UA RUHR

Algorithmen auf Sequenzen

Volltext-Indexdatenstrukturen: Suffixbäume

Sven Rahmann

Genominformatik
Universitätsklinikum Essen
Universität Duisburg-Essen
Universitätsallianz Ruhr

Motivation

- Bei wiederholten Suchen in (langen) Texten ist es sinnvoll, den Text vorzuverarbeiten.
- Durch Indizierung des Textes wird die Suchzeit auf $\mathcal{O}(m) = \mathcal{O}(|P|)$ reduziert.
- Vorverarbeitung optimal $\mathcal{O}(n) = \mathcal{O}(|T|)$
- Gesamtzeit für k Muster: $\mathcal{O}(n + km)$ statt $\mathcal{O}(k(n + m))$

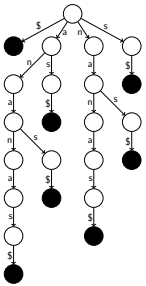
Motivation

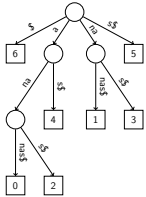
- Bei wiederholten Suchen in (langen) Texten ist es sinnvoll, den Text vorzuverarbeiten.
- Durch Indizierung des Textes wird die Suchzeit auf $\mathcal{O}(m) = \mathcal{O}(|P|)$ reduziert.
- Vorverarbeitung optimal $\mathcal{O}(n) = \mathcal{O}(|T|)$
- Gesamtzeit für k Muster: $\mathcal{O}(n + km)$ statt $\mathcal{O}(k(n + m))$
- Bei natürlichsprachlichen Texten können Wort-basierte Indizes eingesetzt werden.
- Indizes, die die Suche nach beliebigen Teilstrings (wortübergreifend) erlauben, heißen Volltext-Indizes.

Motivation

- Bei wiederholten Suchen in (langen) Texten ist es sinnvoll, den Text vorzuverarbeiten.
- Durch Indizierung des Textes wird die Suchzeit auf $\mathcal{O}(m) = \mathcal{O}(|P|)$ reduziert.
- Vorverarbeitung optimal $\mathcal{O}(n) = \mathcal{O}(|T|)$
- Gesamtzeit für k Muster: $\mathcal{O}(n + km)$ statt $\mathcal{O}(k(n + m))$
- Bei natürlichsprachlichen Texten können Wort-basierte Indizes eingesetzt werden.
- Indizes, die die Suche nach beliebigen Teilstrings (wortübergreifend) erlauben, heißen Volltext-Indizes.
- Annahme: Alphabet konstanter Größe: $|\Sigma| = \mathcal{O}(1)$.

Überblick





6	$
0	ananas$
2	anas$
4	as$
1	nanas$
3	nas$
5	s$

$ananas
ananas$
anas$an
as$anan
nanas$a
nas$ana
s$anana

Suffix Trie | Suffixbaum | Suffixarray | FM-Index

Grundidee der Volltext-Indizes

Indizierung sämtlicher Suffixe eines Textes T:
Jeder Teilstring von T ist Präfix eines Suffixes von T.

```
mississippi
 ississippi
  ssissippi
   sissippi
    issippi
     ssippi
      sippi
       ippi
        ppi
         pi
          i
```

Grundidee der Volltext-Indizes

Indizierung sämtlicher Suffixe eines Textes T:
Jeder Teilstring von T ist Präfix eines Suffixes von T.

```
mississippi
 ississippi
  ssissippi
   sissippi
    issippi
     ssippi
      sippi
       ippi
        ppi
         pi
          i
```

Quadratisch viele
$(n \cdot (n+1))/2$ Zeichen,
wenn alle Suffixe betrachtet werden.

Suffix Trie

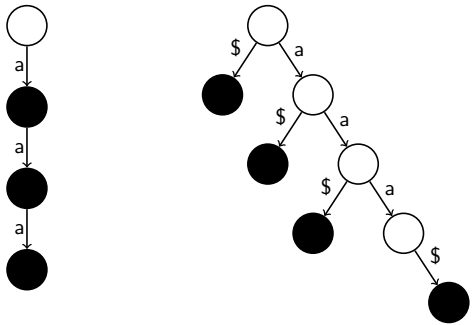
- Trie aus allen Suffixen eines Wortes (ähnlich Aho-Corasick)

Suffix Trie

- Trie aus allen Suffixen eines Wortes (ähnlich Aho-Corasick)
- Gewünscht: Bijektion zwischen Blättern des Baums und den Suffixen des Textes.
- Einführung eines Wächters (sentinel) \$ am Ende des Textes
  - $\Sigma \cap \$ = \emptyset$
  - $\$ < \sigma$ für alle $\sigma \in \Sigma$

Suffix Trie

Suffix Trie für den Text $T = \texttt{aaa}$ mit und ohne sentinel:



Alphabet-Abhängigkeit

Ist die Alphabetgröße nicht konstant, hängt die Laufzeit von der Datenstruktur ab, mit der Kinderknoten gesucht werden.

Sei $c_v \in \mathcal{O}(|\Sigma|)$ die Anzahl der Kinder von Knoten v.

Datenstruktur	*Laufzeit*	*Platz pro Knoten*	*Platz gesamt*
Verkettete Liste	$\mathcal{O}(c_v)$	$\mathcal{O}(c_v)$	$\mathcal{O}(n)$
Balancierter Baum	$\mathcal{O}(\log c_v)$	$\mathcal{O}(c_v)$	$\mathcal{O}(n)$
Array Größe $\|\Sigma\|$	$\mathcal{O}(1)$	$\mathcal{O}(\|\Sigma\|)$	$\mathcal{O}(n\|\Sigma\|)$
Perfektes Hashing[1]	$\mathcal{O}(1)$	$\mathcal{O}(c_v)$	$\mathcal{O}(n)$

[1] theoretisch möglich

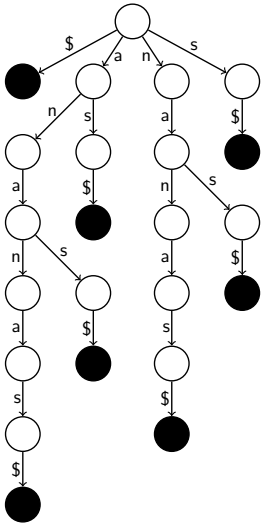
Exakte Mustersuche mit einem Suffix Trie

Mustersuche:

- Die Mustersuche beginnt an der Wurzel.
- Der Trie wird Zeichen für Zeichen des Musters traversiert, bis entweder das komplette Muster erkannt wurde (Treffer), oder von einem Knoten aus das passende Zeichen nicht existiert (kein Treffer).
- Im Erfolgsfall erhält man alle Startpositionen des Musters im Text, indem man die Blätter unter der gefundenen Position betrachtet.
- Im Misserfolgsfall kennt man die Positionen des längsten Präfixes des Musters, das im Text vorkommt.

Suffix Trie

Beispiel
$T = \texttt{ananas\$}$:



Implementierung

Ein Knoten des Tries wird als dictionary $\Sigma \to$ Kindknoten implementiert.

```
1  def build_trie(T):
2      root, n = dict(), len(T)
3      for t in [T[j:] for j in range(n)]:
4          node = root
5          for c in t:
6              if c not in node:
7                  node[c] = dict()
8              node = node[c]
9      return root
10
11 def has_pattern(node, P):
12     for c in P:
13         if c not in node: return False
14         node = node[c]
15     return True
```

Zusammenfassung: Suffix Trie

- Erstellung des Suffix Tries kostet $\mathcal{O}(n^2)$ Zeit
- Suffix Trie benötigt $\mathcal{O}(n^2)$ Speicher
- Mustersuche in $\mathcal{O}(m)$ Zeit
- Gezeigte Implementierung: Keine Positionen an den Blättern
- Tries werden wegen des hohen Speicherverbrauchs nicht genutzt.

Beobachtungen

- Suffix Trie hat Ketten von Knoten, die alle nur einen Nachfolger haben.
- Bei der Erstellung wird der i-te Buchstabe $i + 1$ mal betrachtet.
  Wünschenswert wäre, wenn dies nur einmal geschähe.

Beobachtungen

- Suffix Trie hat Ketten von Knoten, die alle nur einen Nachfolger haben.
- Bei der Erstellung wird der i-te Buchstabe $i + 1$ mal betrachtet.
  Wünschenswert wäre, wenn dies nur einmal geschähe.
- Diese Verbesserungen realisiert der Suffixbaum (engl.: suffix tree).

Σ^+-Baum

Definition (Σ^+-Baum)

Sei Σ ein Alphabet.
Ein Σ^+-Baum ein gewurzelter Baum, dessen Kanten mit einem nichtleeren String über Σ annotiert sind, so dass kein Knoten zwei ausgehende Kanten hat, die mit dem gleichen Buchstaben beginnen.

Σ^+-Baum

UA RUHR

Definition (Σ^+-Baum)

Sei Σ ein Alphabet.
Ein Σ^+-Baum ein gewurzelter Baum, dessen Kanten mit einem nichtleeren String über Σ annotiert sind, so dass kein Knoten zwei ausgehende Kanten hat, die mit dem gleichen Buchstaben beginnen.

Definition (Kompakter Σ^+-Baum)

Ein Σ^+-Baum heißt **kompakt**, wenn jeder Knoten (außer ggf. der Wurzel) mindestens zwei Kinder besitzt.

Definitionen

UA RUHR

Definition (Tiefe eines Knotens)

Die Tiefe $dep(s)$ eines Knotens s ist die Anzahl der Kanten des Pfades von der Wurzel zu s.
Die Wurzel r hat per Definition $dep(r) := 0$.

Definitionen

UA RUHR

Definition (Tiefe eines Knotens)

Die Tiefe $dep(s)$ eines Knotens s ist die Anzahl der Kanten des Pfades von der Wurzel zu s.
Die Wurzel r hat per Definition $dep(r) := 0$.

Definition (Stringtiefe eines Knotens)

Für Knoten s sei $str(s)$ die Konkatenation der Kantenbeschriftungen auf dem Pfad von der Wurzel zu s. Die Stringtiefe von s ist definiert als $strdep(s) := |str(s)|$.

Suffixbaum

Der Suffixbaum eines Textes $T\$$ ist ein Σ^+-Baum mit folgenden Eigenschaften:

- Es gibt eine Bijektion zwischen den Blättern und den Text-Suffixen.

Suffixbaum

Der Suffixbaum eines Textes $T\$$ ist ein Σ^+-Baum mit folgenden Eigenschaften:

- Es gibt eine Bijektion zwischen den Blättern und den Text-Suffixen.
- Die Kanten des Baums sind mit nicht-leeren Teilstrings von $T\$$ annotiert.

Suffixbaum

Der Suffixbaum eines Textes $T\$$ ist ein Σ^+-Baum mit folgenden Eigenschaften:

- Es gibt eine Bijektion zwischen den Blättern und den Text-Suffixen.
- Die Kanten des Baums sind mit nicht-leeren Teilstrings von $T\$$ annotiert.
- Ausgehende Kanten eines Knotens beginnen mit verschiedenen Buchstaben.

Suffixbaum

Der Suffixbaum eines Textes $T\$$ ist ein Σ^+-Baum mit folgenden Eigenschaften:

- Es gibt eine Bijektion zwischen den Blättern und den Text-Suffixen.
- Die Kanten des Baums sind mit nicht-leeren Teilstrings von $T\$$ annotiert.
- Ausgehende Kanten eines Knotens beginnen mit verschiedenen Buchstaben.
- Jeder innere Knoten hat ≥ 2 Kinder.

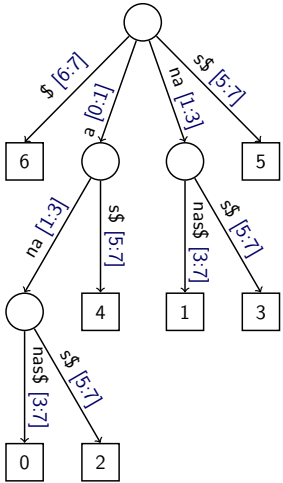
Suffixbaum

Der Suffixbaum eines Textes $T\$$ ist ein Σ^+-Baum mit folgenden Eigenschaften:

- Es gibt eine Bijektion zwischen den Blättern und den Text-Suffixen.
- Die Kanten des Baums sind mit nicht-leeren Teilstrings von $T\$$ annotiert.
- Ausgehende Kanten eines Knotens beginnen mit verschiedenen Buchstaben.
- Jeder innere Knoten hat ≥ 2 Kinder.
- Jeder Teilsting von $T\$$ kann auf einem Pfad von der Wurzel abgelesen werden.

Suffixbaum

Beispiel:
$T = \texttt{ananas\$}$:



Eigenschaften des Suffixbaums

- Speicherverbrauch pro Kante/Knoten ist $\mathcal{O}(1)$, da nur noch das Indexpaar (b, e) und die Nachkommen des Knotens gespeichert werden.
  Das Paar (b, e) entspricht dem Teilstring $T[b : e]$.

Eigenschaften des Suffixbaums

- Speicherverbrauch pro Kante/Knoten ist $\mathcal{O}(1)$, da nur noch das Indexpaar (b, e) und die Nachkommen des Knotens gespeichert werden.
  Das Paar (b, e) entspricht dem Teilstring $T[b : e]$.
- Da es genau n Blätter gibt und jeder innere Knoten mindestens zwei Kinder hat, ist die Anzahl der inneren Knoten durch n beschränkt.

Eigenschaften des Suffixbaums

- Speicherverbrauch pro Kante/Knoten ist $\mathcal{O}(1)$, da nur noch das Indexpaar (b, e) und die Nachkommen des Knotens gespeichert werden.
  Das Paar (b, e) entspricht dem Teilstring $T[b : e]$.
- Da es genau n Blätter gibt und jeder innere Knoten mindestens zwei Kinder hat, ist die Anzahl der inneren Knoten durch n beschränkt.
- Der Speicherverbrauch liegt somit bei $\mathcal{O}(n)$.

Eigenschaften des Suffixbaums

- Speicherverbrauch pro Kante/Knoten ist $\mathcal{O}(1)$, da nur noch das Indexpaar (b, e) und die Nachkommen des Knotens gespeichert werden.
  Das Paar (b, e) entspricht dem Teilstring $T[b : e]$.
- Da es genau n Blätter gibt und jeder innere Knoten mindestens zwei Kinder hat, ist die Anzahl der inneren Knoten durch n beschränkt.
- Der Speicherverbrauch liegt somit bei $\mathcal{O}(n)$.
- Bemerkenswert: Konstruktion in $\mathcal{O}(n)$ Zeit möglich
  (Algorithmus von Ukkonen)

Ukkonen-Algorithmus

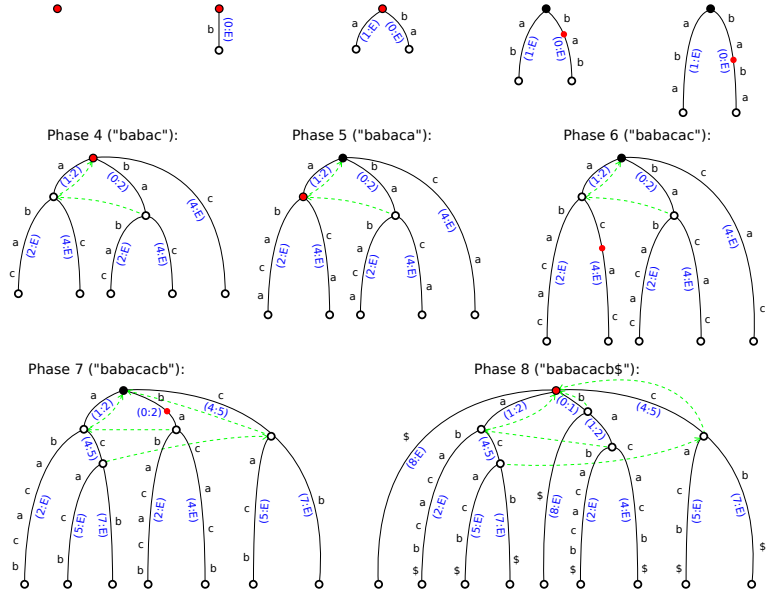
- Der Algorithmus konstruiert zu einem Text $T\$$ mit $n := |T\$|$ einen Suffixbaum in n Iterationen $0, 1, \ldots, n-1$.
- In Iteration i wird $T[i]$ zum Suffixbaum hinzugefügt.

Ukkonen-Algorithmus

- Der Algorithmus konstruiert zu einem Text $T\$$ mit $n := |T\$|$ einen Suffixbaum in n Iterationen $0, 1, \ldots, n-1$.
- In Iteration i wird $T[i]$ zum Suffixbaum hinzugefügt.
- Der Algorithmus ist ein *online*-Algorithmus.
  Nach Iteration i liegt "eine Art" Suffixbaum für das Präfix $T[..i]$ vor.

Ukkonen-Algorithmus

- Der Algorithmus konstruiert zu einem Text $T\$$ mit $n := |T\$|$ einen Suffixbaum in n Iterationen $0, 1, \ldots, n-1$.
- In Iteration i wird $T[i]$ zum Suffixbaum hinzugefügt.
- Der Algorithmus ist ein *online*-Algorithmus.
  Nach Iteration i liegt "eine Art" Suffixbaum für das Präfix $T[..i]$ vor.
- Um lineare Laufzeit zu erreichen,
  darf jede Iteration amortisiert nur $\mathcal{O}(1)$ Zeit dauern.
- Verschiedene "Tricks" sind dafür notwendig.


Initialisierung:
Phase 0 ("b"):
Phase 1 ("ba"):
Phase 2 ("bab"):
Phase 3 ("baba"):
Phase 4 ("babac"):
Phase 5 ("babaca"):
Phase 6 ("babacac"):
Phase 7 ("babacacb"):
Phase 8 ("babacacb$"):

Tricks des Ukkonen-Algorithmus

1. **Aktive Position** am Ende von Phase i: Längstes Suffix von $T[..i]$, das bereits Teilstring von $T[..(i-1)]$ war

Tricks des Ukkonen-Algorithmus

1. **Aktive Position** am Ende von Phase i: Längstes Suffix von $T[..i]$, das bereits Teilstring von $T[..(i-1)]$ war
2. Einfügen neuer Kanten und Blätter nur, wenn aktive Position diese benötigt: Diskrepanz zwischen Anzahl ℓ eingefügter Blätter und Phase i

Tricks des Ukkonen-Algorithmus

UA RUHR

1. **Aktive Position** am Ende von Phase i: Längstes Suffix von $T[..i]$, das bereits Teilstring von $T[..(i-1)]$ war
2. Einfügen neuer Kanten und Blätter nur, wenn aktive Position diese benötigt: Diskrepanz zwischen Anzahl ℓ eingefügter Blätter und Phase i
3. Nachträgliches Einfügen von Knoten und Blättern: Um vom Knoten s mit $str(s) = ax$ und $a \in \Sigma, x \in \Sigma^+$ zum Knoten w mit $str(w) = x$ in konstanter Zeit zu gelangen, werden Verbindungen (**Suffixlinks**) eingesetzt.

Tricks des Ukkonen-Algorithmus

1. **Aktive Position** am Ende von Phase i: Längstes Suffix von $T[..i]$, das bereits Teilstring von $T[..(i-1)]$ war
2. Einfügen neuer Kanten und Blätter nur, wenn aktive Position diese benötigt: Diskrepanz zwischen Anzahl ℓ eingefügter Blätter und Phase i
3. Nachträgliches Einfügen von Knoten und Blättern: Um vom Knoten s mit $str(s) = ax$ und $a \in \Sigma, x \in \Sigma^+$ zum Knoten w mit $str(w) = x$ in konstanter Zeit zu gelangen, werden Verbindungen (**Suffixlinks**) eingesetzt.
4. Beim Traversieren des Baums können beliebig lange Kanten in konstanter Zeit übersprungen werden

Tricks des Ukkonen-Algorithmus

1. **Aktive Position** am Ende von Phase i: Längstes Suffix von $T[..i]$, das bereits Teilstring von $T[..(i-1)]$ war
2. Einfügen neuer Kanten und Blätter nur, wenn aktive Position diese benötigt: Diskrepanz zwischen Anzahl ℓ eingefügter Blätter und Phase i
3. Nachträgliches Einfügen von Knoten und Blättern: Um vom Knoten s mit $str(s) = ax$ und $a \in \Sigma, x \in \Sigma^+$ zum Knoten w mit $str(w) = x$ in konstanter Zeit zu gelangen, werden Verbindungen (**Suffixlinks**) eingesetzt.
4. Beim Traversieren des Baums können beliebig lange Kanten in konstanter Zeit übersprungen werden
5. Aktualisierung aller existierenden Blattkanten in Schritt i würde $\mathcal{O}(i)$ Zeit, insgesamt also $\mathcal{O}(n^2)$ Zeit kosten. Um die Aktualisierung zu sparen, wird das Indexpaar an Blattkanten mit (b, ∞) beschriftet. Zum Abschluss werden die ∞ einmalig auf den korrekten Wert gesetzt.

Laufzeitanalyse (amortisiert)

Theorem (Laufzeit des Ukkonen-Algorithmus)

Ukkonen-Algorithmus konstruiert Suffixbaum zu $T\$$ mit $|T\$| = n$ in $\mathcal{O}(n)$ Zeit.

- Einfacher Fall:
  Aktive Position folgt existierendem Pfad,
  Stringtiefe erhöht sich um 1,
  "zahlt für" folgende Phasen, in denen Blätter hinzukommen

Laufzeitanalyse (amortisiert)

UA RUHR

Theorem (Laufzeit des Ukkonen-Algorithmus)

Ukkonen-Algorithmus konstruiert Suffixbaum zu $T\$$ mit $|T\$| = n$ in $\mathcal{O}(n)$ Zeit.

- Einfacher Fall:
  Aktive Position folgt existierendem Pfad,
  Stringtiefe erhöht sich um 1,
  "zahlt für" folgende Phasen, in denen Blätter hinzukommen
- Komplexer Fall: Neue Verzweigung(en) und Blatt/Blätter
  Folge Suffixlinks, um alle betroffenen Suffixe zu bearbeiten
  Stringtiefe verringert sich jeweils um 1

Laufzeitanalyse (amortisiert)

UA RUHR

Theorem (Laufzeit des Ukkonen-Algorithmus)

Ukkonen-Algorithmus konstruiert Suffixbaum zu $T\$$ mit $|T\$| = n$ in $\mathcal{O}(n)$ Zeit.

- Einfacher Fall:
  Aktive Position folgt existierendem Pfad,
  Stringtiefe erhöht sich um 1,
  "zahlt für" folgende Phasen, in denen Blätter hinzukommen
- Komplexer Fall: Neue Verzweigung(en) und Blatt/Blätter
  Folge Suffixlinks, um alle betroffenen Suffixe zu bearbeiten
  Stringtiefe verringert sich jeweils um 1
- Folgen/Erzeugen eines Suffixlinks in $\mathcal{O}(1)$ amortisierter Zeit:

Mustersuche im Suffixbaum

UA RUHR

Fragestellung: kommt P in T vor?

- Suche startet in der Wurzel
- Versucht, über Knoten und Kanten P zu folgen

Mustersuche im Suffixbaum

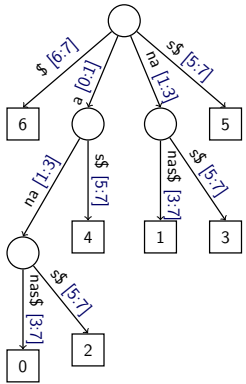
Fragestellung: kommt P in T vor?

- Suche startet in der Wurzel
- Versucht, über Knoten und Kanten P zu folgen
- $\mathcal{O}(1)$ Aufwand pro Buchstabe:
  Knoten: ausgehende Kante finden
  Kante: Zeichen vergleichen
- Laufzeit: $\mathcal{O}(m)$ für Entscheidung

Mustersuche im Suffixbaum

Fragestellung: kommt P in T vor?

- Suche startet in der Wurzel
- Versucht, über Knoten und Kanten P zu folgen
- $\mathcal{O}(1)$ Aufwand pro Buchstabe:
  Knoten: ausgehende Kante finden
  Kante: Zeichen vergleichen
- Laufzeit: $\mathcal{O}(m)$ für Entscheidung
- Anzahl und Positionen der Treffer:
  Unterbaum (Blätter) traversiseren,
  $\mathcal{O}(m + z)$ Zeit, z Ausgabegröße

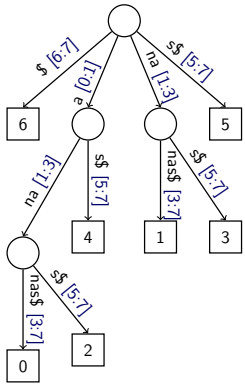


Längster wiederholter Teilstring

Sei $s^* := \arg\max_{s \in nodes(S)} \{strdep(s) \mid s \text{ ist innerer Knoten}\}$.
Dann ist der längste wiederholte Teilstring (LRS) $str(s^*)$.

Längster wiederholter Teilstring

Sei $s^* := \arg\max_{s \in nodes(S)} \{strdep(s) \mid s \text{ ist innerer Knoten}\}$.
Dann ist der längste wiederholte Teilstring (LRS) $str(s^*)$.
Beispiel: $T =$ `ananas$`; LRS ist `ana`.

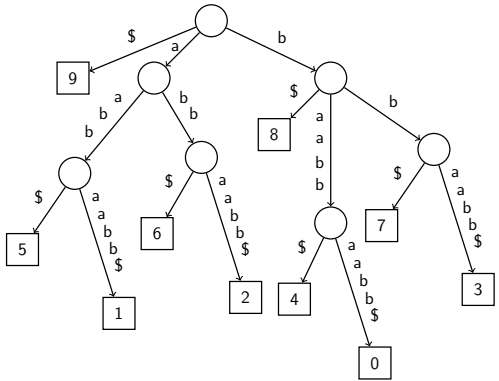


Kürzester eindeutiger Teilstring

Sei $s^* := \text{argmin}_{s \in nodes(S)}\{strdep(s) \mid s \text{ hat Blattkante } e, |e| > 1\}$.
Dann ist der kürzeste eindeutige Teilstring (SUS) $str(s^*) \circ e[0]$.

Kürzester eindeutiger Teilstring

Sei $s^* := \text{argmin}_{s \in nodes(S)}\{strdep(s) \mid s \text{ hat Blattkante } e, |e| > 1\}$.
Dann ist der kürzeste eindeutige Teilstring (SUS) $str(s^*) \circ e[0]$.
Beispiel: $T =$ `baabbaabb$`. SUS: `bba`



Längster gemeinsamer Teilstring

- Gegeben seien die Texte T_1 und T_2.
- Gesucht ist der längste gemeinsame Teilstring von T_1 und T_2.

Längster gemeinsamer Teilstring

- Gegeben seien die Texte T_1 und T_2.
- Gesucht ist der längste gemeinsame Teilstring von T_1 und T_2.
- Sei dementsprechend der verallgemeinerte Text $T = T_1\$_1 T_2\$_2$ mit $\$_1 < \$_2$.
- Suffixbaum zu T konstruieren

Längster gemeinsamer Teilstring

- Gegeben seien die Texte T_1 und T_2.
- Gesucht ist der längste gemeinsame Teilstring von T_1 und T_2.
- Sei dementsprechend der verallgemeinerte Text $T = T_1\$_1 T_2\$_2$ mit $\$_1 < \$_2$.
- Suffixbaum zu T konstruieren
- Alle Blätter mit Position $< |T_1\$_1|$ mit 1 beschriften.
- Alle restlichen Blätter mit 2 beschriften.

Längster gemeinsamer Teilstring

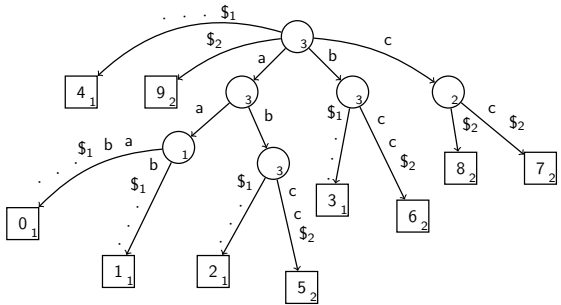
- Gegeben seien die Texte T_1 und T_2.
- Gesucht ist der längste gemeinsame Teilstring von T_1 und T_2.
- Sei dementsprechend der verallgemeinerte Text $T = T_1\$_1 T_2\$_2$ mit $\$_1 < \$_2$.
- Suffixbaum zu T konstruieren
- Alle Blätter mit Position $< |T_1\$_1|$ mit 1 beschriften.
- Alle restlichen Blätter mit 2 beschriften.
- Innere Knoten bottom-up mit dem Bitweisen Oder ihrer Kinder beschriften.

Längster gemeinsamer Teilstring

Sei $s^* := \arg\max_{s \in nodes(S)} \{strdep(s) \mid s \text{ innerer Knoten}, label(s) = 3\}$.
Dann ist der längste gemeinsame Teilstring (LCS) $str(s^*)$.

Längster gemeinsamer Teilstring

Sei $s^* := \arg\max_{s \in nodes(S)} \{strdep(s) \mid s \text{ innerer Knoten}, label(s) = 3\}$.
Dann ist der längste gemeinsame Teilstring (LCS) $str(s^*)$.
Beispiel: $T = \texttt{aaab\$}_1\texttt{abcc\$}_2$. LCS: `ab`



Zusammenfassung

- Volltext-Indizes erlauben schnellen ($\mathcal{O}(1)$) Zugriff auf alle Teilstrings eines Texts.
- Suffixbaum benötigt linearen Platz (Suffix Trie: quadratisch, ungeeignet)
- Ukkonen-Algorithmus erstellt Suffixbaum online in $\mathcal{O}(n)$ Zeit
- Anwendungen
  - Mustersuche: Kommt P in T vor? Wenn ja, wo?
  - längster wiederholter Teilstring im Text T
  - kürzester eindeutiger Teilstring im Text T
  - längster gemeinsamer Teilstring zweier Texte T_1, T_2
- Implementierung: hohe Konstante beim Speicherverbrauch,
  bei guten Implementierungen ca. 20 Bytes pro Zeichen.