Indexing genomic sequences: suffix tree and suffix array

Solon P. Pissis

CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands

solon.pissis@cwi.nl

December 1, 2024

Preliminaries: Suffix Trees

Preliminaries: Suffix Trees

Let $T = T[0 \mathinner{.\,.} n-1]$ be our sequence.

Preliminaries: Suffix Trees

Let $T = T[0 \,..\, n-1]$ be our sequence.

Let $T_i = T[i \,..\, n-1]$, $i \in [0, n]$, be the ith suffix of T.

Preliminaries: Suffix Trees

Let $T = T[0 \, . \, . \, n-1]$ be our sequence.

Let $T_i = T[i \, . \, . \, n-1]$, $i \in [0, n]$, be the ith suffix of T.

For any subset $C \in [0, n]$, let $T_C = \{ T_i | i \in C \}$.

Preliminaries: Suffix Trees

Let $T = T[0\,..\,n-1]$ be our sequence.

Let $T_i = T[i\,..\,n-1]$, $i \in [0, n]$, be the ith suffix of T.

For any subset $C \in [0, n]$, let $T_C = \{T_i | i \in C\}$.

$T_{[0\,..\,n]}$ contains $n + 1$ strings.

Preliminaries: Suffix Trees

Let $T = T[0 \mathinner{.\,.} n-1]$ be our sequence.

Let $T_i = T[i \mathinner{.\,.} n-1]$, $i \in [0, n]$, be the ith suffix of T.

For any subset $C \in [0, n]$, let $T_C = \{T_i | i \in C\}$.

$T_{[0 \mathinner{.\,.} n]}$ contains $n+1$ strings.

The total length of strings in $T_{[0 \mathinner{.\,.} n]}$ is $1 + \cdots + n + 1 = \Theta(n^2)$.

Preliminaries: Suffix Trees

Let $T = T[0\,..\,n-1]$ be our sequence.

Let $T_i = T[i\,..\,n-1]$, $i \in [0, n]$, be the ith suffix of T.

For any subset $C \in [0, n]$, let $T_C = \{T_i | i \in C\}$.

$T_{[0\,..\,n]}$ contains $n+1$ strings.

The total length of strings in $T_{[0\,..\,n]}$ is $1 + \cdots + n + 1 = \Theta(n^2)$.

Suffix tree is a **compacted trie** for $T_{[0\,..\,n]}$: the set of suffixes of T.

Preliminaries: Suffix Trees

Let $T = T[0 \mathinner{.\,.} n-1]$ be our sequence.

Let $T_i = T[i \mathinner{.\,.} n-1]$, $i \in [0, n]$, be the ith suffix of T.

For any subset $C \in [0, n]$, let $T_C = \{T_i | i \in C\}$.

$T_{[0 \mathinner{.\,.} n]}$ contains $n + 1$ strings.

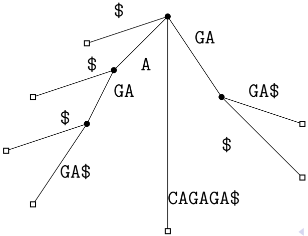
The total length of strings in $T_{[0 \mathinner{.\,.} n]}$ is $1 + \cdots + n + 1 = \Theta(n^2)$.

Suffix tree is a **compacted trie** for $T_{[0 \mathinner{.\,.} n]}$: the set of suffixes of T.

Example

Let $T = \texttt{CAGAGA\$}$. $T_1 = \texttt{AGAGA\$}$ and $T_{\{3,5\}} = \{\texttt{AGA\$}, \texttt{A\$}\}$.

\$
A\$
GA\$
AGA\$
GAGA\$
AGAGA\$
CAGAGA\$



Preliminaries: Suffix Trees

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.
- All nodes in the suffix tree representing a suffix are **leaves**.

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.
- All nodes in the suffix tree representing a suffix are **leaves**.

This simplifies algorithms!

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.
- All nodes in the suffix tree representing a suffix are **leaves**.

This simplifies algorithms!

We also decorate **leaves** with starting positions from T.

Preliminaries: Suffix Trees

We assume there is an extra **unique** letter \$ at the end of T:

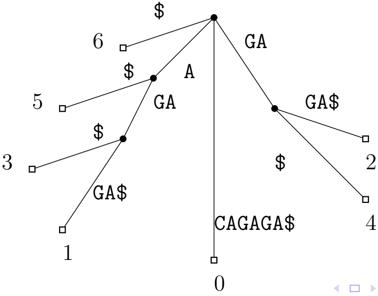
- No suffix is a prefix of another suffix: $T_{[0..n]}$ is **prefix free**.
- All nodes in the suffix tree representing a suffix are **leaves**.

This simplifies algorithms!

We also decorate **leaves** with starting positions from T.

Example

Let $T = \texttt{CAGAGA\$}$.



Preliminaries: Suffix Trees

Preliminaries: Suffix Trees

We usually assume that the suffix tree is ordered:

Preliminaries: Suffix Trees

We usually assume that the suffix tree is ordered:

- The edges of branching nodes are **sorted** by the first letter.

Preliminaries: Suffix Trees

We usually assume that the suffix tree is ordered:

- The edges of branching nodes are **sorted** by the first letter.
- $ is the lexicographically smallest letter.

Sort the edges by alphabetical order

Preliminaries: Suffix Trees

We usually assume that the suffix tree is ordered:

- The edges of branching nodes are **sorted** by the first letter.
- $ is the lexicographically smallest letter.

This simplifies searching. Think of binary search!

Preliminaries: Suffix Trees

We usually assume that the suffix tree is ordered:

- The edges of branching nodes are **sorted** by the first letter.
- \$ is the lexicographically smallest letter.

This simplifies searching. Think of binary search!

The **suffix array** of T is a sorted array of $T_{[0..n]}$.

Preliminaries: Suffix Trees

We usually assume that the suffix tree is ordered:

- The edges of branching nodes are **sorted** by the first letter.
- \$ is the lexicographically smallest letter.

This simplifies searching. Think of binary search!

The **suffix array** of T is a sorted array of $T_{[0..n]}$.

Example

Let $T =$ `CAGAGA$`. The suffix array of T is $[6, 5, 3, 1, 0, 4, 2]$.

6: `$`

5: `A$`

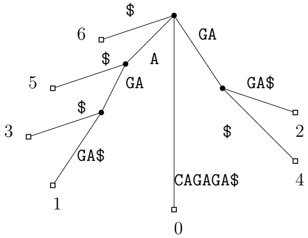
3: `AGA$`

1: `AGAGA$`

0: `CAGAGA$`

4: `GA$`

2: `GAGA$`



Preliminaries: Suffix Trees

Preliminaries: Suffix Trees

Why compacted?

Preliminaries: Suffix Trees

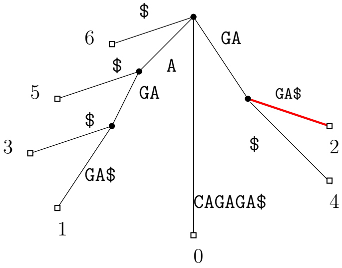
Why compacted? How much space does it take?

Preliminaries: Suffix Trees

Why compacted? How much space does it take?

Example

Let $T =$ `CAGAGA$`.

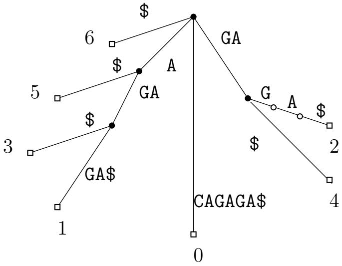


Preliminaries: Suffix Trees

Why compacted? How much space does it take?

Example

Let $T =$ `CAGAGA$`.

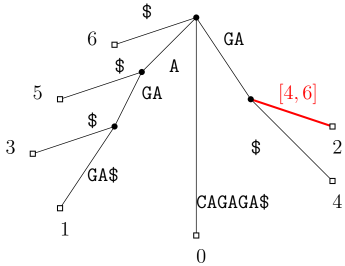


Preliminaries: Suffix Trees

Why compacted? How much space does it take?

Example

Let $T =$ `CAGAGA$`.

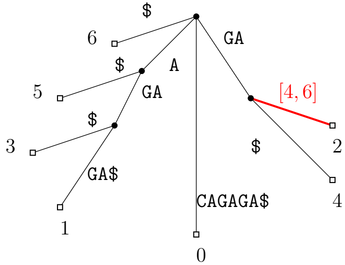


Preliminaries: Suffix Trees

Why compacted? How much space does it take?

Example

Let $T = \texttt{CAGAGA\$}$.



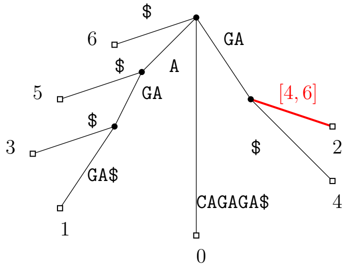
- Edge labels are substrings of T: represented by T intervals.

Preliminaries: Suffix Trees

Why compacted? How much space does it take?

Example

Let $T =$ `CAGAGA$`.



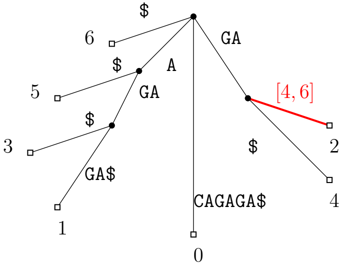
- Edge labels are substrings of T: represented by T intervals.
- Exactly $n + 1$ leaves and at most n internal nodes.

Preliminaries: Suffix Trees

Why compacted? How much space does it take?

Example

Let $T = \texttt{CAGAGA\$}$.



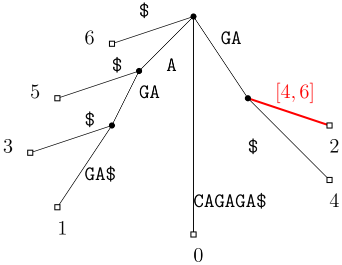
- Edge labels are substrings of T: represented by T intervals.
- Exactly $n + 1$ leaves and at most n internal nodes.
- At most $2n$ edges.

Preliminaries: Suffix Trees

Why compacted? How much space does it take?

Example

Let $T = \texttt{CAGAGA\$}$.



- Edge labels are substrings of T: represented by T intervals.
- Exactly $n + 1$ leaves and at most n internal nodes.
- At most $2n$ edges.

Space linear in n: $O(n)$.

Preliminaries: Suffix Trees

What about construction?

Preliminaries: Suffix Trees

What about construction?

Theorem (Farach, FOCS 1997)

Let T be a string of length n over a linearly-sortable alphabet. The suffix tree of T can be constructed in $O(n)$ time.

Preliminaries: Suffix Trees

What about construction?

Theorem (Farach, FOCS 1997)

Let T be a string of length n over a linearly-sortable alphabet. The suffix tree of T can be constructed in $O(n)$ time.

Linearly-sortable alphabet: $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$.

Preliminaries: Suffix Trees

What about construction?

Theorem (Farach, FOCS 1997)

Let T be a string of length n over a linearly-sortable alphabet. The suffix tree of T can be constructed in $O(n)$ time.

Linearly-sortable alphabet: $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$.

Farach's algorithm is in fact optimal for all alphabets!

Preliminaries: Suffix Trees

What about construction?

Theorem (Farach, FOCS 1997)

Let T be a string of length n over a linearly-sortable alphabet. The suffix tree of T can be constructed in $O(n)$ time.

Linearly-sortable alphabet: $\Sigma = \{1, 2, \ldots, n^{O(1)}\}$.

Farach's algorithm is in fact optimal for all alphabets!

In bioinformatics we usually have that $|\Sigma| = O(1)$.

Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

- Let $\mathrm{str}(u)$ denote the string represented by node u.

Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

- Let $\mathrm{str}(u)$ denote the string represented by node u.
- $\mathrm{slink}(u) = v \iff \mathrm{str}(u) = T[i\mathinner{.\,.}j]$ and $\mathrm{str}(v) = T[i+1\mathinner{.\,.}j]$.

Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

- Let str(u) denote the string represented by node u.
- $\text{slink}(u) = v \iff \text{str}(u) = T[i\,..\,j]$ and $\text{str}(v) = T[i+1\,..\,j]$.
- Constructible for all **internal nodes** in $O(n)$ time.

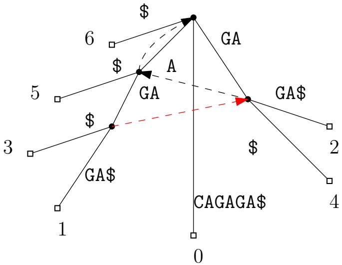
Preliminaries: Suffix Trees

An important auxiliary tool are **suffix links**:

- Let str(u) denote the string represented by node u.
- $\text{slink}(u) = v \iff \text{str}(u) = T[i\,..\,j]$ and $\text{str}(v) = T[i+1\,..\,j]$.
- Constructible for all **internal nodes** in $O(n)$ time.

Example

Let $T =$ `CAGAGA$`. str(u) = `AGA` and str(v) = `GA`.



Application 1: Exact string matching

PREPROCESS: a sequence T
QUERY: a pattern P; return all **occ** starting positions of P in T

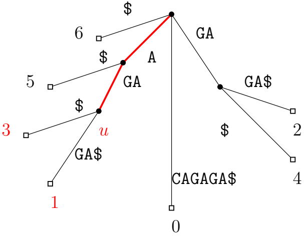
Application 1: Exact string matching

PREPROCESS: a sequence T
QUERY: a pattern P; return all **occ** starting positions of P in T

Example

Let $T = \texttt{CAGAGA\$}$ and $P = \texttt{AGA}$. Spell P and arrive at u.

6: `$`
5: `A$`
3: `AGA$`
1: `AGAGA$`
0: `CAGAGA$`
4: `GA$`
2: `GAGA$`



Application 1: Exact string matching

PREPROCESS: a sequence T

QUERY: a pattern P; return all **occ** starting positions of P in T

Example

Let $T = \texttt{CAGAGA\$}$ and $P = \texttt{AGA}$. Spell P and arrive at u.

6: `$`

5: `A$`

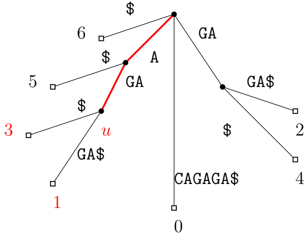
3: `AGA$`

1: `AGAGA$`

0: `CAGAGA$`

4: `GA$`

2: `GAGA$`



Traverse the subtree of **occ** leaves rooted at u.

Application 1: Exact string matching

PREPROCESS: a sequence T
QUERY: a pattern P; return all **occ** starting positions of P in T

Example

Let $T = \texttt{CAGAGA\$}$ and $P = \texttt{AGA}$. Spell P and arrive at u.

6: `$`

5: `A$`

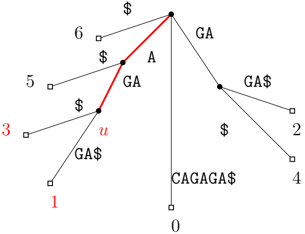
3: `AGA$`

1: `AGAGA$`

0: `CAGAGA$`

4: `GA$`

2: `GAGA$`



Traverse the subtree of **occ** leaves rooted at u. Its size is $O(\text{occ})$.

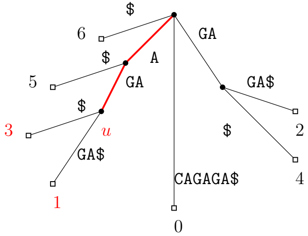
Application 1: Exact string matching

PREPROCESS: a sequence T
QUERY: a pattern P; return all **occ** starting positions of P in T

Example

Let $T = \texttt{CAGAGA\$}$ and $P = \texttt{AGA}$. Spell P and arrive at u.

6: `$`
5: `A$`
3: `AGA$`
1: `AGAGA$`
0: `CAGAGA$`
4: `GA$`
2: `GAGA$`



Traverse the subtree of **occ** leaves rooted at u. Its size is $O(\text{occ})$.

Theorem

Exact string matching queries can be answered in $O(|P| + occ)$ time after $O(n)$ time preprocessing.

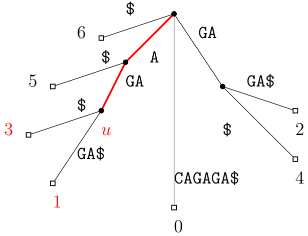
Application 1: Exact string matching

PREPROCESS: a sequence T
QUERY: a pattern P; return all **occ** starting positions of P in T

Example

Let T = `CAGAGA$` and P = `AGA`. Spell P and arrive at u.

6: `$`
5: `A$`
3: `AGA$`
1: `AGAGA$`
0: `CAGAGA$`
4: `GA$`
2: `GAGA$`



Alternatively: binary search for P in the suffix array of T.

Theorem

Exact string matching queries can be answered in $O(|P| \log n + occ)$ *time using the suffix array.*

Application 2: Number of distinct substrings

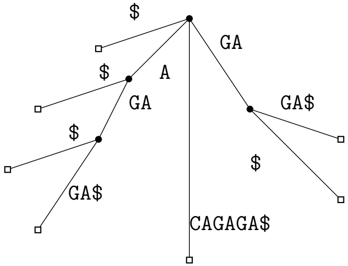
INPUT: a sequence T
OUTPUT: the number of distinct substrings

Application 2: Number of distinct substrings

INPUT: a sequence T
OUTPUT: the number of distinct substrings

Example

Let $T = \texttt{CAGAGA\$}$.

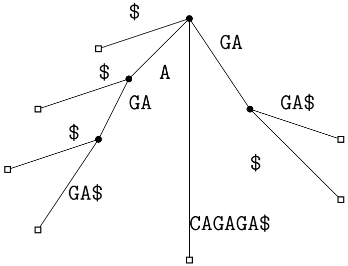


Application 2: Number of distinct substrings

INPUT: a sequence T
OUTPUT: the number of distinct substrings

Example

Let $T = \texttt{CAGAGA\$}$.



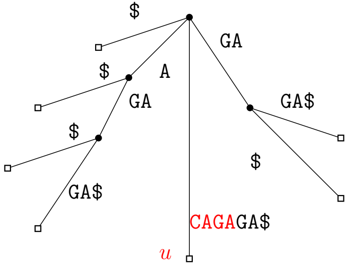
Every *locus* **(node, depth)** in the suffix tree represents a substring of the sequence and every substring is represented by some locus.

Application 2: Number of distinct substrings

INPUT: a sequence T
OUTPUT: the number of distinct substrings

Example

Let $T =$ `CAGAGA$`. Locus $(u, 4)$ represents `CAGA`.



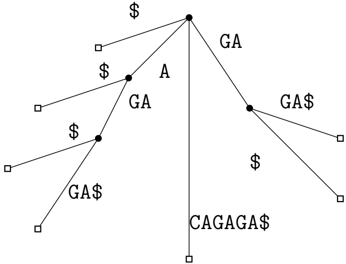
Every *locus* **(node, depth)** in the suffix tree represents a substring of the sequence and every substring is represented by some locus.

Application 2: Number of distinct substrings

INPUT: a sequence T
OUTPUT: the number of distinct substrings

Example

Let $T =$ `CAGAGA$`.

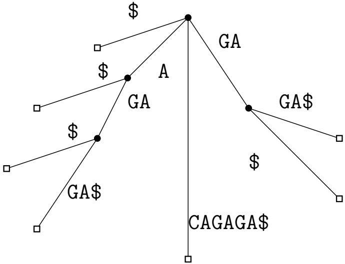


Application 2: Number of distinct substrings

INPUT: a sequence T
OUTPUT: the number of distinct substrings

Example

Let $T =$ `CAGAGA$`.



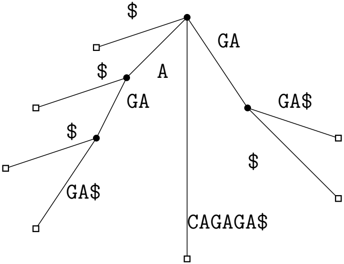
Count the number of distinct loci using a suffix tree traversal.

Application 2: Number of distinct substrings

INPUT: a sequence T
OUTPUT: the number of distinct substrings

Example

Let $T = \texttt{CAGAGA\$}$.



Theorem

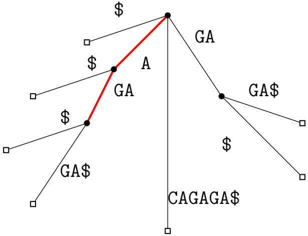
The number of distinct substrings can be computed in $O(n)$ time.

Application 3: Longest repeating substring

INPUT: a sequence T
OUTPUT: a longest string occurring at least twice in T

Example

Let $T =$ `CAGAGA$`. The answer is `AGA`.



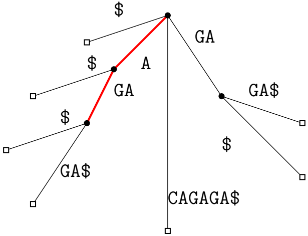
Find a deepest internal node using a traversal of the suffix tree.

Application 3: Longest repeating substring

INPUT: a sequence T
OUTPUT: a longest string occurring at least twice in T

Example

Let $T =$ `CAGAGA$`. The answer is `AGA`.



Find a deepest internal node using a traversal of the suffix tree.

Theorem

A longest repeating substring can be found in $O(n)$ time.

Application 4: Longest common substring

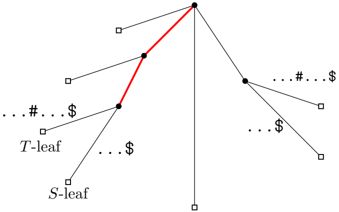
INPUT: a sequence T and a sequence S
OUTPUT: a longest common substring

Application 4: Longest common substring

INPUT: a sequence T and a sequence S
OUTPUT: a longest common substring

Example

Suffix tree of $T\#S\$$.

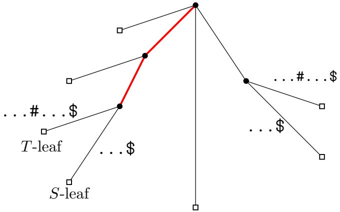


Application 4: Longest common substring

INPUT: a sequence T and a sequence S
OUTPUT: a longest common substring

Example

Suffix tree of $T\#S\$$.



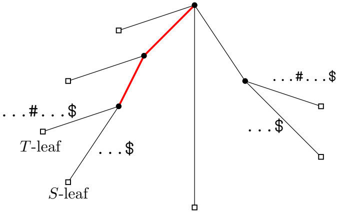
Find a deepest internal node containing both T- and S-leaves.

Application 4: Longest common substring

INPUT: a sequence T and a sequence S
OUTPUT: a longest common substring

Example

Suffix tree of $T\#S\$$.



Find a deepest internal node containing both T- and S-leaves.

Theorem

A longest common substring can be found in $O(n + |S|)$ *time.*

Application 5: Matching statistics

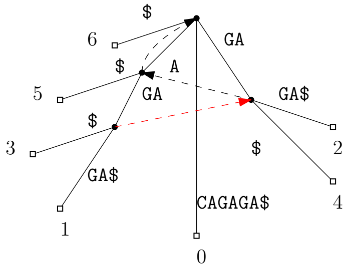
PREPROCESS: a sequence T
QUERY: a sequence S; return the longest prefix of S_i that is a substring of T, for all $i \in [0, |S| - 1]$

Application 5: Matching statistics

PREPROCESS: a sequence T
QUERY: a sequence S; return the longest prefix of S_i that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T = \texttt{CAGAGA\$}$.



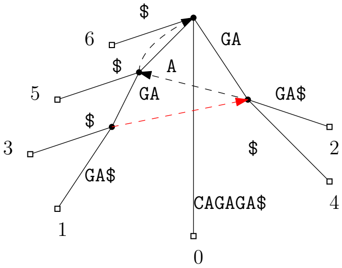
Scan S using the suffix tree of T.

Application 5: Matching statistics

PREPROCESS: a sequence T
QUERY: a sequence S; return the longest prefix of S_i that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T = \texttt{CAGAGA\$}$.



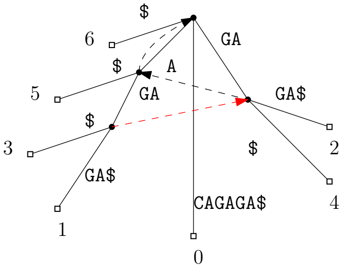
Spell S_i as much as possible;

Application 5: Matching statistics

PREPROCESS: a sequence T
QUERY: a sequence S; return the longest prefix of S_i that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T = \texttt{CAGAGA\$}$.



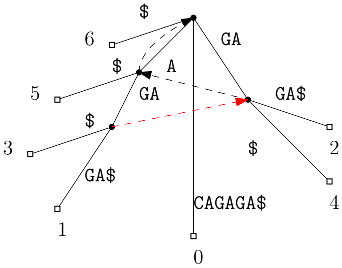
Spell S_i as much as possible; say $S[i \mathinner{.\,.} j-1]$.

Application 5: Matching statistics

PREPROCESS: a sequence T
QUERY: a sequence S; return the longest prefix of S_i that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T =$ `CAGAGA$`.



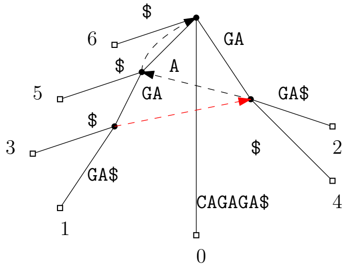
Mismatch at $S[i\,.\,.\,j]$?

Application 5: Matching statistics

PREPROCESS: a sequence T
QUERY: a sequence S; return the longest prefix of S_i that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T = \texttt{CAGAGA\$}$.



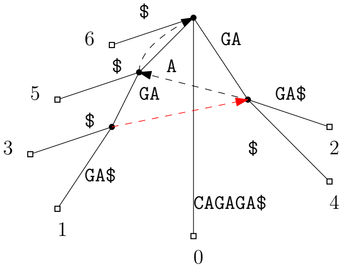
Mismatch at $S[i\,..\,j]$? Use suffix link as the failure transition!

Application 5: Matching statistics

PREPROCESS: a sequence T
QUERY: a sequence S; return the longest prefix of S_i that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T =$ `CAGAGA$`.



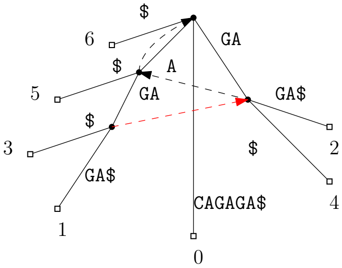
This takes us at node u: $\text{str}(u) = S[i+1\,..\,j]$.

Application 5: Matching statistics

PREPROCESS: a sequence T
QUERY: a sequence S; return the longest prefix of S_i that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T = \texttt{CAGAGA\$}$.



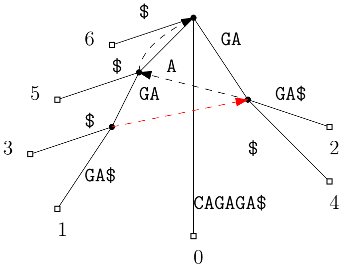
This takes us at node u: $\text{str}(u) = S[i + 1 \mathinner{.\,.} j]$. Repeat from here!

Application 5: Matching statistics

PREPROCESS: a sequence T
QUERY: a sequence S; return the longest prefix of S_i that is a substring of T, for all $i \in [0, |S| - 1]$

Example

Let $T = \texttt{CAGAGA\$}$.



Theorem

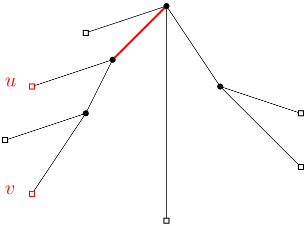
Matching statistics of S with respect to T can be computed in $O(|S|)$ time after $O(n)$ time preprocessing.

Application 6: Longest common prefix (LCP)

PREPROCESS: a sequence T
QUERY: a pair (i, j); return the length of the LCP of (T_i, T_j)

Application 6: Longest common prefix (LCP)

PREPROCESS: a sequence T
QUERY: a pair (i, j); return the length of the LCP of (T_i, T_j)
The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v.

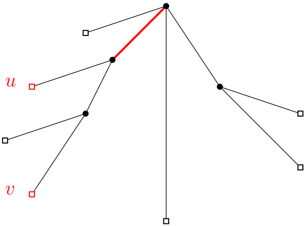


Application 6: Longest common prefix (LCP)

PREPROCESS: a sequence T

QUERY: a pair (i, j); return the length of the LCP of (T_i, T_j)

The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v.



Theorem (Bender and Farach-Colton, LATIN 2000)

Any tree of size $O(N)$ can be preprocessed in $O(N)$ time so that the LCA of any two nodes can be computed in $O(1)$ time.

Application 6: Longest common prefix (LCP)

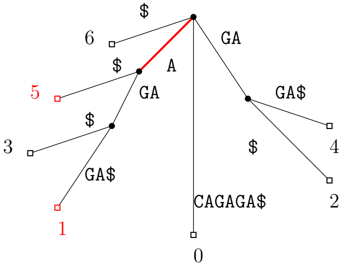
PREPROCESS: a sequence T
QUERY: a pair (i, j); return the length of the LCP of (T_i, T_j)

Application 6: Longest common prefix (LCP)

PREPROCESS: a sequence T
QUERY: a pair (i, j); return the length of the LCP of (T_i, T_j)

Example

Let $T = \texttt{CAGAGA\$}$. Let $(1, 5)$ be the query. The answer is $1 = |\texttt{A}|$.

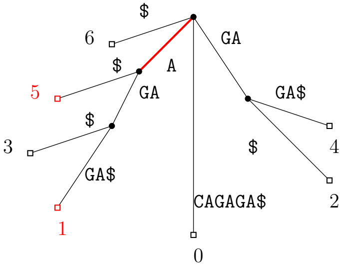


Application 6: Longest common prefix (LCP)

PREPROCESS: a sequence T
QUERY: a pair (i, j); return the length of the LCP of (T_i, T_j)

Example

Let $T = \texttt{CAGAGA\$}$. Let $(1, 5)$ be the query. The answer is $1 = |\texttt{A}|$.



Theorem

Longest common prefix queries can be answered in $O(1)$ time after $O(n)$ time preprocessing.

Application 7: Longest palindromic substring

INPUT: a sequence T
OUTPUT: a longest palindromic substring of T

Application 7: Longest palindromic substring

INPUT: a sequence T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \texttt{ATGTA} = S^R = \texttt{ATGTA}$.

Application 7: Longest palindromic substring

INPUT: a sequence T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \mathtt{ATGTA} = S^R = \mathtt{ATGTA}$.

- Construct the suffix tree of $W = T\mathtt{\#}T^R\mathtt{\$}$.

Application 7: Longest palindromic substring

INPUT: a sequence T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \texttt{ATGTA} = S^R = \texttt{ATGTA}$.

- Construct the suffix tree of $W = T\texttt{\#}T^R\texttt{\$}$.
- Preprocess the suffix tree for LCA queries.

Application 7: Longest palindromic substring

INPUT: a sequence T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \texttt{ATGTA} = S^R = \texttt{ATGTA}$.

- Construct the suffix tree of $W = T\texttt{\#}T^R\texttt{\$}$.
- Preprocess the suffix tree for LCA queries.
- Say we are interested in odd-length palindromes.

Application 7: Longest palindromic substring

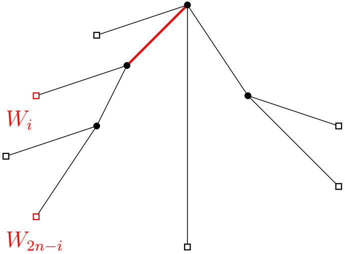
INPUT: a sequence T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \texttt{ATGTA} = S^R = \texttt{ATGTA}$.

- Construct the suffix tree of $W = T\#T^R\$$.
- Preprocess the suffix tree for LCA queries.
- Say we are interested in odd-length palindromes.
- Answer LCP queries for W_i and W_{2n-i}, for all i.

Application 7: Longest palindromic substring

INPUT: a sequence T
OUTPUT: a longest palindromic substring of T
Palindrome: $S = \texttt{ATGTA} = S^R = \texttt{ATGTA}$.

- Construct the suffix tree of $W = T\#T^R\$$.
- Preprocess the suffix tree for LCA queries.
- Say we are interested in odd-length palindromes.
- Answer LCP queries for W_i and W_{2n-i}, for all i.



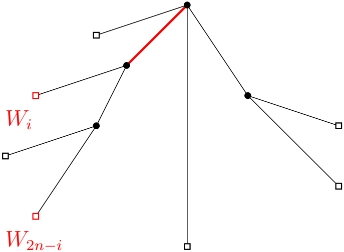
$T = \texttt{C}\underline{\texttt{ATGTA}}\texttt{TT}$

$T^R = \texttt{TTATGTAC}$

$W = T\#T^R\$ = \texttt{CAT}\overset{3}{\underline{\texttt{GTA}}}\underset{\times}{\texttt{T}}\texttt{T\#TTAT}\overset{13}{\underline{\texttt{GTA}}}\underset{\times}{\texttt{C}}\$$

Application 7: Longest palindromic substring

INPUT: a sequence T
OUTPUT: a longest palindromic substring of T



$T = \texttt{CA}\underline{\texttt{TGTA}}\texttt{TT}$

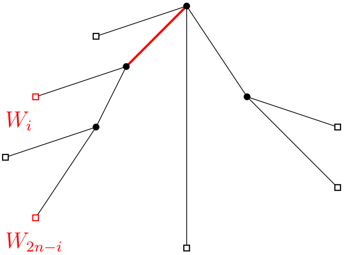
$T^R = \texttt{TTATGTAC}$

$W = T\#T^R\$ = \texttt{CAT}\underset{\times}{\overset{3}{\underline{\texttt{GTA}}}}\texttt{TT\#TTAT}\underset{\times}{\overset{13}{\underline{\texttt{GTA}}}}\texttt{C\$}$

Application 7: Longest palindromic substring

INPUT: a sequence *T*

OUTPUT: a longest palindromic substring of *T*



$T = \texttt{C}\underline{\texttt{ATGTA}}\texttt{TT}$

$T^R = \texttt{TTATGTAC}$

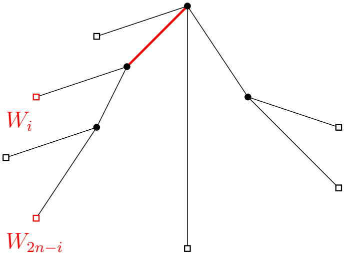
$W = T\#T^R\$ = \texttt{CAT}\overset{3}{\underline{\texttt{G}}}\underline{\texttt{TA}}\underset{\times}{\texttt{T}}\texttt{T\#TTAT}\overset{13}{\underline{\texttt{G}}}\underline{\texttt{TA}}\underset{\times}{\texttt{C}}\texttt{\$}$

- A longest LCP represents a longest odd-length palindrome.

Application 7: Longest palindromic substring

INPUT: a sequence T

OUTPUT: a longest palindromic substring of T



$T = \texttt{C}\underline{\texttt{AT}\textcolor{red}{\texttt{GTA}}}\texttt{TT}$

$T^R = \texttt{TTAT}\textcolor{red}{\texttt{GTA}}\texttt{C}$

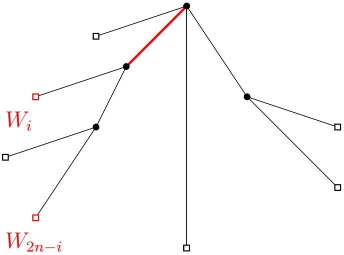
$W = T\#T^R\$ = \texttt{CAT}\overset{3}{\underline{\texttt{GTA}}}\underset{\times}{\texttt{T}}\texttt{T\#TTAT}\overset{13}{\underline{\texttt{GTA}}}\underset{\times}{\texttt{C}}\$$

- A longest LCP represents a longest odd-length palindrome.
- Even-length palindromes are handled analogously.

Application 7: Longest palindromic substring

INPUT: a sequence T

OUTPUT: a longest palindromic substring of T



$T = \texttt{C}\underline{\texttt{ATGTA}}\texttt{TT}$

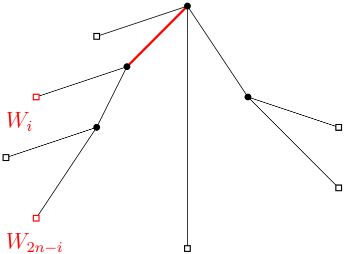
$T^R = \texttt{TTATGTAC}$

$W = T\#T^R\$ = \texttt{CAT}\underline{\overset{3}{\texttt{G}}\texttt{TA}}\underset{\times}{\texttt{T}}\texttt{T\#TTAT}\underline{\overset{13}{\texttt{G}}\texttt{TA}}\underset{\times}{\texttt{C}}\texttt{\$}$

- A longest LCP represents a longest odd-length palindrome.
- Even-length palindromes are handled analogously.
- Take the longer of the two as the globally longest.

Application 7: Longest palindromic substring

INPUT: a sequence T
OUTPUT: a longest palindromic substring of T



$T = \mathtt{C\underline{AT\textcolor{red}{GTA}}TT}$

$T^R = \mathtt{TTAT\textcolor{red}{GTA}C}$

$W = T\#T^R\$ = \mathtt{CAT\underline{\overset{3}{G}TA}\underset{\times}{T}T\#TTAT\underline{\overset{13}{G}TA}\underset{\times}{C}\$}$

- A longest LCP represents a longest odd-length palindrome.
- Even-length palindromes are handled analogously.
- Take the longer of the two as the globally longest.

Theorem

A longest palindromic substring can be computed in $O(n)$ time.

Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\mathtt{GCTA}, \mathtt{GCAA}) = 1$; $d_H(\mathtt{GCTA}, \mathtt{ACAA}) = 2$.

Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\texttt{GCTA}, \texttt{GCAA}) = 1$; $d_H(\texttt{GCTA}, \texttt{ACAA}) = 2$.

- Construct the suffix tree of $P\texttt{\#}T\texttt{\$}$.

Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\texttt{GCTA}, \texttt{GCAA}) = 1$; $d_H(\texttt{GCTA}, \texttt{ACAA}) = 2$.

- Construct the suffix tree of $P\texttt{\#}T\texttt{\$}$.
- Answer LCP query for T_i and $P\texttt{\#}T\texttt{\$}$, for $i = 0$.

Application 8: Approximate string matching

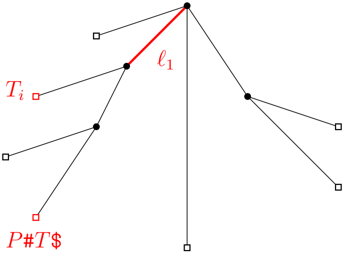
INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\texttt{GCTA}, \texttt{GCAA}) = 1$; $d_H(\texttt{GCTA}, \texttt{ACAA}) = 2$.

- Construct the suffix tree of $P\#T\$$.
- Answer LCP query for T_i and $P\#T\$$, for $i = 0$.
- Say this gives an LCP of length ℓ_1.

Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$
Hamming distance d_H: $d_H(\mathtt{GCTA}, \mathtt{GCAA}) = 1$; $d_H(\mathtt{GCTA}, \mathtt{ACAA}) = 2$.

- Construct the suffix tree of $P\#T\$$.
- Answer LCP query for T_i and $P\#T\$$, for $i = 0$.
- Say this gives an LCP of length ℓ_1.



Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

- “Jump” over the mismatch $T[i + \ell_1] \neq P[\ell_1]$.

Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

- “Jump” over the mismatch $T[i + \ell_1] \neq P[\ell_1]$.
- Via answering the LCP query for $T_{i+\ell_1+1}$ and P_{ℓ_1+1}.

Application 8: Approximate string matching

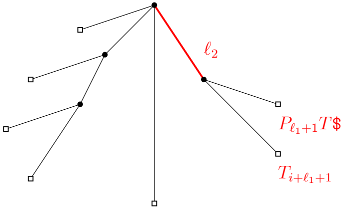
INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

- "Jump" over the mismatch $T[i + \ell_1] \neq P[\ell_1]$.
- Via answering the LCP query for $T_{i+\ell_1+1}$ and P_{ℓ_1+1}.
- This gives an LCP of length ℓ_2.

Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

- "Jump" over the mismatch $T[i + \ell_1] \neq P[\ell_1]$.
- Via answering the LCP query for $T_{i+\ell_1+1}$ and P_{ℓ_1+1}.
- This gives an LCP of length ℓ_2.

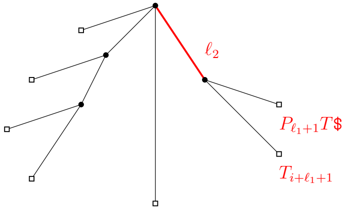


Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$

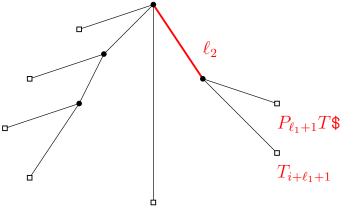
Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$



Application 8: Approximate string matching

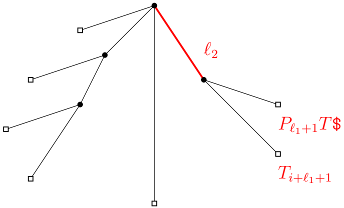
INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i+|P|-1], P) \leq k$



- Answer (at most) $k+1$ LCP queries per i.

Application 8: Approximate string matching

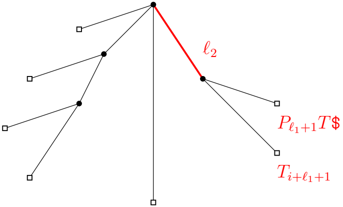
INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$



- Answer (at most) $k + 1$ LCP queries per i.
- Report i if the total length $\ell_1 + 1 + \ell_2 + 1 + \cdots$ is at least $|P|$.

Application 8: Approximate string matching

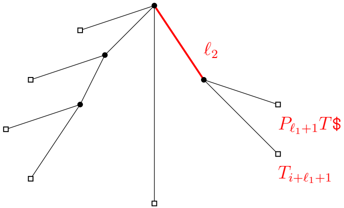
INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$



- Answer (at most) $k + 1$ LCP queries per i.
- Report i if the total length $\ell_1 + 1 + \ell_2 + 1 + \cdots$ is at least $|P|$.
- Repeat for all $i \in [1, n]$.

Application 8: Approximate string matching

INPUT: a sequence T, a pattern P, and an integer $k > 0$
OUTPUT: all positions i in T: $d_H(T[i + |P| - 1], P) \leq k$



- Answer (at most) $k + 1$ LCP queries per i.
- Report i if the total length $\ell_1 + 1 + \ell_2 + 1 + \cdots$ is at least $|P|$.
- Repeat for all $i \in [1, n]$.

Theorem (Landau and Vishkin, TCS 1986)

Approximate string matching can be solved in $O(kn)$ time.

Application 9: Shortest unique substring

INPUT: a sequence T
OUTPUT: a shortest unique substring of T

Application 9: Shortest unique substring

INPUT: a sequence T
OUTPUT: a shortest unique substring of T

- Construct the suffix tree of T.

Application 9: Shortest unique substring

INPUT: a sequence T
OUTPUT: a shortest unique substring of T

- Construct the suffix tree of T.
- For each leaf node labeled i, for all $i \in [0, n]$, pick up the closest ancestor v using a depth-first traversal.

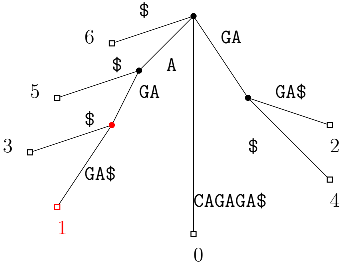
Application 9: Shortest unique substring

INPUT: a sequence T
OUTPUT: a shortest unique substring of T

- Construct the suffix tree of T.
- For each leaf node labeled i, for all $i \in [0, n]$, pick up the closest ancestor v using a depth-first traversal.

Example

Let $T =$ `CAGAGA$`.



Application 9: Shortest unique substring

INPUT: a sequence T
OUTPUT: a shortest unique substring of T

Application 9: Shortest unique substring

INPUT: a sequence T

OUTPUT: a shortest unique substring of T

- str(v) concatenated with the succeeding letter is the shortest unique substring starting at i.

Application 9: Shortest unique substring

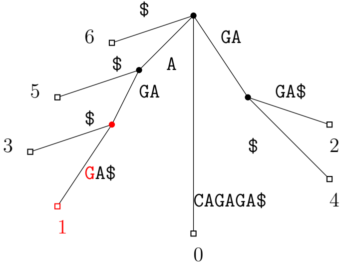
INPUT: a sequence T

OUTPUT: a shortest unique substring of T

- str(v) concatenated with the succeeding letter is the shortest unique substring starting at i.

Example

Let $T =$ `CAGAGA$`. The shortest unique substring starting at 1 is `AGAG`.



Application 9: Shortest unique substring

INPUT: a sequence T
OUTPUT: a shortest unique substring of T

Application 9: Shortest unique substring

INPUT: a sequence T
OUTPUT: a shortest unique substring of T

- Take a shortest substring among all i.

Application 9: Shortest unique substring

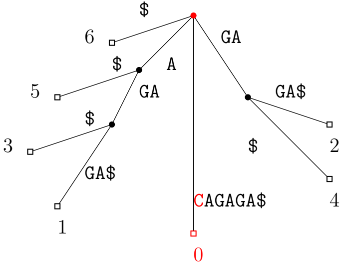
INPUT: a sequence T

OUTPUT: a shortest unique substring of T

- Take a shortest substring among all i.

Example

Let $T = \texttt{CAGAGA\$}$. The shortest unique substring is `C`.



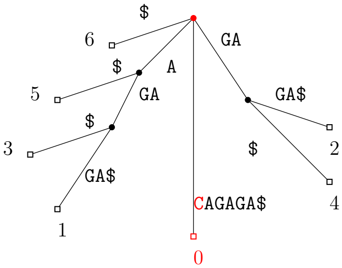
Application 9: Shortest unique substring

INPUT: a sequence T
OUTPUT: a shortest unique substring of T

- Take a shortest substring among all i.

Example

Let $T =$ `CAGAGA$`. The shortest unique substring is `C`.



Theorem

A shortest unique substring can be computed in $O(n)$ time.

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: LZ factorization of T

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: LZ factorization of T
LZ factorization of T:

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: LZ factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T

OUTPUT: LZ factorization of T

LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- F_i: longest prefix of $F_i \cdots F_k$ with some occurrence to the left.

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: LZ factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- F_i: longest prefix of $F_i \cdots F_k$ with some occurrence to the left.
- (or a single letter in case this prefix is empty.)

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: LZ factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- F_i: longest prefix of $F_i \cdots F_k$ with some occurrence to the left.
- (or a single letter in case this prefix is empty.)

Example

Let $T = \texttt{abbaabbbaaabab}$.

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: LZ factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- F_i: longest prefix of $F_i \cdots F_k$ with some occurrence to the left.
- (or a single letter in case this prefix is empty.)

Example

Let $T = \texttt{abbaabbbaaabab}$. The LZ factorization of T is

$$\texttt{a} \cdot \texttt{b} \cdot \texttt{b} \cdot \texttt{a} \cdot \texttt{abb} \cdot \texttt{baa} \cdot \texttt{ab} \cdot \texttt{ab}.$$

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: LZ factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- F_i: longest prefix of $F_i \cdots F_k$ with some occurrence to the left.
- (or a single letter in case this prefix is empty.)

Example

Let $T = \texttt{abbaabbbaaabab}$. The LZ factorization of T is

$$\texttt{a} \cdot \texttt{b} \cdot \texttt{b} \cdot \texttt{a} \cdot \texttt{abb} \cdot \texttt{baa} \cdot \texttt{ab} \cdot \texttt{ab}.$$

Why do we care?

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: LZ factorization of T
LZ factorization of T:

- $T = F_0 \cdot F_1 \cdots F_k$;
- F_i: longest prefix of $F_i \cdots F_k$ with some occurrence to the left.
- (or a single letter in case this prefix is empty.)

Example

Let $T = \texttt{abbaabbbaaabab}$. The LZ factorization of T is

$$\texttt{a} \cdot \texttt{b} \cdot \texttt{b} \cdot \texttt{a} \cdot \texttt{abb} \cdot \texttt{baa} \cdot \texttt{ab} \cdot \texttt{ab}.$$

Why do we care? LZ factorization is a basic and powerful technique for text compression (and string algorithms)!

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: Lempel-Ziv factorization of T

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T

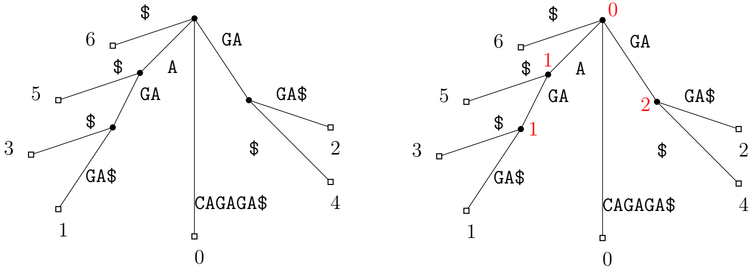
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.

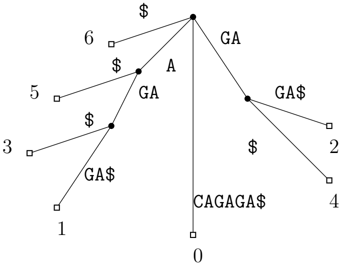


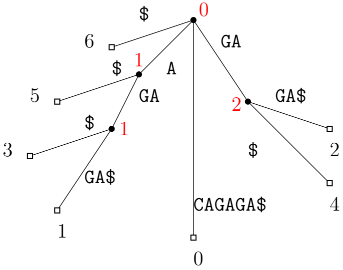
- How?

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.



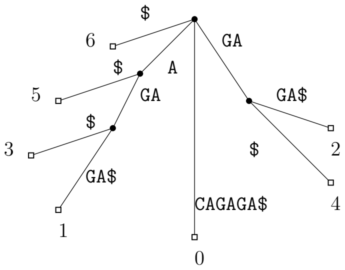


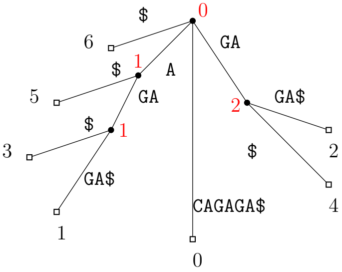
- How? Use a depth-first traversal, propagate the starting positions upwards, and keep the minimum value.

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.





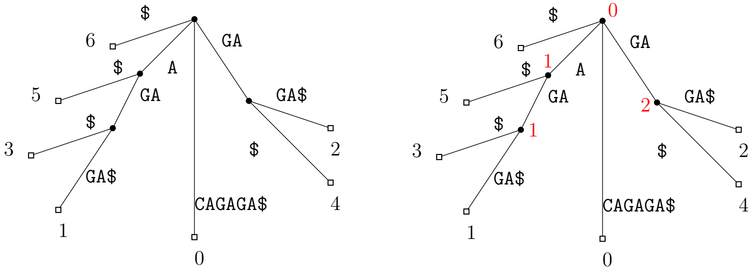
- Spell T, from left to right, in the suffix tree of T.

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T

OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.



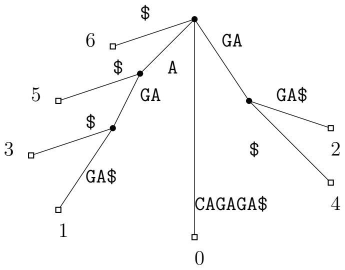
- Spell T, from left to right, in the suffix tree of T.
- For each match $T[i\,..\,j]$ check the leftmost starting position p.

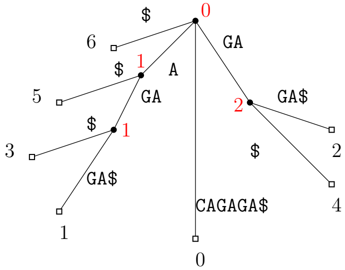
Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T

OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.





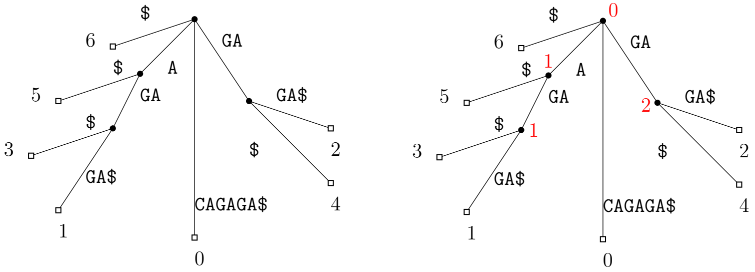
- If $p \geq i$, we have not seen $T[i\,..\,j]$ previously, increment i.

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T

OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.

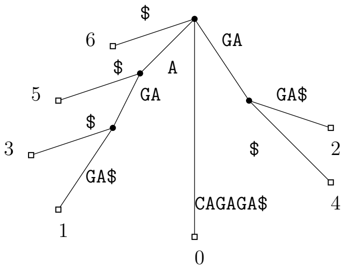


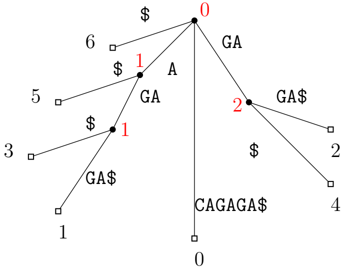
- If $p \geq i$, we have not seen $T[i \mathinner{..} j]$ previously, increment i.
- If $p < i$, we have seen $T[i \mathinner{..} j]$ previously, increment j.

Application 10: Lempel-Ziv (LZ) factorization

INPUT: a sequence T
OUTPUT: Lempel-Ziv factorization of T

- Construct the suffix tree of T.
- Decorate each internal node with the leftmost starting position the string it represents occurs.





Theorem

The Lempel-Ziv factorization can be computed in $O(n)$ time.