Suffix Trees

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2024

Motivation

What have we learned so far

Optimal worst-case: KMP for $O(n+m)$ pattern search,
for a pattern P of length m and text T of length n

Observation: $m \ll n$ in many applications

- mapping millions of sequenced DNA fragments to the human genome ($n > 3 \cdot 10^9$ bp)
- full text search on websites, forums, etc.
- finding motifs in a large set of sequences

Idea

Build an **index** over the text to allow very fast searches in $O(m)$ time.
Today: Suffix tries and suffix trees

Motivation: Running times

	online search	**index-based search**
Preprocessing	$O(m)$	$O(n)$
Search one pattern	$O(n)$	$O(m)$
Preprocess and search k patterns	$O(k(m+n))$	$O(n+km)$

Trees

- A **rooted tree** is a connected acyclic graph
  with a special node r, the **root node**,
  such that all edges point away from the root.
- The **depth** $depth(v)$ of a node v is its distance from the root,
  i.e. the number of edges on the unique path from the root to v.
  In particular, $depth(r) = 0$.

Edge-labeled Trees

- **Σ-tree** or **trie**: rooted tree; each edge is annotated with one single letter from Σ, such that no node has two outgoing edges labeled with the same letter.

Edge-labeled Trees

- **Σ-tree** or **trie**: rooted tree; each edge is annotated with one single letter from Σ, such that no node has two outgoing edges labeled with the same letter.
- **Σ^+-tree**: rooted tree; each edge is annotated with a non-empty string from Σ, such that no node has two outgoing edges starting with the same character.

Edge-labeled Trees

- **Σ-tree** or **trie**: rooted tree; each edge is annotated with one single letter from Σ, such that no node has two outgoing edges labeled with the same letter.
- **Σ^+-tree**: rooted tree; each edge is annotated with a non-empty string from Σ, such that no node has two outgoing edges starting with the same character.
- $string(v)$: concatenation of the edge labels on the path from the root to v.

Edge-labeled Trees

- **Σ-tree** or **trie**: rooted tree; each edge is annotated with one single letter from Σ, such that no node has two outgoing edges labeled with the same letter.
- **Σ^+-tree**: rooted tree; each edge is annotated with a non-empty string from Σ, such that no node has two outgoing edges starting with the same character.
- $\mathit{string}(v)$: concatenation of the edge labels on the path from the root to v.
- **string depth** of a node v: $\mathit{stringdepth}(v) := |\mathit{string}(v)|$.

Edge-labeled Trees

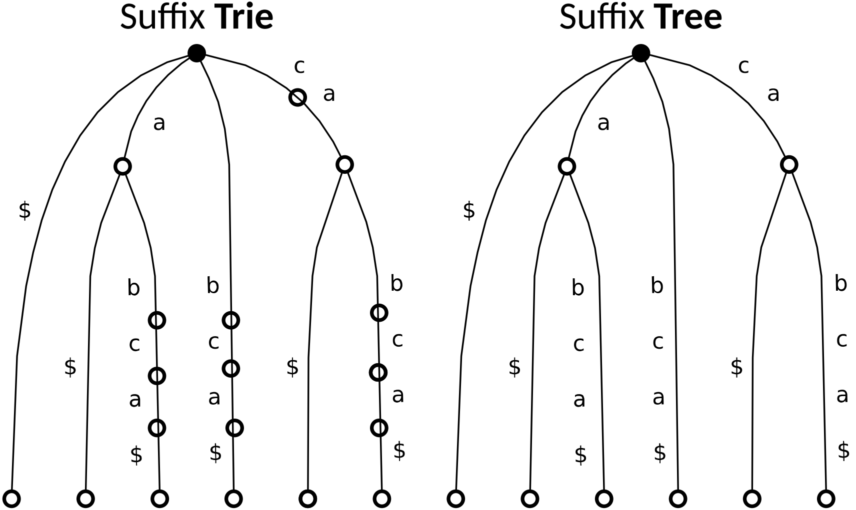
- **Σ-tree** or **trie**: rooted tree; each edge is annotated with one single letter from Σ, such that no node has two outgoing edges labeled with the same letter.
- **Σ^+-tree**: rooted tree; each edge is annotated with a non-empty string from Σ, such that no node has two outgoing edges starting with the same character.
- $string(v)$: concatenation of the edge labels on the path from the root to v.
- **string depth** of a node v: $stringdepth(v) := |string(v)|$.
- tree is **compact** if no node (other than possibly root r) has exactly one child.

Edge-labeled Trees

- **Σ-tree** or **trie**: rooted tree; each edge is annotated with one single letter from Σ, such that no node has two outgoing edges labeled with the same letter.
- **Σ^+-tree**: rooted tree; each edge is annotated with a non-empty string from Σ, such that no node has two outgoing edges starting with the same character.
- $string(v)$: concatenation of the edge labels on the path from the root to v.
- **string depth** of a node v: $stringdepth(v) := |string(v)|$.
- tree is **compact** if no node (other than possibly root r) has exactly one child.
- node with no outgoing edges is called **leaf**.

Σ-Trie vs. Compact Σ^+-Tree

(The example actually represents a true suffix trie/tree.)



Suffix Trees

- A Σ-tree or Σ^+-tree T **spells** $x \in \Sigma^*$, if x can be read along a path starting from root.
- *words*(T): set of strings spelled by T.

Suffix Trees

- A Σ-tree or Σ^+-tree T **spells** $x \in \Sigma^*$,
  if x can be read along a path starting from root.
- *words*(T)**:** set of strings spelled by T.

Suffix Tree

The **suffix tree** of $s \in \Sigma^*$ is a compact Σ^+-tree with
$\mathit{words}(T) = \{s' \mid s' \text{ is a substring of } s\}$.

Suffix Trees

- A Σ-tree or Σ^+-tree T **spells** $x \in \Sigma^*$, if x can be read along a path starting from root.
- *words*(T): set of strings spelled by T.

Suffix Tree

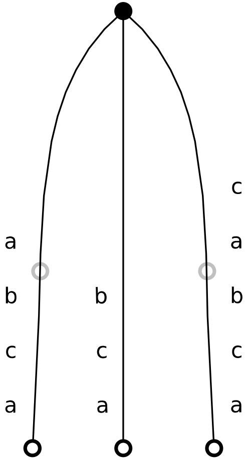
The **suffix tree** of $s \in \Sigma^*$ is a compact Σ^+-tree with $\mathit{words}(T) = \{s' \mid s' \text{ is a substring of } s\}$.

Sentinel Character

- special **sentinel character** \$ not part of Σ
- Consider the suffix tree of $s\$$ (instead of s).
- implies bijection between suffixes and leaves

Effect of the Sentinel: `cabca` vs. `cabca$`

T=cabca

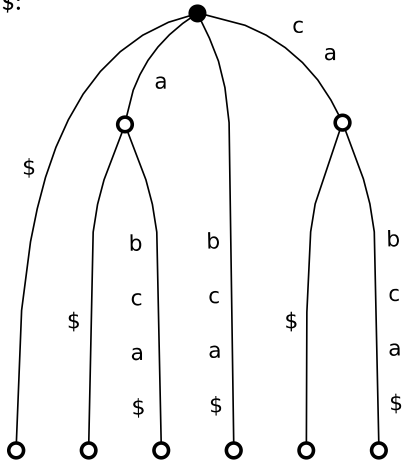


implicit suffix tree

suffixes of T$:

0	cabca$
1	abca$
2	bca$
3	ca$
4	a$
5	$

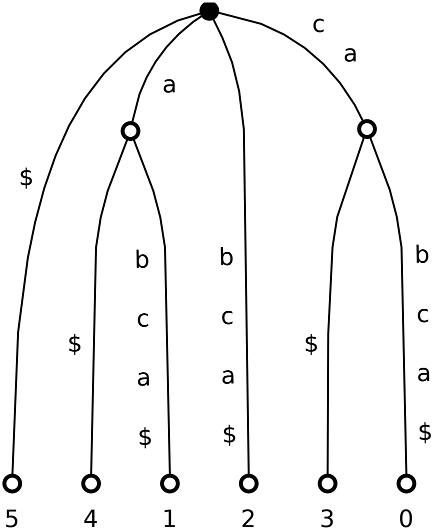
T$=cabca$



Using Suffix Trees for Pattern Search

Three variants of the search problem

1. **Decision:**
   Is P a substring of s?
2. **Counting:**
   How often does P occur in s?
3. **Enumeration:**
   Where does P occur in s?



Running Times: Using Suffix Trees for Pattern Search

The three variants of the search problem

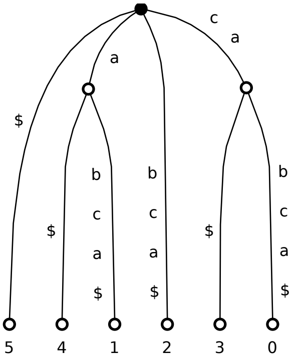
Let $m := |P|$, let and z is the number of occurrences.

1. **Decision:** Is P a substring of s?
   $\rightarrow O(m)$ time
2. **Counting:** How often does P occur in s?
   $\rightarrow O(m+z)$ time, or $O(m)$ with pre-computed counts
3. **Enumeration:** At what positions does P occur in s?
   $\rightarrow O(m+z)$ time

Applications: Longest repeated substring

Let $s \in \Sigma^*$. The suffix tree of $s\$$ spells all substrings of $s\$$.

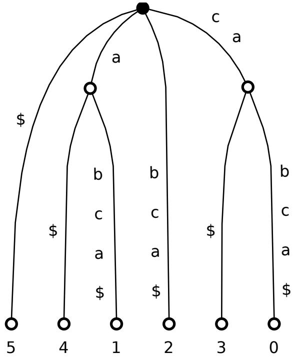
- **Question:**
  How do you find the longest repeated substring?
- **Answer:**
  A substring t of s occurs more than once,
  if after reading t from the root,
  you end in an **inner node** or on an edge above it.
  So a l.r.s. is an inner node with largest string depth.
  It can be found by a tree traversal.



Suffix tree for $s = \text{cabca\$}$

Applications: Shortest unique substring

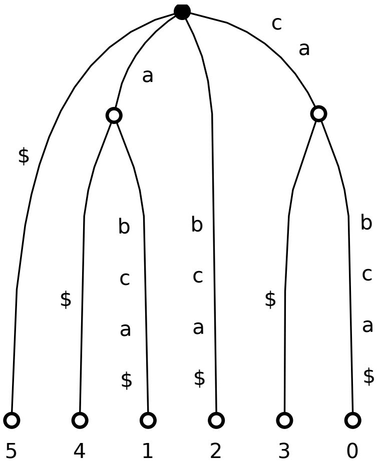
- **Question:**
  How do you find the shortest unique substring (without the sentinel)?
- **Answer:**
  Unique substrings end in a leaf edge in the tree.
  We look for an **inner node** v (including the root) with the **shortest path label** that **contains a leaf edge** that is not simply \$.
  Path label v plus the first letter on the leaf edge denotes the shortest unique substring.



Suffix tree for $s =$ cabca\$

Linear Time Suffix Tree Construction

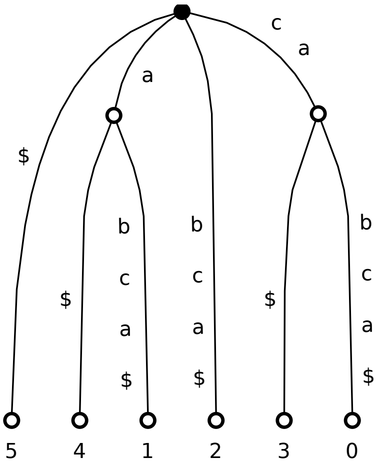
Issues to be solved



Naive implementation

- Space consumption?

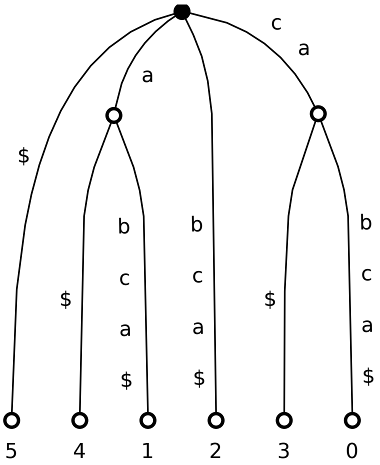
Issues to be solved



Naive implementation

- Space consumption? $O(n^2)$
- Construction time?

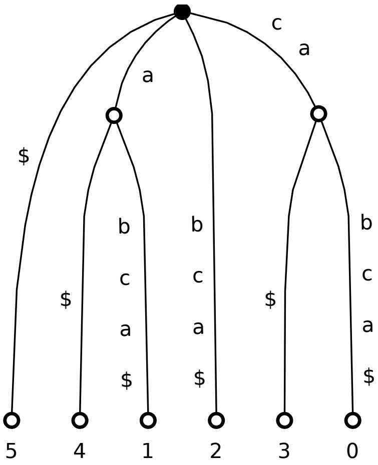
Issues to be solved



Naive implementation

- Space consumption? $O(n^2)$
- Construction time? $O(n^2)$

Issues to be solved



Naive implementation

- Space consumption? $O(n^2)$
- Construction time? $O(n^2)$

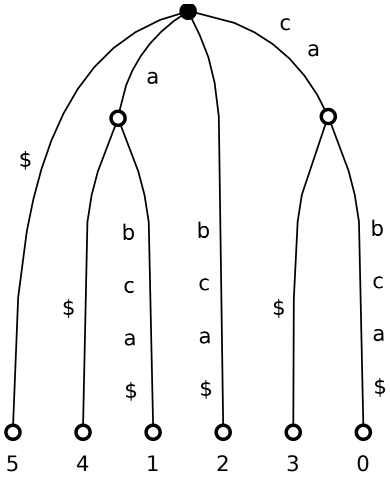
Goals

- Linear space: $O(n)$
- Linear time: $O(n)$

History of Linear-time Suffix Tree Algorithms

- Peter Weiner introduced suffix trees in 1973
  (named bi-trees at the time; “algorithm of the year)
- Edward McCreight (1976) gave the first linear-time algorithm,
  starting from longest suffixes.
- Esko Ukkonen introduced an on-line algorithm in 1992,
  later known as Ukkonen’s algorithm (we will do this one)

Number of Nodes and Edges



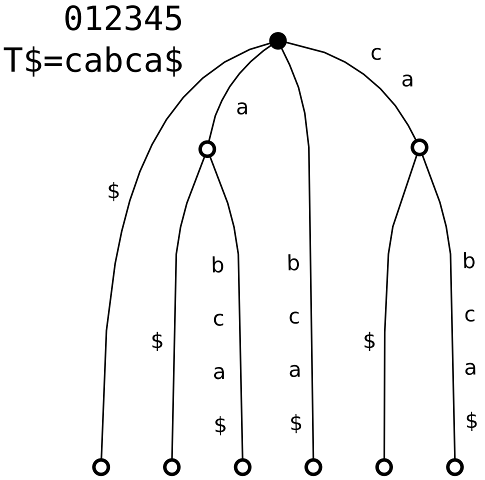
Lemma

A suffix tree of string $T\$$ with $|T\$| = n$ has exactly n leaves.
There exist at most $n - 1$ inner nodes, and at most $2(n - 1)$ edges.

Proof

Left as an exercise.

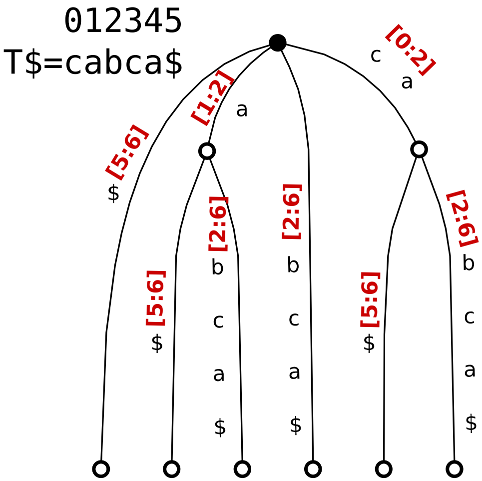
Space Consumption



Space

- Edge labels take $O(n^2)$ space ($1 + 2 + \cdots + n = n(n+1)/2$).
-

Space Consumption



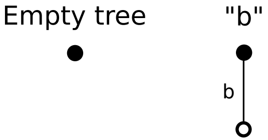
Space

- Edge labels take $O(n^2)$ space $(1 + 2 + \cdots + n = n(n+1)/2)$.
- **Indices** into T take $O(1)$ per edge, and $O(n)$ in total.

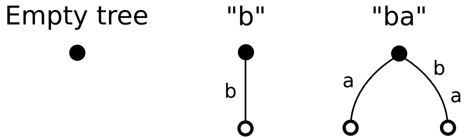
Idea: Online Construction (`babacacb$`)

Empty tree

Idea: Online Construction (`babacacb$`)

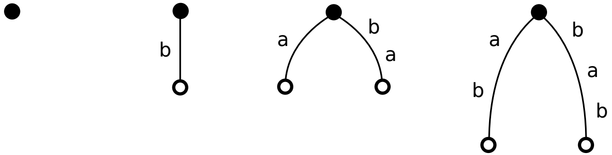


Idea: Online Construction (`babacacb$`)

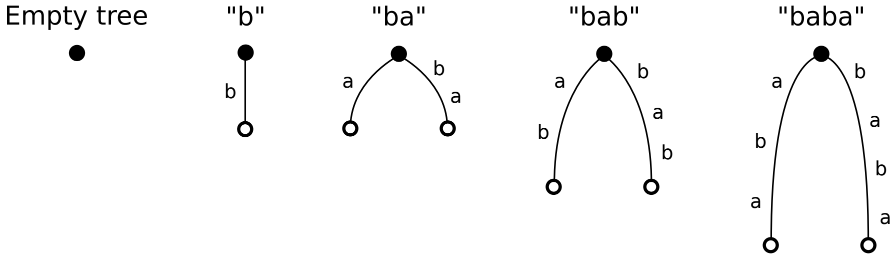


Idea: Online Construction (`babacacb$`)

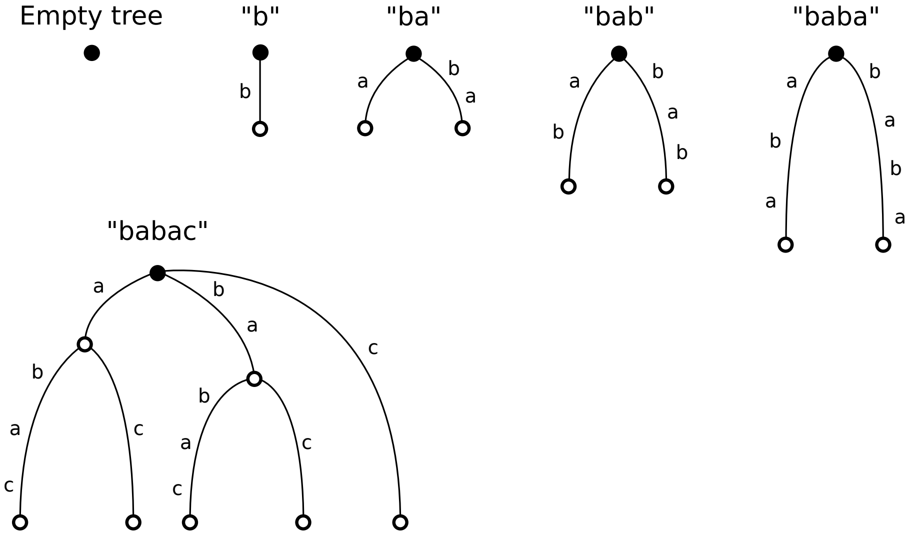
Empty tree "b" "ba" "bab"



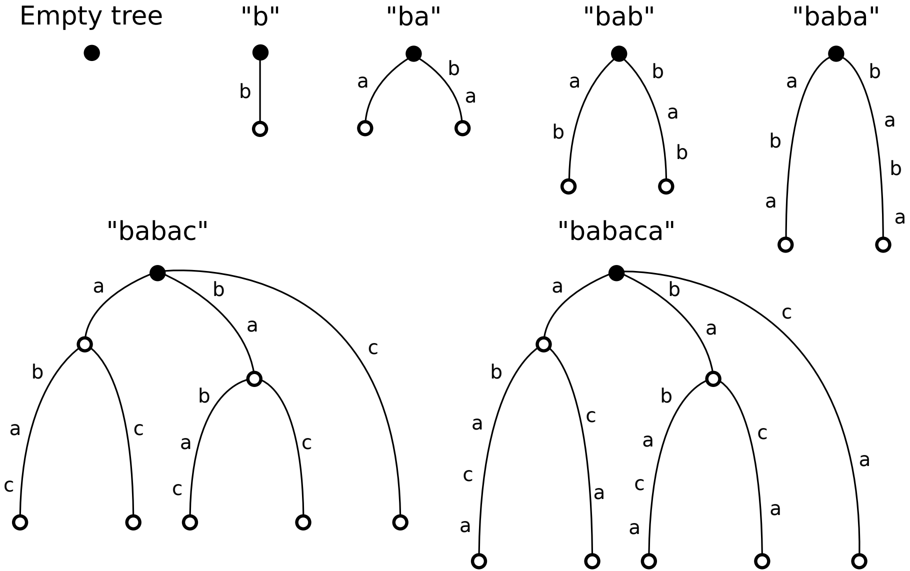
Idea: Online Construction (`babacacb$`)



Idea: Online Construction (`babacacb$`)



Idea: Online Construction (`babacacb$`)



Online Construction

Key Question

How can we achieve linear time when we extend $O(n)$ different suffixes in each step?

Online Construction

Key Question

How can we achieve linear time when we extend $O(n)$ different suffixes in each step?

Idea of Ukkonen's algorithm

In Phase i, we process $T[i]$.
This extends existing suffixes and introduces a new suffix.
Each suffix is processed according to one of 3 actions:

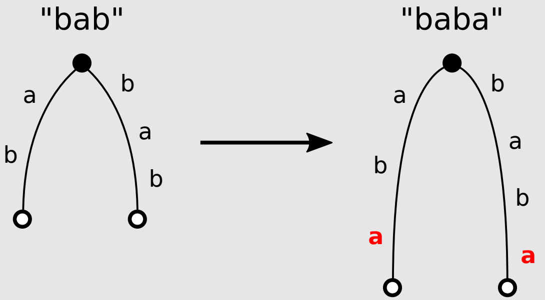
- Action 1: implicit leaf extension
- Action 2: new leaf creation
- Action 3: move along existing path

Active position after phase i:
longest suffix of $T[\dots i]$ that is a repeated substring of $T[\dots i]$

Action 1: Implicit Leaf Extension

Scenario

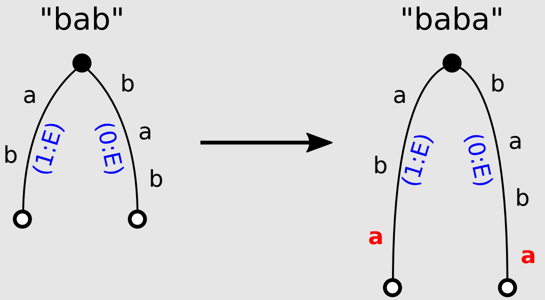
Suffix ends in a leaf.



Action 1: Implicit Leaf Extension

Scenario

Suffix ends in a leaf.



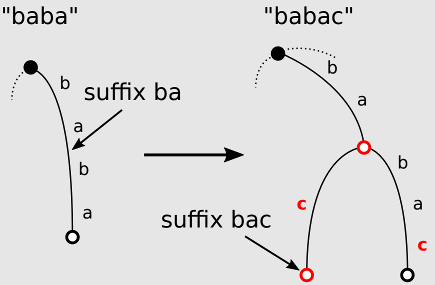
Approach

Special end marker E: substring up to the end of the current text

Action 2: New Leaf Creation

Scenario

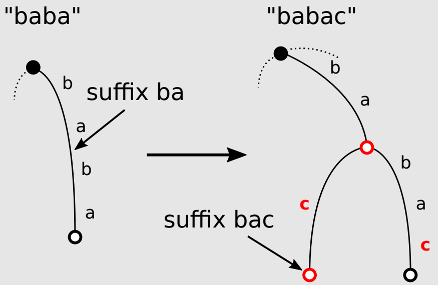
Suffix ends inside tree (edge label or at inner node),
next **character not yet present** below this position in the tree.



Action 2: New Leaf Creation

Scenario

Suffix ends inside tree (edge label or at inner node),
next **character not yet present** below this position in the tree.



Approach

Insert leaf. Create inner node if suffix ends inside edge label.

Action 3: Move Along Existing Path

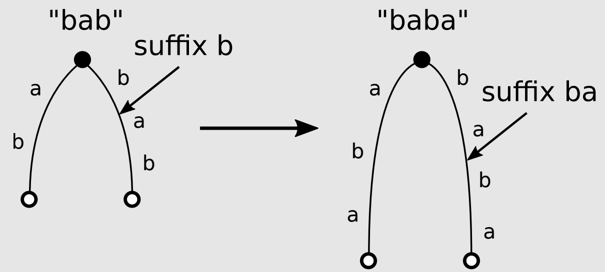
Scenario

Suffix ends inside tree (edge label or at inner node),
next **character is present** below this position in the tree.

Action 3: Move Along Existing Path

Scenario

Suffix ends inside tree (edge label or at inner node), next **character is present** below this position in the tree.



Approach

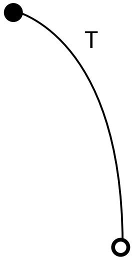
We move the active position down along the existing character.

Example of Actions

	suffix starting at					
	0	1	2	3	4	5
Phase 0	2					
Phase 1						
Phase 2						
Phase 3						
Phase 4						
Phase 5						

- Action 1: implicit leaf extension
- Action 2: new leaf creation
- Action 3: move along existing path

```
TAATA$
012345
```

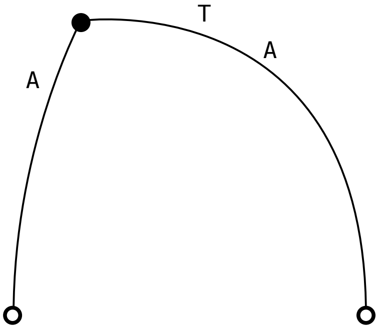


Example of Actions

	suffix starting at					
	0	1	2	3	4	5
Phase 0	2					
Phase 1	1	2				
Phase 2						
Phase 3						
Phase 4						
Phase 5						

- Action 1: implicit leaf extension
- Action 2: new leaf creation
- Action 3: move along existing path

TAATA$
012345

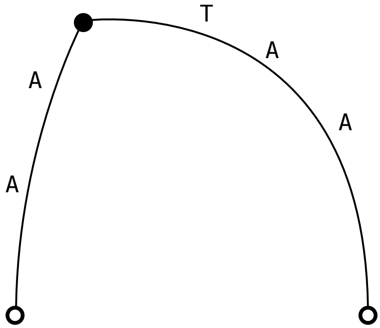


Example of Actions

	suffix starting at					
	0	1	2	3	4	5
Phase 0	2					
Phase 1	1	2				
Phase 2	1	1	3			
Phase 3						
Phase 4						
Phase 5						

- Action 1: implicit leaf extension
- Action 2: new leaf creation
- Action 3: move along existing path

TAATA$
012345

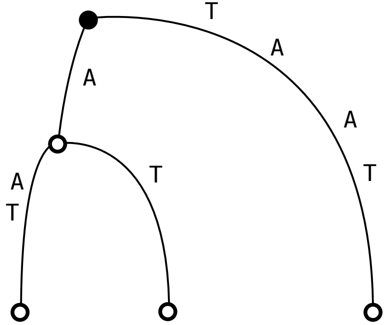


Example of Actions

	suffix starting at					
	0	1	2	3	4	5
Phase 0	2					
Phase 1	1	2				
Phase 2	1	1	3			
Phase 3	1	1	2	3		
Phase 4						
Phase 5						

- Action 1: implicit leaf extension
- Action 2: new leaf creation
- Action 3: move along existing path

TAATA$
012345

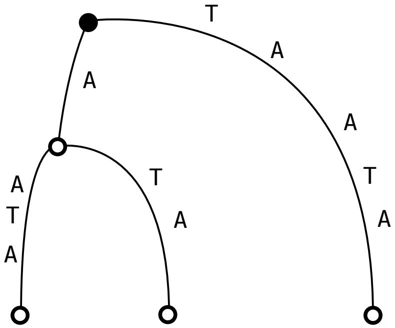


Example of Actions

	suffix starting at					
	0	1	2	3	4	5
Phase 0	2					
Phase 1	1	2				
Phase 2	1	1	3			
Phase 3	1	1	2	3		
Phase 4	1	1	1	3	3	
Phase 5						

- Action 1: implicit leaf extension
- Action 2: new leaf creation
- Action 3: move along existing path

TAATA$
012345

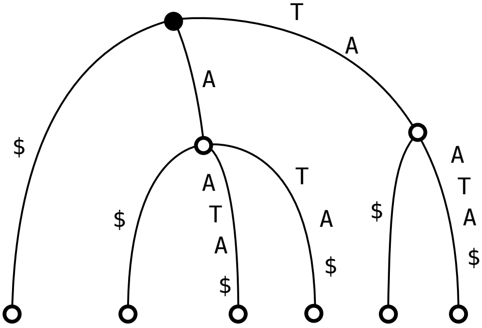


Example of Actions

	suffix starting at					
	0	1	2	3	4	5
Phase 0	2					
Phase 1	1	2				
Phase 2	1	1	3			
Phase 3	1	1	2	3		
Phase 4	1	1	1	3	3	
Phase 5	1	1	1	2	2	2

- Action 1: implicit leaf extension
- Action 2: new leaf creation
- Action 3: move along existing path

TAATA$
012345



Ukkonen's Algorithm: Open Questions

Situation

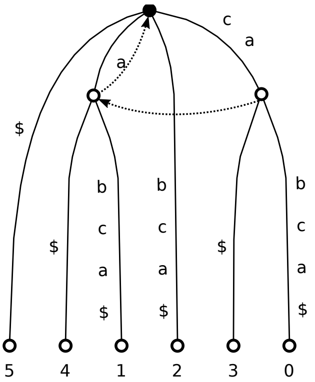
- We apply Action 2 exactly n times:
  Action 2 for the suffix starting at i is used in a phase $\geq i$.
- Action 1 does not entail any work to be done (zero time!)
- Action 3 only moves the active position down by one character ($O(1)$ time).

Missing ingredients

- When and where do we apply Action 2 for each character?
- How do we move from location to location where we apply Action 2?

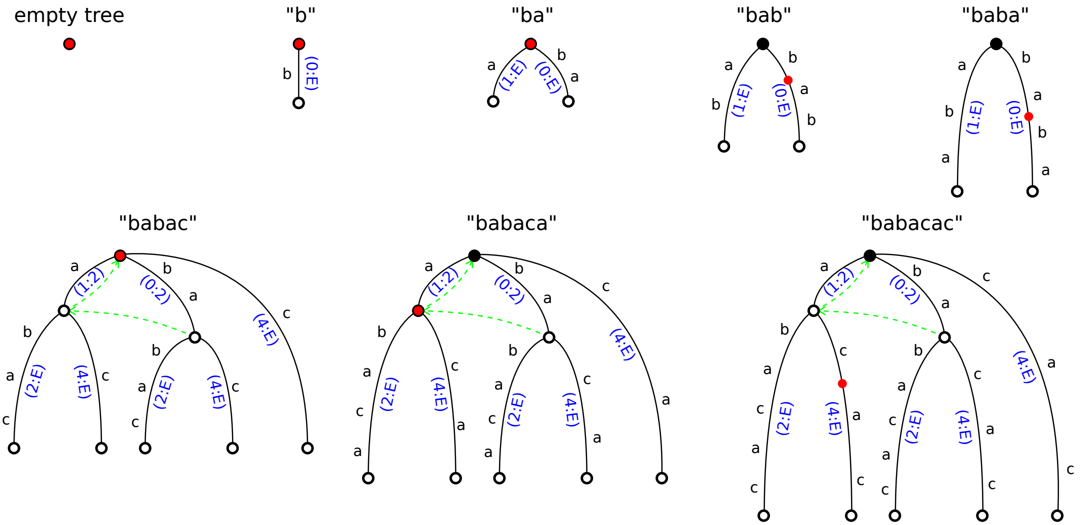
Suffix links

Suffix tree for $T = \texttt{cabca\$}$:

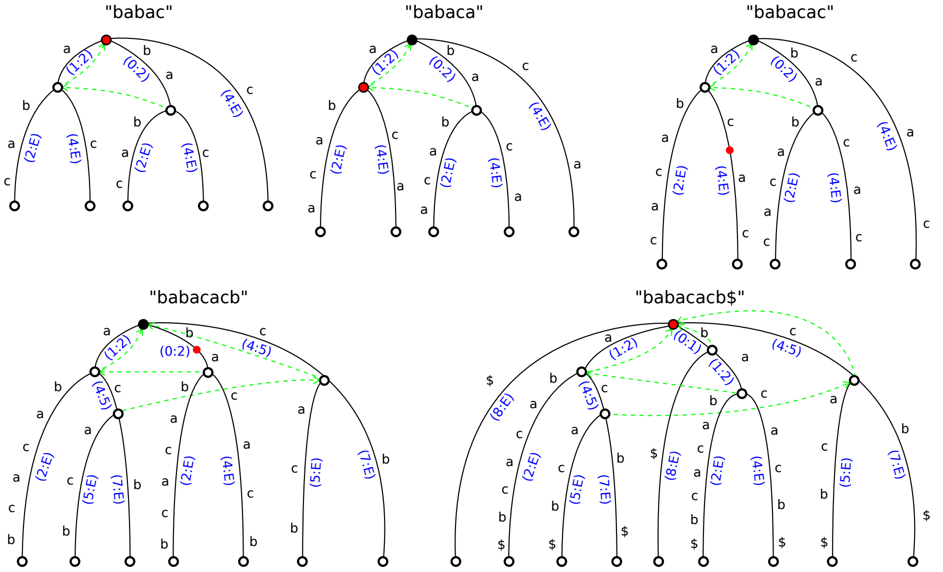


For an internal node v with path label $c\alpha$, $c \in \Sigma$, $\alpha \in \Sigma^*$,
there is another node v', with path label α (why?).
An edge $v \to v'$ (string $c\alpha \to \alpha$) is a **suffix link** ("cut off the first character").

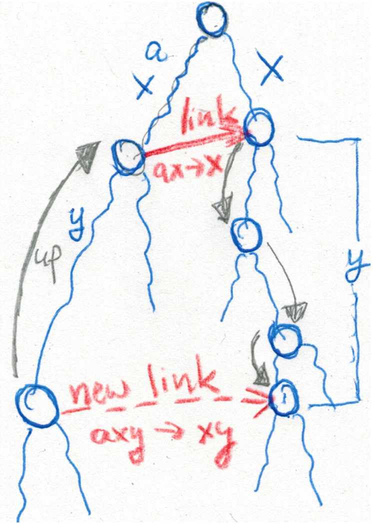
Ukkonen Example (babacacb$)



Ukkonen Example (babacacb$)



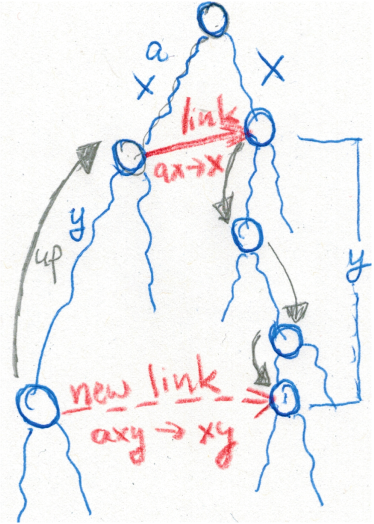
Suffix Links: Skip & Count



Skip & Count Trick

1. From active position (node axy), jump up to parent node ax, count $|y|$ in $O(1)$ time.
2. Use suffix link to x in $O(1)$ time.
3. Walk down along y, hop from node to node, skipping & counting characters in $O(h_i)$ time, with h_i: number of hops for phase i.

Suffix Links: Skip & Count



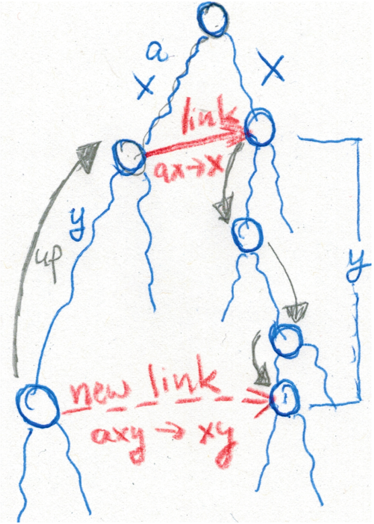
Skip & Count Trick

1. From active position (node axy), jump up to parent node ax, count $|y|$ in $O(1)$ time.
2. Use suffix link to x in $O(1)$ time.
3. Walk down along y, hop from node to node, skipping & counting characters in $O(h_i)$ time, with h_i: number of hops for phase i.

Amortized Analysis

- $h_i = O(n)$ for each phase $i \Rightarrow O(n^2)$ total.
- Need to show in fact $\sum_{i=0}^{n-1} h_i = O(n)$:

Suffix Links: Skip & Count



Skip & Count Trick

1. From active position (node axy), jump up to parent node ax, count $|y|$ in $O(1)$ time.
2. Use suffix link to x in $O(1)$ time.
3. Walk down along y, hop from node to node, skipping & counting characters in $O(h_i)$ time, with h_i: number of hops for phase i.

Amortized Analysis

- $h_i = O(n)$ for each phase $i \Rightarrow O(n^2)$ total.
- Need to show in fact $\sum_{i=0}^{n-1} h_i = O(n)$:
- Node depth cannot increase arbitrarily: $\leq n$.
- Each leaf insertion decreases depth by ≤ 1.

Ukkonen's Suffix Tree Construction Algorithm

Text $T\$$ with $n = |T\$|$: Construction uses n phases $i = 0, \ldots, n-1$.

Initialization

- Start with a root-only tree. The **active position** is the root.

Ukkonen's Suffix Tree Construction Algorithm

Text $T\$$ with $n = |T\$|$: Construction uses n phases $i = 0, \dots, n-1$.

Initialization

- Start with a root-only tree. The **active position** is the root.

Phase i with $j \leq i$ leaves already inserted

1. Apply Action 1 for each existing leaf (implicit leaf extension); no time
2. Check whether $T[i]$ already exists at the active position:
   If yes, apply Action 3, move active position down, done.
3. If not, start inserting leaves $j, j+1, \dots$ up to i or until Action 3 applies.
   To move from j to $j+1$, use existing suffix links and insert new suffix links.

Ukkonen's Suffix Tree Construction Algorithm

Text $T\$$ with $n = |T\$|$: Construction uses n phases $i = 0, \ldots, n-1$.

Initialization

- Start with a root-only tree. The **active position** is the root.

Phase i with $j \leq i$ leaves already inserted

1. Apply Action 1 for each existing leaf (implicit leaf extension); no time
2. Check whether $T[i]$ already exists at the active position:
   If yes, apply Action 3, move active position down, done.
3. If not, start inserting leaves $j, j+1, \ldots$ up to i or until Action 3 applies.
   To move from j to $j+1$, use existing suffix links and insert new suffix links.

Termination

- $T[n-1] = \$$ is unique. All missing leaves are created.
- Finally, replace end marker E by n on each edge.

Implementation Notes

Active position

The active position can be represented as a triple (v, c, ℓ),
with a node v, character c of an outgoing edge,
and number of characters $\ell \geq 0$ along that edge.

Implementation Notes

Active position

The active position can be represented as a triple (v, c, ℓ),
with a node v, character c of an outgoing edge,
and number of characters $\ell \geq 0$ along that edge.

Data structures for children of a node

Consider a node with c children, $c \leq |\Sigma|$:

	space/node	access time	total space	used for
linked list	$O(c)$	$O(c)$	$O(n)$	large alphabets
array	$O(\|\Sigma\|)$	$O(1)$	$O(n\|\Sigma\|)$	small alphabets
balanced tree	$O(c)$	$O(\log c)$	$O(n)$	large alphabets
hash table	$O(c)$	$O(1)$	$O(n)$	very large alphabets

Summary

Suffix Trees

- Definition and representation
- Applications
  - Pattern search
  - Longest repeated substring
  - Shortest unique substring
- Construction: Ukkonen's linear-time algorithm
  - substring representation on edges by indices
  - implicit zero-time edge extension by end marker E
  - suffix links
  - skip & count trick: amortized analysis
- Suffix links: useful also in other contexts

Possible Exam Questions

- Define a suffix tree. What is a suffix trie?
- Construct the suffix tree with suffix links of an example string.
- What is the running time of pattern search with a suffix tree?
- How can the longest repeated substring and the shortest unique substring be found in linear time with suffix trees?
- Explain Ukkonen's algorithm.
- How do we achieve linear space consumption in Ukkonen's algorithm?
- What is a suffix link? What are suffix links used for in Ukkonen's algorithm?
- Apply Ukkonnen's algorithm to an example string.
- Why does Ukkonen's algorithm run in $O(n)$ time?
- Explain the skip & count trick.
- Explain how one could implement the elements of a suffix tree.
  What are alternative ways of storing the children of a suffix tree node?