (set $j^* := i + 1$ in this case), remember the location in the tree where the last extension ended
3. set j_{i+1} to $j^* - 1$

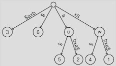


Single Phase Algorithm for Phase $i > 1$

1. increment global variable that holds the leaf label end positions (or similar)
2. explicitly compute extensions using the Single Extension Algorithm starting with extension $j_i + 1$ until the first case extension j^* where case 3 aplies or until all extensions of this phase are done (set $j^* := i + 1$ in this case), remember the location in the tree where the last extension ended
3. set j_{i+1} to $j^* - 1$

Observations

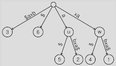
- phase $i + 1$ begins explicit extensions for the same j as the last explicit extension of the previous phase, j^*
  (e.g. the first case 3 of phase i)
- the later extension found the end of $S[j^*..i]$, which is saved
- the first extension of each phase therefore only needs constant time to extend $S[j^*..i]$ by $S[i + 1]$



Linear Running Time Result

Theorem 22

Using suffix links, tricks 1, 2 and 3 and edge label compression, above algorithm computes the implicit suffix tree of S in time $O(m)$.



Linear Running Time Result

Theorem 22

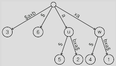
Using suffix links, tricks 1, 2 and 3 and edge label compression, above algorithm computes the implicit suffix tree of S in time $O(m)$.

Proof.

The time required for steps 1 and 3 of the Single Phase Algorithm is constant and so is $O(m)$ over the m phases.
The total number of explicit extensions is

$$\begin{aligned} &\leq 1 + (j_2 - j_1 + 1) + (j_3 - j_2 + 1) + \cdots + (j_m - j_{m-1} + 1) \\ &= j_m - j_1 + m \qquad \text{(telescope sum)} \\ &\leq 2m \end{aligned}$$

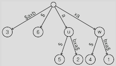
The time required for an explicit extension is constant plus time proportional to the number of nodes passed in the down-walk. The last explicit extension has the same depth as the first explicit extension of the next phase, and the first extension of each phase does not require down-walks. There are therefore at most m explicit extensions with down-walks. As the depth is bounded by m, the total number of down-walks is therefore $O(m)$. □



Constructing the Suffix Tree

Making the implicit suffix tree explicit

- append the unique termination character \$ to the end of S and run above algorithm
- the resulting implicit suffix tree will also be a suffix tree, as no suffix of $S\$$ is prefix of another suffix
- correctly set all end positions of leaf edges to $|S\$|$ by an $O(m)$ algorithm



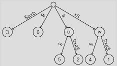
Constructing the Suffix Tree

Making the implicit suffix tree explicit

- append the unique termination character \$ to the end of S and run above algorithm
- the resulting implicit suffix tree will also be a suffix tree, as no suffix of $S\$$ is prefix of another suffix
- correctly set all end positions of leaf edges to $|S\$|$ by an $O(m)$ algorithm

Theorem 23

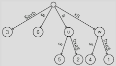
Ukkonen's algorithm builds the suffix tree of string S along with all its suffix links in $O(m)$ time.



Ukkonen's Algorithm

Example 24 (Ukkonen's Algorithm on $S = abaabbabab\$$)

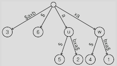
(chalk board)



Generalized Suffix Trees for Sets of Strings

Sets of strings

- have a set of strings $\{S_1, S_2, \ldots, S_z\}$
- in some applications we want to find substrings common to several or all strings in the set
- solution: a generalized suffix tree that holds suffixes of all strings in the set



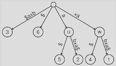
Generalized Suffix Trees for Sets of Strings

Construction of a generalized suffix tree (theoretical way)

1. construct $S = S_1\$S_2€S_3£\cdots S_z¥$,
   where \$, €, £, ..., ¥ are terminal symbols assumed to be not in any of the S_i
2. build suffix tree of S with Ukkonen's algorithm in $O(m)$ where $m := |S_1| + \cdots + |S_z|$

   (chalk board)
3. remove artificial suffixes that span more than 1 string in set, determine and store at each leaf the index $i \in \{1, \ldots, z\}$ of the string and shift sequence coordinates at leaf labels

   (chalk board)



Generalized Suffix Trees for Sets of Strings

Construction of a generalized suffix tree (theoretical way)

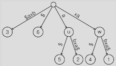
1. construct $S = S_1\$S_2€S_3£\cdots S_z¥$,
   where \$, €, £, ..., ¥ are terminal symbols assumed to be not in any of the S_i
2. build suffix tree of S with Ukkonen's algorithm in $O(m)$ where $m := |S_1| + \cdots + |S_z|$

   (*chalk board)*
3. remove artificial suffixes that span more than 1 string in set, determine and store at each leaf the index $i \in \{1, \ldots, z\}$ of the string and shift sequence coordinates at leaf labels

   (*chalk board)*

Observation

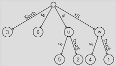
Because the terminal symbols are unique, after step 2 every *internal* node has a path label that is a substring of one or more of the S_i (no artificial substring spanning different strings).



Generalized Suffix Trees for Sets of Strings

Step 3

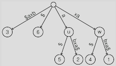
- for $i = 0, 1, \ldots, z$ let $\ell(i) = \sum_{k=1}^{i}(1 + |S_k|)$



Generalized Suffix Trees for Sets of Strings

Step 3

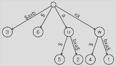
- for $i = 0, 1, \ldots, z$ let $\ell(i) = \sum_{k=1}^{i}(1 + |S_k|)$
- let $i^*(c) := \min\{i \mid \ell(i) \geq c\}$ be the index of the string that suffix starting at i "really" represents



Generalized Suffix Trees for Sets of Strings

Step 3

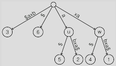
- for $i = 0, 1, \ldots, z$ let $\ell(i) = \sum_{k=1}^{i}(1 + |S_k|)$
- let $i^*(c) := \min\{i \mid \ell(i) \geq c\}$ be the index of the string that suffix starting at i "really" represents
- in the suffix tree for S relabel a leaf labeled j as $(i^*(j), j - \ell(i^*(j) - 1))$



Generalized Suffix Trees for Sets of Strings

Step 3

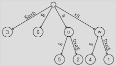
- for $i = 0, 1, \ldots, z$ let $\ell(i) = \sum_{k=1}^{i}(1 + |S_k|)$
- let $i^*(c) := \min\{i \mid \ell(i) \geq c\}$ be the index of the string that suffix starting at i "really" represents
- in the suffix tree for S relabel a leaf labeled j as $(i^*(j), j - \ell(i^*(j) - 1))$
- in the suffix tree for S relabel an edge labeled (c, d) as follows
  - $c \leftarrow c - \ell(i^*(c) - 1)$
  - $d \leftarrow \min\{d - \ell(i^*(c) - 1), |S_{i^*(c)}| + 1\}$
  - also store the index $i^*(c)$ of the string with the edge



Generalized Suffix Trees for Sets of Strings

Step 3

- for $i = 0, 1, \ldots, z$ let $\ell(i) = \sum_{k=1}^{i}(1 + |S_k|)$
- let $i^*(c) := \min\{i \mid \ell(i) \geq c\}$ be the index of the string that suffix starting at i "really" represents
- in the suffix tree for S relabel a leaf labeled j as $(i^*(j), j - \ell(i^*(j) - 1))$
- in the suffix tree for S relabel an edge labeled (c, d) as follows
  - $c \leftarrow c - \ell(i^*(c) - 1)$
  - $d \leftarrow \min\{d - \ell(i^*(c) - 1), |S_{i^*(c)}| + 1\}$
  - also store the index $i^*(c)$ of the string with the edge
- remove all but one edge from the root that start with a special end-of-string-character



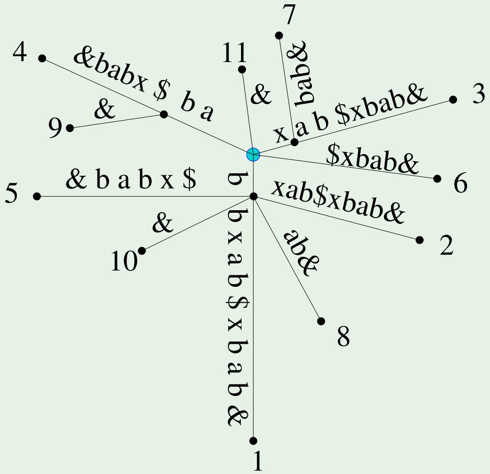
Generalized Suffix Trees for Sets of Strings

Step 3

- for $i = 0, 1, \ldots, z$ let $\ell(i) = \sum_{k=1}^{i}(1 + |S_k|)$
- let $i^*(c) := \min\{i \mid \ell(i) \geq c\}$ be the index of the string that suffix starting at i "really" represents
- in the suffix tree for S relabel a leaf labeled j as $(i^*(j), j - \ell(i^*(j) - 1))$
- in the suffix tree for S relabel an edge labeled (c, d) as follows
  - $c \leftarrow c - \ell(i^*(c) - 1)$
  - $d \leftarrow \min\{d - \ell(i^*(c) - 1), |S_{i^*(c)}| + 1\}$
  - also store the index $i^*(c)$ of the string with the edge
- remove all but one edge from the root that start with a special end-of-string-character
- step 3 requires only time in $O(m)$ if i^* is computed in constant time after preprocessing

Generalized Suffix Trees for Sets of Strings

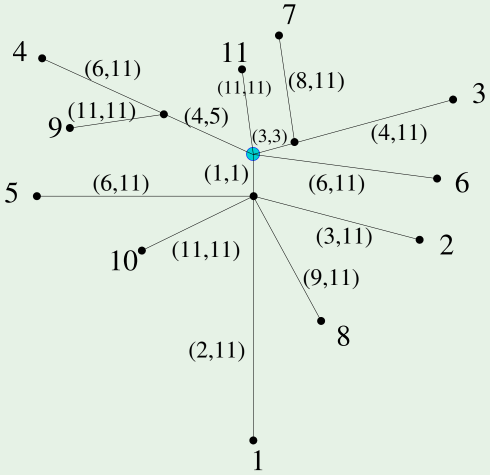
Example 25 (suffix tree for S_1 = bbxab and S_2 = xbab)



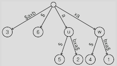
suffix tree for concatenation bbxab$xbab&

Generalized Suffix Trees for Sets of Strings

Example 25 (suffix tree for $S_1 = \text{bbxab}$ and $S_2 = \text{xbab}$)

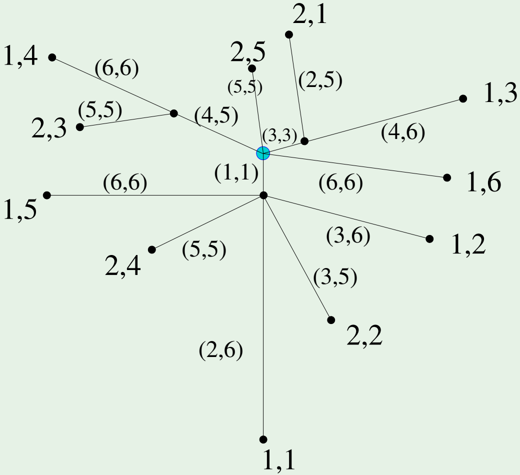


same tree in edge label compression display

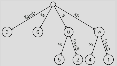


Generalized Suffix Trees for Sets of Strings

Example 25 (suffix tree for S_1 = bbxab and S_2 = xbab)

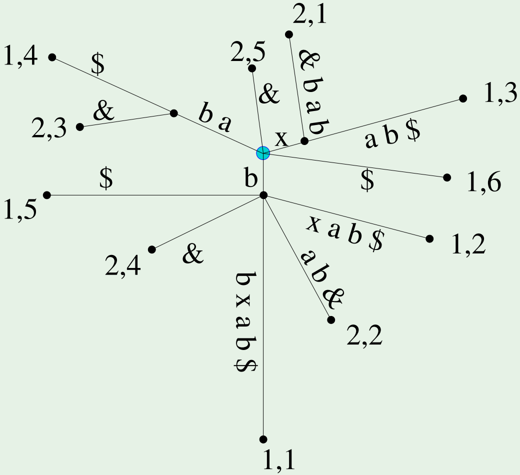


generalized suffix tree after step 3; first number at leaf marks whether suffix belongs to S_1 or S_2



Generalized Suffix Trees for Sets of Strings

Example 25 (suffix tree for $S_1 = $ bbxab and $S_2 = $ xbab)



generalized suffix tree with uncompressed edge labels