ADS: Algorithmen und Datenstrukturen 1

Teil 13

Uwe Quasthoff

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

23. Januar 2018

[Letzte Aktualisierung: 22/01/2018, 16:35]

Hinweise zur PRÜFUNG

Zeit und Ort:

- Dienstag, 06.02.2018 von 13:00 - 14:00
- drei Hörsäle: Audimax, Hörsaal 3 und Felix-Klein-Hörsaal (Paulinum)
- Bitte 15 Min. vorher ankommen, um ggf. Hörsaal wechseln zu können

Letzte Vorlesung am 30.1.: Fragestunde

- Fragen werden diskutiert und gemeinsam beantwortet.
- Bitte Fragen vorher einreichen an: adshelp@informatik.uni-leipzig.de
- Bis Ende der Woche: 27.1.2018

Prüfungsvorleistung: 50% der Punkte aus den Übungsaufgaben
Sobald alle Übungsaufgaben korrigiert sind, gibt es auf http://adsprak.informatik.uni-leipzig.de/ eine Liste mit Matrikelnummern und Punktzahl. Zusätzlich gibt es eine Liste derjenigen, die die 50% nur knapp verfehlt haben. Auf Wunsch (mitgeteilt an adshelp@informatik.uni-leipzig.de) werden diese Studierenden zu einem Extraseminar um den 30.01. eingeladen und können sich nachträglich qualifizieren.
Bei Unklarheiten bitte E-Mail an adshelp@informatik.uni-leipzig.de.

Suchen in sublinearer Zeit?

Bisher:

- einen Text der Länge m in $O(m)$ in einem Text der Länge n finden
- alle k Treffer in $O(m + k)$ Zeit
- longest common substring in $\Theta(n_1 + n_2)$
- alle k maximalen Paare in $O(n + k)$

Ziele:

- Generell: "sublineare" Suche
- n "groß": 10^9 Buchstaben
- m "klein": $10^{2.5}$ Buchstaben
- "kleine" Suchanfragen oft wiederholt

Lösungsansätze

- Textsuche (KMP): zu langsam wenn Anzahl der Suchanfragen k groß ($O(k(m+n))$)
- Suffix Bäume
- Suffix Arrays

Den großen Text vorverarbeiten (als Suffixbaum) um später schnell suchen zu können

Jeder Substring ist Prefix eines Suffixes

Definition: Suffixbaum

- Ein Suffixbaum T für den Eingabestring $S = s_1 \ldots s_n\$, \quad n \geq 1$ ist ein gewurzelter Baum
- $s_1 \ldots s_n \neq \$$
- T hat $n + 1$ Blätter
- jeder innere Knoten hat mindestens zwei Kinder
- jede Kante hat als Label ein Infix von S
- alle von einem Knoten ausgehende Kanten beginnen mit verschiedenen Buchstaben im Label
- jeder Pfad von der Wurzel zu einem Blatt beschreibt ein Suffix von S
- alle diese Pfade zusammen beschreiben alle Suffixe von S

Alle Suffixe eines Strings

- Gegeben: `S = babab$`
- $ kommt sonst nicht im String S vor

wie lassen sich nun "schnell" alle Vorkommen von `ba` finden?

Alle Suffixe:

```
babab$
abab$
bab$
ab$
b$
$
```

Naive Konstruktion des Suffix-Baumes "wotd"

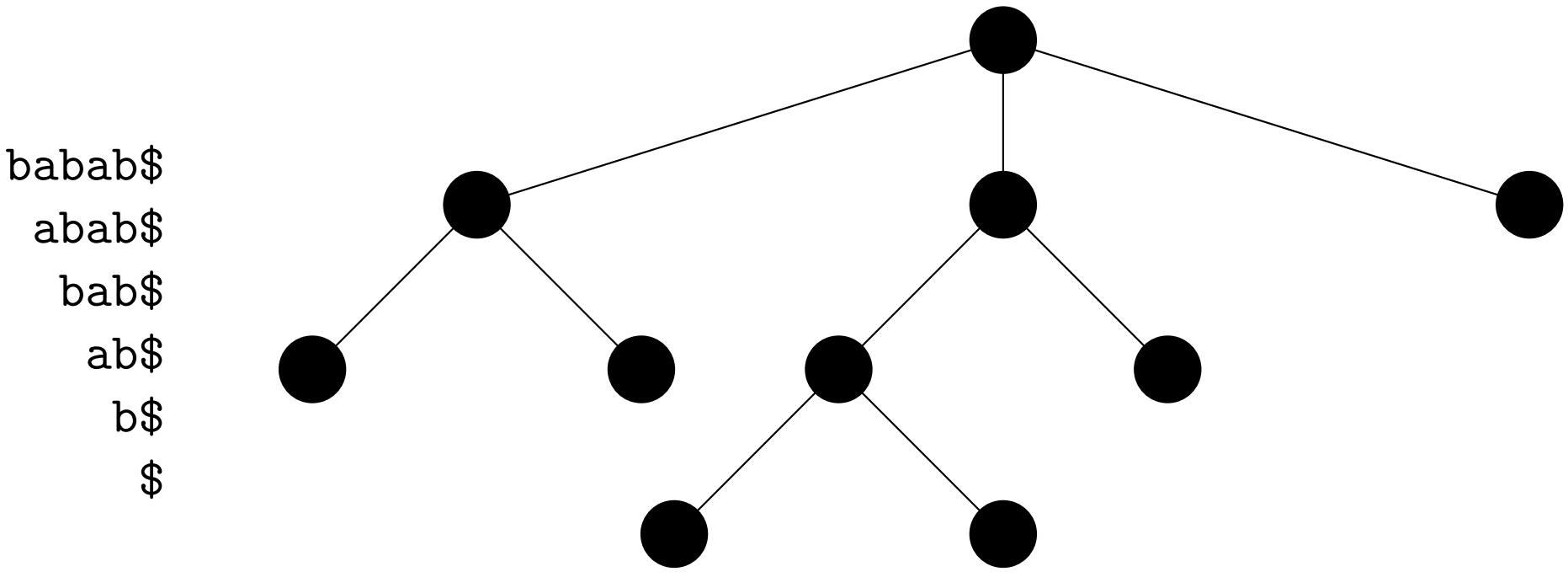
wodt: write-only-top-down

1. von der Wurzel bis zum Knoten $\bar{u}$ haben wir das Label u
2. Knoten $\bar{u}$: $R(\bar{u}) = \{s | us \quad \text{ist suffix von } S\}$
3. für alle Buchstaben $c \in \Sigma$ finde Untermenge an Suffixen die mit c beginnen: $G(\bar{u}, c) = \{w \in \Sigma^* | cw \in R(\bar{u})\}$
4.
   1. $|G(\bar{u}, c)| = 1$: Blattkante mit cw.
   2. sonst finde längstes gemeinsames Prefix $lcp(ucv)$, und setze Kantenlabel auf cv
5. wiederhole rekursiv (mit ucv als neuem u) bis nur noch Blätter erstellt werden

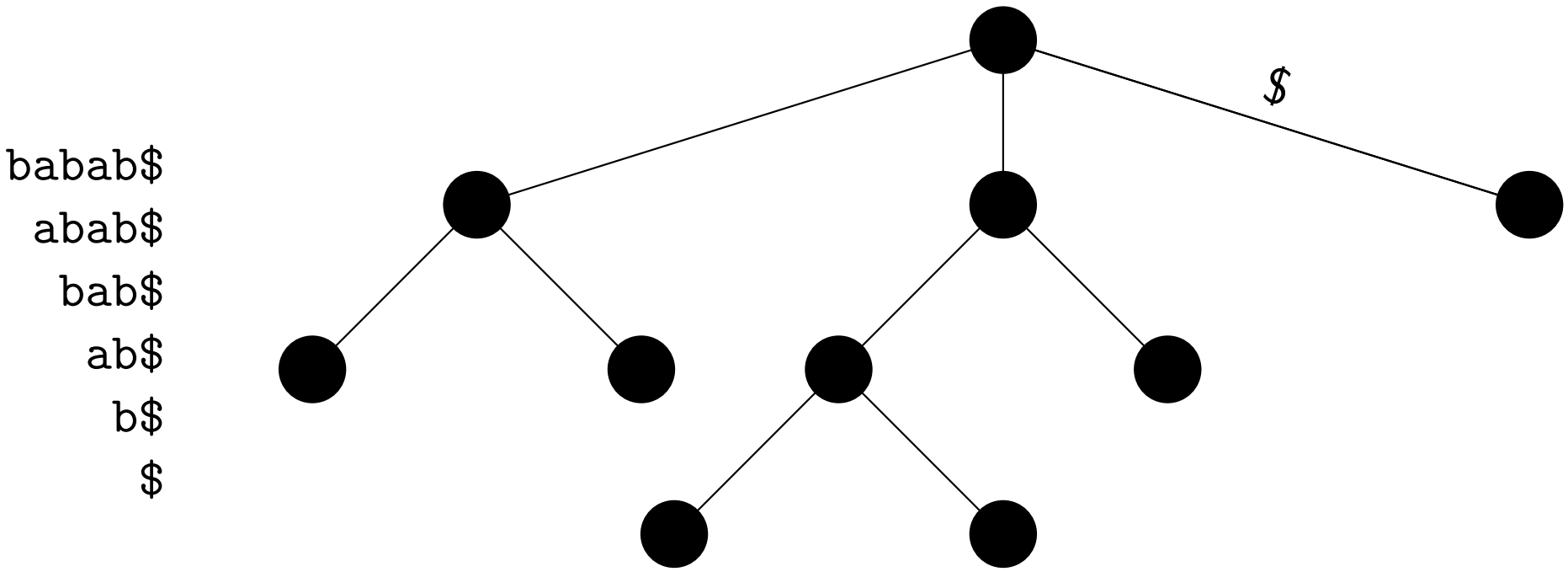
Σ ist das Alphabet, Σ^* alle Strings über Σ (auch der leere String)

(u.a.) Giegerich, Kurtz, Stoye, 2003, *efficient implementation of lazy suffix trees*

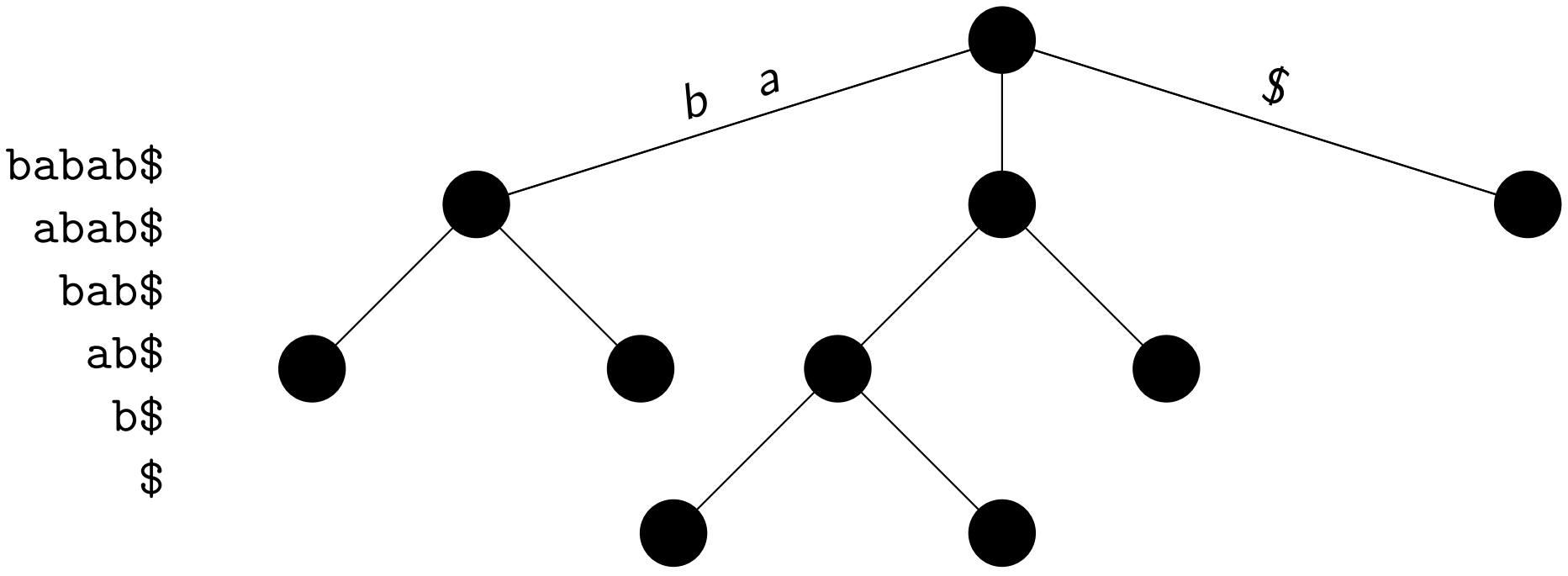
Konstruktion des Suffix-Baumes



Konstruktion des Suffix-Baumes

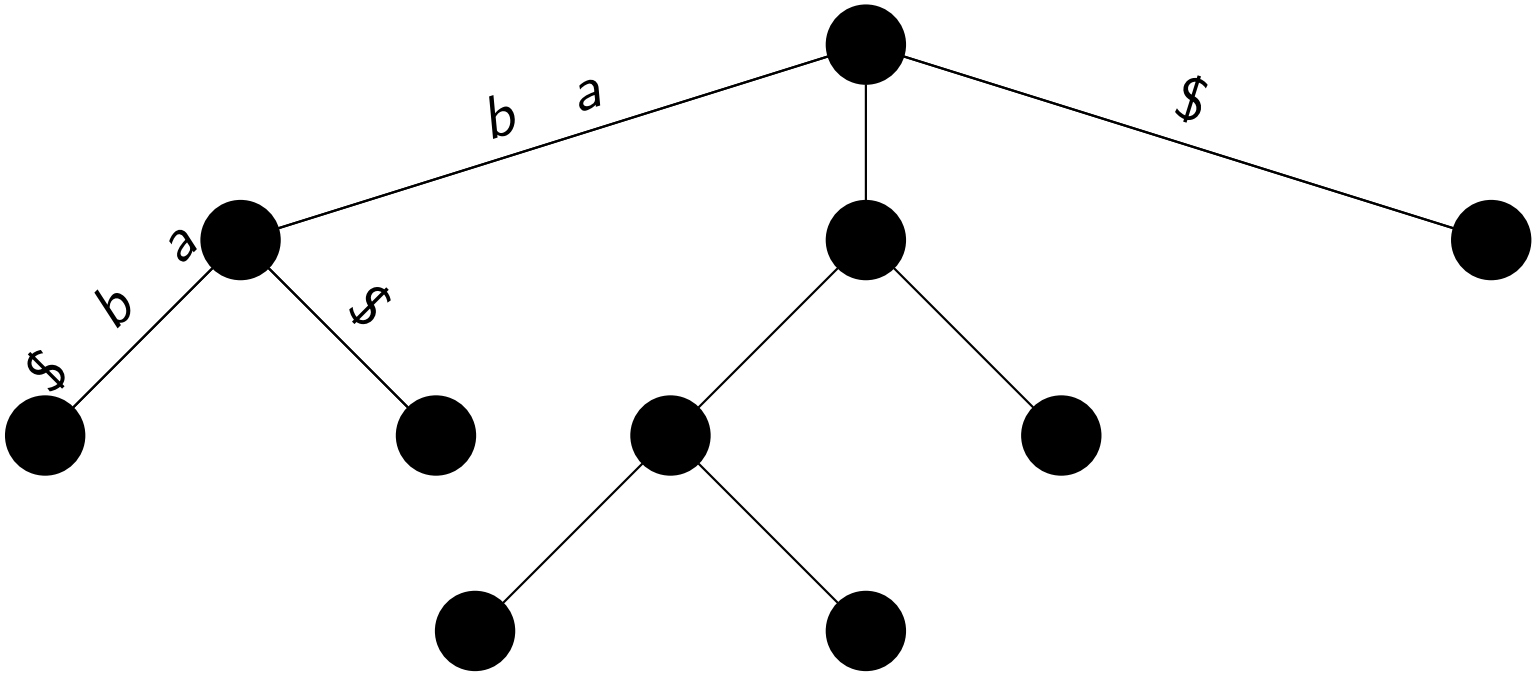


Konstruktion des Suffix-Baumes



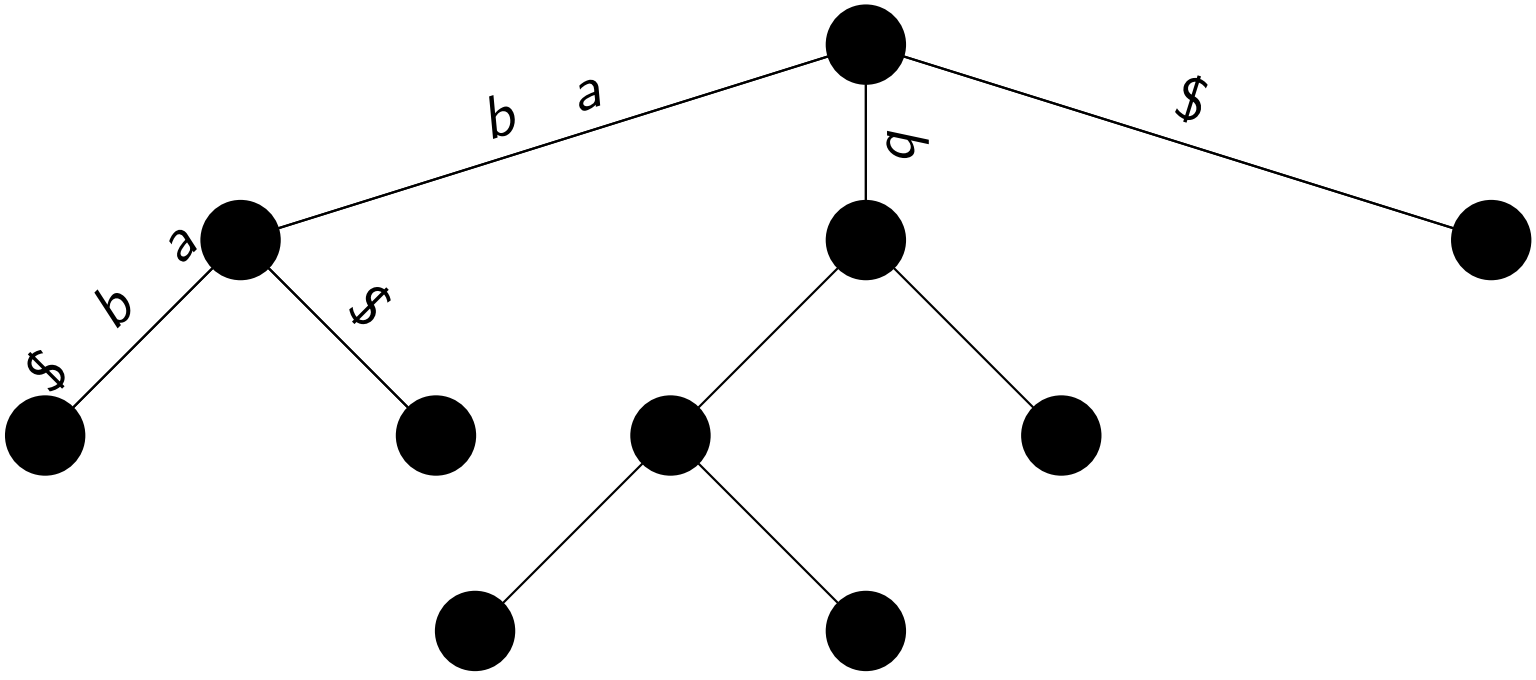
Konstruktion des Suffix-Baumes

babab$
abab$
bab$
ab$
b$
$



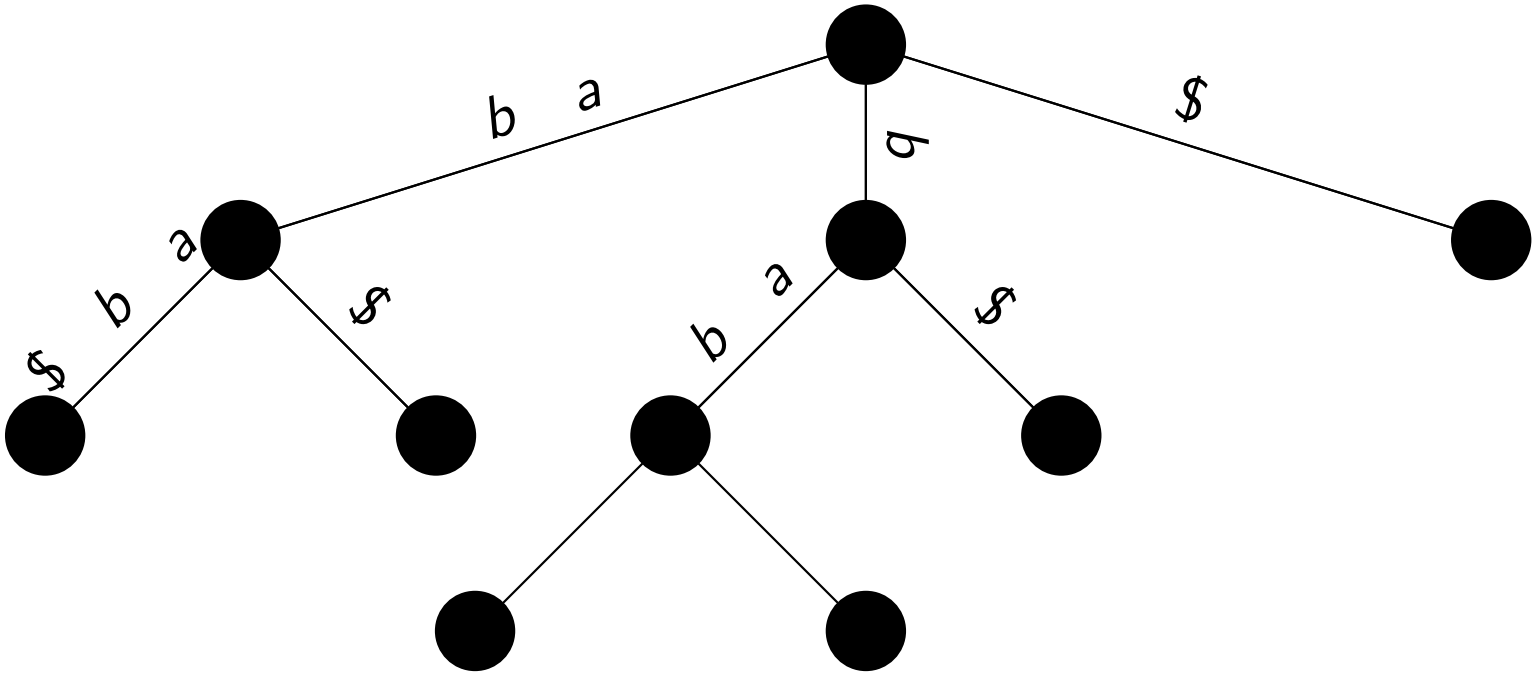
Konstruktion des Suffix-Baumes

babab$
abab$
bab$
ab$
b$
$

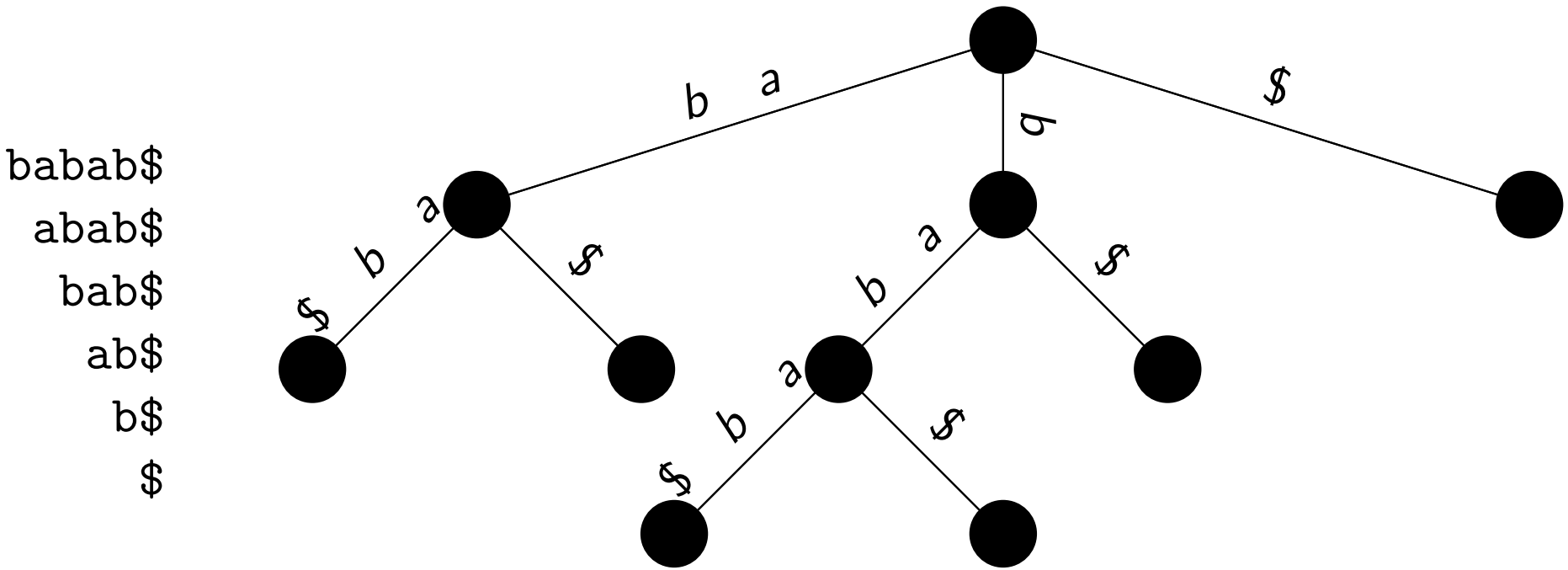


Konstruktion des Suffix-Baumes

babab$
abab$
bab$
ab$
b$
$



Konstruktion des Suffix-Baumes



Hinweise

- $ macht jedes Suffix einzigartig ("welches *ab* ist gemeint")
- kein Suffix soll Prefix eines anderen Suffixes sein
- jedes Suffix beschreibt einen vollständigen Pfad zu einem Blatt

Laufzeit

- $O(n)$ innere Knoten
- $O(n)$ Blätter
- pro Knoten: $O(n)$ für $G(u, c)$
- *lcp* amortisiert: $O(n^2)$
- insgesamt: $O(n^2)$

aber:

- erwartete Laufzeit: $O(n \log n)$
- `wotdlazy` (Giegerich, *et al*) baut nur die Teile des Baumes auf die benötigt werden: beschränken sich die Suchen auf einen kleinen Teilbaum, wird nur dieser gebaut

Optimierungen / Speicherverbrauch

- Kantenlabel nicht als String (“ab”) sondern als Paar (i,j) $S[i \ldots j]$ speichern
- 12 bytes pro Character
- Suffixbäume haben großen Overhead!

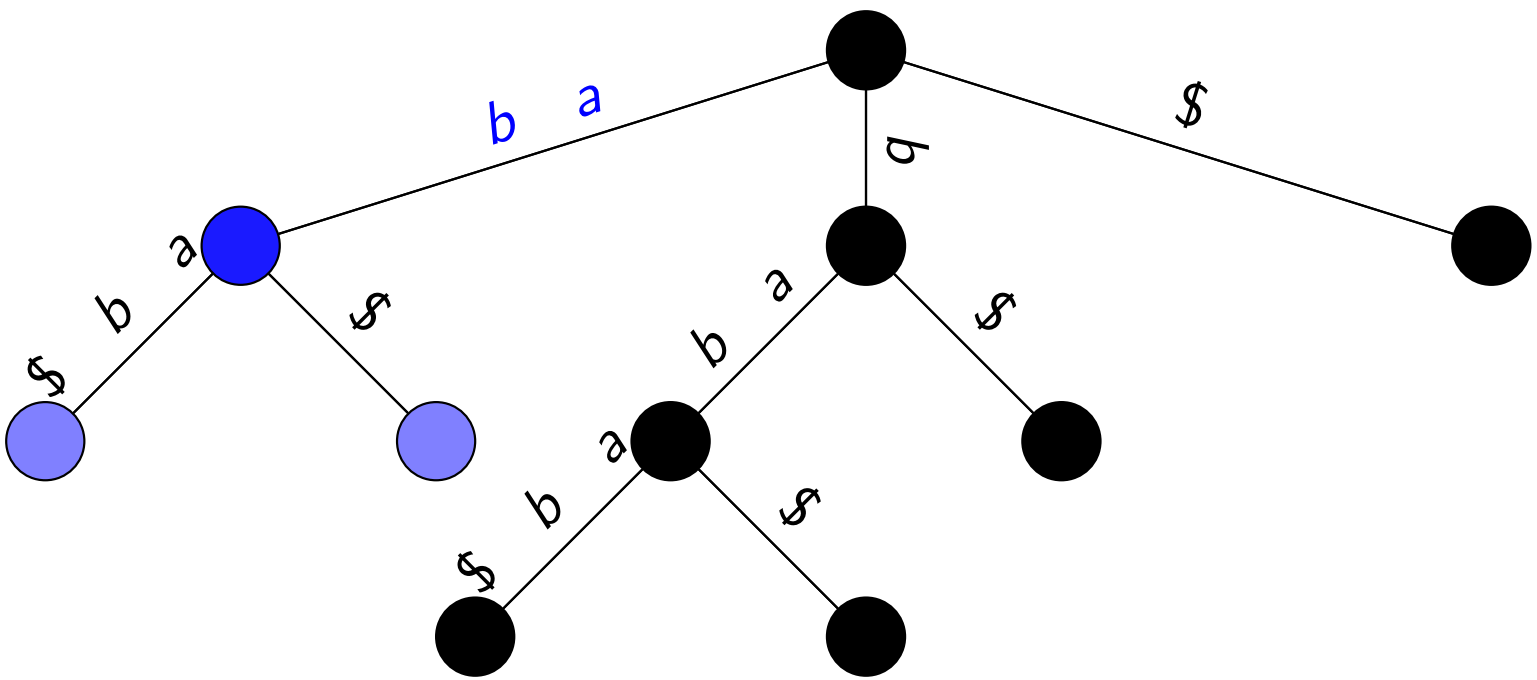
Algorithmus: Textsuche

Starte Suche bei der Wurzel

1. gegeben Suchmuster $Q = q_1 \dots q_m$
2. finde Kantenlabel das mit q_1 beginnt, dies sei $L = l_1 \dots l_k$; sonst: Muster nicht im Text
3.
   1. $m \leq k$ und $Q = L[1 \dots m]$: Muster gefunden
   2. $m \geq k$ und $Q[1 \dots k] = L$: rekursiv weitermachen mit $Q[k+1 \dots m]$ als neues Q
   3. "Mismatch": Muster nicht im Text
4. falls Muster gefunden: folge allen Pfaden zu Blättern um die Anzahl und Position aller (!) Matches zu bekommen

Anwendungen: Textsuche

finde (alle Vorkommen von) *ab* in *babab*\$:

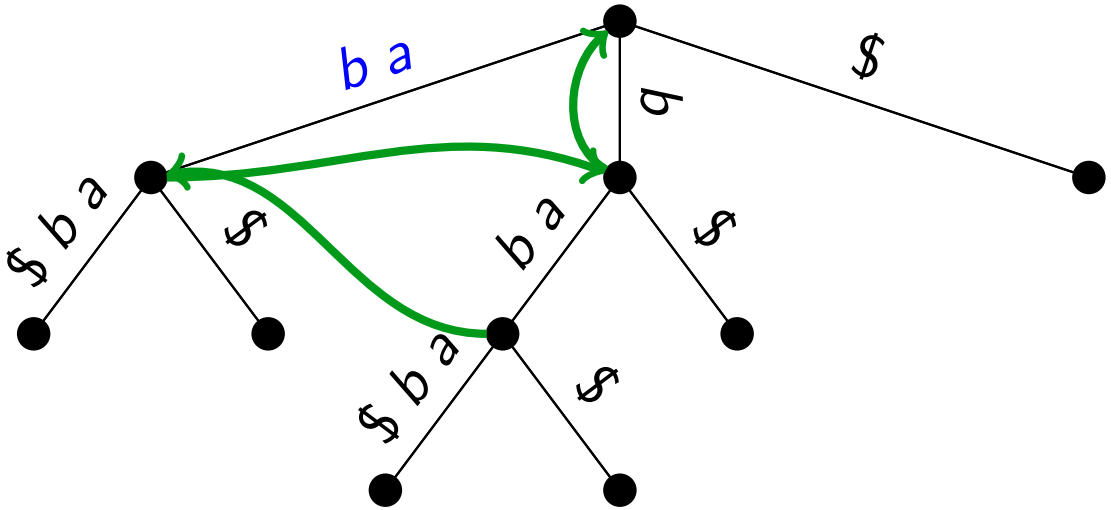


Längster Substring der k-mal auftaucht

1. pre-order: folge allen Pfaden von der Wurzel zu den Blättern, schreibe an jeden Knoten die Länge des Gesamtlabels bis dorthin
2. post-order: schreibe an jeden Knoten die Anzahl der Blätter im Teilbaum
3. finde Knoten mit k Blättern im Teilbaum (filtern), der das längste Gesamtlabel hat (maximieren)

Suffix Links

- verknüpft Knoten für uv mit Knoten für v
  bab $\rightarrow$ ba $\rightarrow$ b $\rightarrow$ "wurzel"
- suffix links zur Wurzel werden typischerweise nicht gezeichnet (wir tun dies hier doch!)



Finde längsten gemeinsamen Substring: $Q = bbba$ in $S = babab\$$

1. folge Q im Baum
2. wenn mismatch: speichere Tiefe im Baum
3. folge Suffix link ($bbba \rightarrow bba \rightarrow ba$) und zähle von vorne

Generalisierter Suffix-Baum

- Ein Baum T mit mehreren Strings
- $S = S_1 S_2 \ldots S_k = s_{11} \ldots s_{1n_1} \$_1 \ldots \$_{k-1} s_{k1} \ldots s_{kn_k} \$_k$
- Konstruktion wie gehabt

Wofür?

- finde alle S_i in denen Q auftaucht und wo
- finde längsten String der in min. l Strings existiert

Zusammenfassung

- Trie für alle Suffixe eines Strings S
- Einfaches Suchen, Zählen, und andere Abfragen
- hoher Speicherverbrauch, ca. 12 Byte pro Character !!!
- langsame $O(n^2)$, "einfache" oder schnelle $O(n)$, aber komplexere Konstruktion (Ukkonens Algorithmus)
- Erweiterbar für mehrere Strings
- 40 Jahre alt und immer noch aktiv beforscht!

Suffix Arrays

Definition (Suffix array (SA))

Ein SA vom String S ist ein Array aus n Integern, so dass gilt:
$SA[i] = k \Rightarrow$ Das Suffix k hat den lexikographischen Rank i über alle Suffixe von S.

- (endliches) Alphabet mit totaler Ordnung
- 4 Byte (8 Byte) pro Character, erlaubt 2^{32} (2^{64}) Zeichen

Beispiel Suffix Array

babab$

$ sei lex. kleiner als alle anderen Buchstaben

1	babab$
2	abab$
3	bab$
4	ab$
5	b$
6	$

Beispiel Suffix Array

`babab$`

$ sei lex. kleiner als alle anderen Buchstaben

1	`babab$`
2	`abab$`
3	`bab$`
4	`ab$`
5	`b$`
6	`$`

1	`babab$`
2	`abab$`
3	`bab$`
4	`ab$`
5	`b$`
6	`$`

Beispiel Suffix Array

babab$

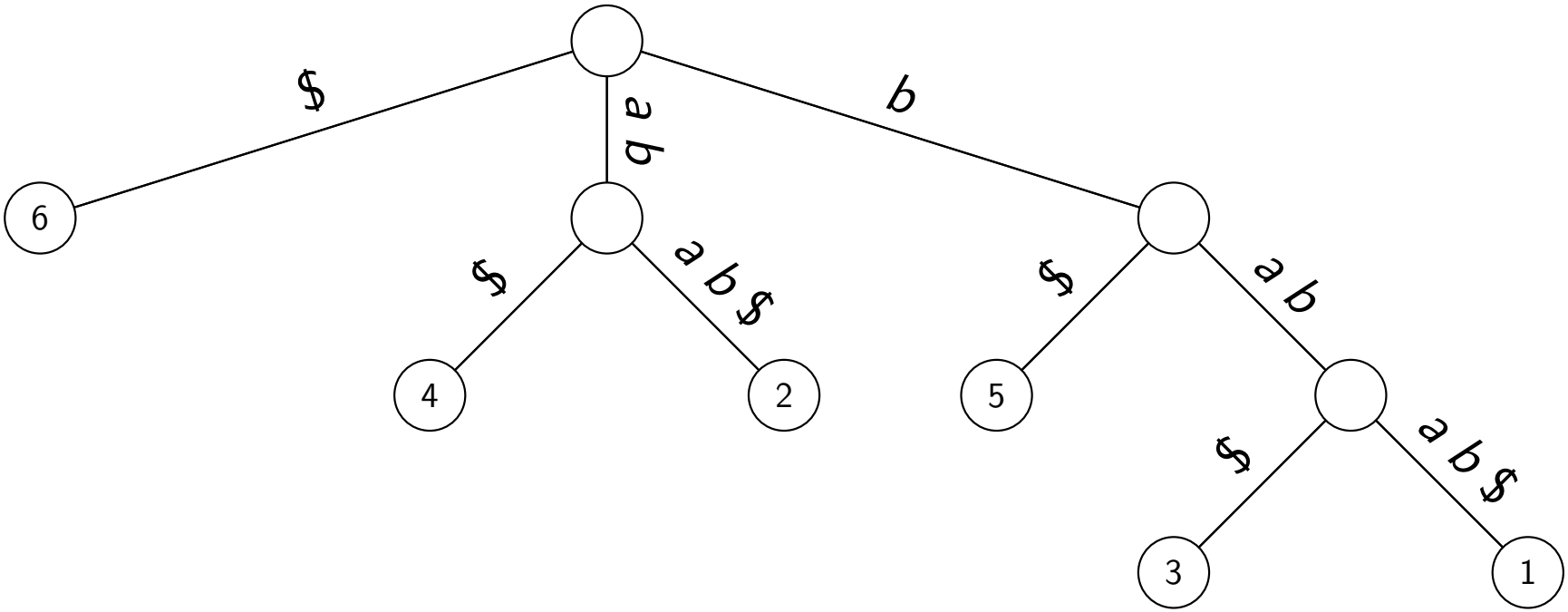
$ sei lex. kleiner als alle anderen Buchstaben

1	babab$
2	abab$
3	bab$
4	ab$
5	b$
6	$

1	babab$
2	abab$
3	bab$
4	ab$
5	b$
6	$

6	$
4	ab$
2	abab$
5	b$
3	bab$
1	babab$

Suffix-Baum vs Suffix Array



1	bababb$
2	ababb$
3	babb$
4	abb$
5	bb$
6	$

6	$
4	abb$
2	ababb$
5	bb$
3	babb$
1	bababb$

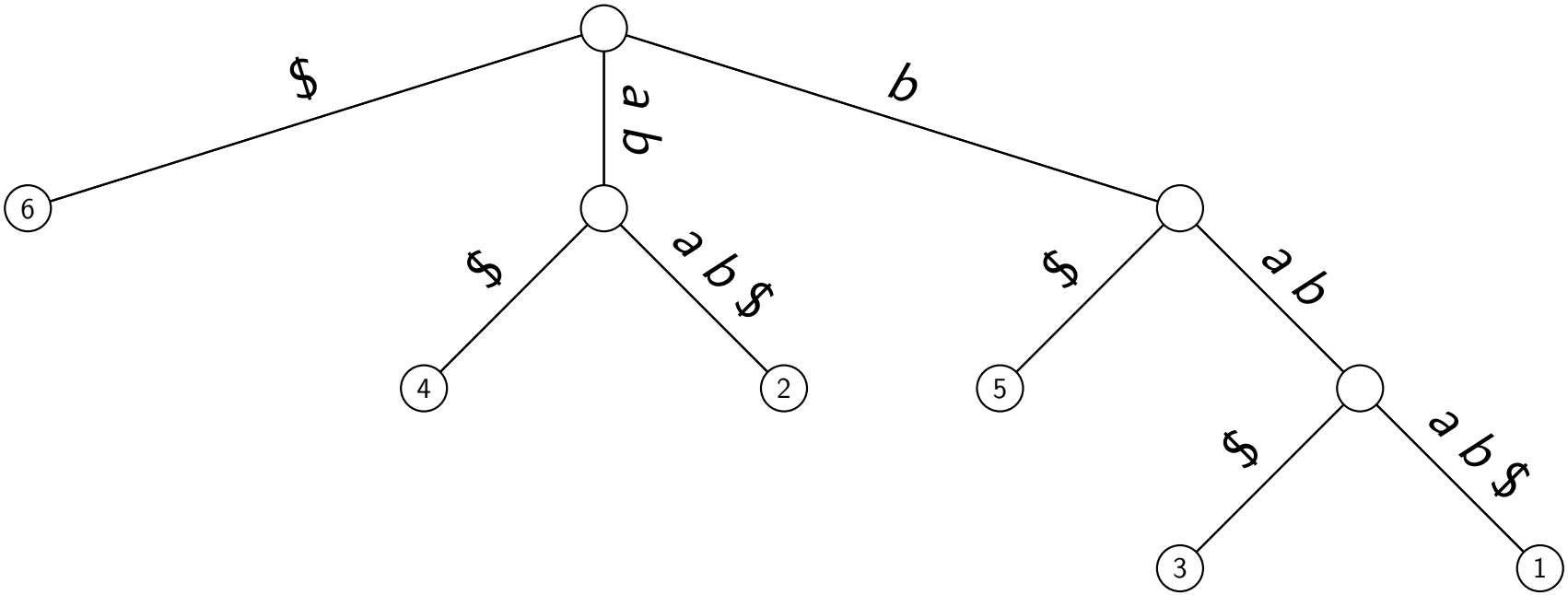
Konstruktion: Sortieren aller Suffixe

1. Gegeben $S = s_1 \dots s_n$
2. Erstelle Array SA mit $SA[i] = i$ der Größe n
3. Sortiere SA wobei $SA[i]$ und $SA[j]$ via $S[i \dots n]$ und $S[j \dots n]$ lexikografisch verglichen werden

die Startindices der Suffixe von S sind nun lexikografisch sortiert in SA.

- Sortieren von S in $O(n \log n)$
- Jeder lexikografische Vergleich i–j kostet $O(n)$
  (Suffixe können $O(n)$ gleiches Prefix haben, $S =$aaaaaa...)
- Insgesamt $O(n^2 \log n)$
- $4n$ Byte (4 Byte Int) Speicherverbrauch + eventueller Overhead durch Sortieralgorithmus

Konstruktion: direkt aus dem Suffix-Baum



1. Ordne Kanten jedes Knoten lexikografisch
2. Tiefensuche im Baum
3. Schreibe Blattlabels in Reihenfolge ihres Besuches heraus
   - Laufzeit: $O(n)$ falls Suffixbaum schon aufgebaut
   - Lexikografische Kantenordnung: bei Konstruktion
   - Speicher: $4n$ Byte

6	$
4	ab$
2	abab$
5	b$
3	bab$
1	babab$

Suffix-Array-Induced-Sorting

Idee: sortieren der Suffixe kann schneller als $O(n^2 \log n)$ sein, da (1) die Suffixe voneinander abhängen und (2) das Alphabet beschränk ist.
"Schrittweises Bucketsort"

(1) Klassifiziere den Suffix $S[i]$ als 'S' (smaller) wenn $S[i+1] > S[i]$ (lexikografisch), sonst als 'L' (larger)

(2) Markiere $S[i]$ als S^* wenn $S[i]$ vom Typ 'S' ist und $S[i-1] > S[i]$.

(3) Teile den Suffixarray in Buckets auf, wobei jeder Eimer mit die Suffixe enthält, die mit dem gleichen Zeichen beginnen. A besteht zunächst aus Intervallen für das 1. Zeichen jedes Suffix.

(4) Teile den Eimer für jedes Zeichen jeweils in einen 'S'- und einen 'L'-Eimer

(5) Sortiere die S^*-Suffixe lexikografisch in den zugehörigen 'S'-Eimer ein.

(6) Scanne A von links nach rechts:
Falls $A[i]$ vorhanden und $S[A[i]-1]$ vom Typ 'L' ist, schreibe $A[i]-1$ an die nächste freie Stelle im Typ 'L' Eimer für den Buchstaben $S_{A[i]-1}$.

(7) Scanne A von rechts nach links:
Falls $A[i]$ vorhanden und $S[A[i]-1]$ vom Typ 'S' ist, schreibe $A[i]-1$ an die nächste freie Stelle im Typ 'S' Eimer für den Buchstaben $S_{A[i]-1}$.

Suffix-Array-Induced-Sorting

Der Haken: das Sortieren der S* Suffixe!
Verwende SAIS rekursiv!
Beobachtung: 2 aufeinanerfolgende Zeichen können nicht mit S* markiert sein.
Laufzeit $T(n) = O(n) + T(n/2)$, daher $T(n) = O(n)$.

Ge Nong, Sen Zhang, Wai Hong Chan: Two Efficient Algorithms for Linear Time Suffix Array Construction. IEEE Trans. Computers 60(10): 1471-1484 (2011).

Suffix-Array-Induced-Sorting

$S =$ immissiissippi\$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_i	i	m	m	i	s	s	i	i	s	s	i	p	p	i	\$
Typ	S	L	L	S*	L	L	S*	S	L	L	S*	L	L	L	S*
Eimer	\$	i						m		p		s			
E-Typ	S	L	S					L		L		L			

Suffix-Array-Induced-Sorting

Schritt 5: Einsortieren der S* Suffixe

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_i	i	m	m	i	s	s	i	i	s	s	i	p	p	i	$
Typ	S	L	L	S*	L	L	S*	S	L	L	S*	L	L	L	S*
Eimer	$	i						m		p		s			
E-Typ	S	L	S					L		L		L			
A	15		7	11	4										

Suffix-Array-Induced-Sorting

Schritt 6: von links nach rechs:
Falls $A[i]$ vorhanden und $S[A[i]-1]$ vom Typ 'L' ist, schreibe $A[i]-1$ an die nächste freie Stelle im Typ 'L' Eimer für den Buchstaben $S_{A[i]-1}$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_i	i	m	m	i	s	s	i	i	s	s	i	p	p	i	$
Typ	S	L	L	S*	L	L	S*	S	L	L	S*	L	L	L	S*
Eimer	$	i						m		p		s			
E-Typ	S	L	S					L		L		L			
A	15	14	7	11	4			3	2	13	12	6	10	5	9

Suffix-Array-Induced-Sorting

Nach Schritt 6:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_i	i	m	m	i	s	s	i	i	s	s	i	p	p	i	$
Typ	S	L	L	S*	L	L	S*	S	L	L	S*	L	L	L	S*
Eimer	$	i						m		p		s			
E-Typ	S	L	S					L		L		L			
A	15	14	7	11	4			3	2	13	12	6	10	5	9

Suffix-Array-Induced-Sorting

Schritt 7: von rechts nach links:
Falls $A[i]$ vorhanden und $S[A[i]-1]$ vom Typ 'S' ist, schreibe $A[i]-1$ an die nächste freie Stelle im Typ 'S' Eimer für den Buchstaben $S_{A[i-1]}$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_i	i	m	m	i	s	s	i	i	s	s	i	p	p	i	$
Typ	S	L	L	S*	L	L	S*	S	L	L	S*	L	L	L	S*
Eimer	$	i						m		p		s			
E-Typ	S	L	S					L		L		L			
A	15	14	7	11	4			3	2	13	12	6	10	5	9
						4	8								
					11										
				1											

Suffix-Array-Induced-Sorting

Endergebnis

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S_i	i	m	m	i	s	s	i	i	s	s	i	p	p	i	$
Typ	S	L	L	S*	L	L	S*	S	L	L	S*	L	L	L	S*
Eimer	$	i						m		p		s			
E-Typ	S	L	S					L		L		L			
A	15	14	7	1	11	4	8	3	2	13	12	6	10	5	9

Stringsuche via Suffix Array

Finde alle Positionen k in S in denen $Q[1 \ldots q] = S[k \ldots k+q-1]$, also alle Infixe Q in S:

Finde $Q =$ba in $S=$babab\$:

babab\$ $\ldots S$

bab\$ $\ldots$ 2. Suffix beginnend mit ba $\ldots$

ba $\ldots Q$

- Binäre Suche um ersten Match zu finden $O(\log n * q)$
- Lineare Traversierung aller weiteren k Matches $O(kq)$
- $O(q)$ für jeden Stringvergleich
- Gesamt: $O(q(\log n + k))$

6	\$
4	ab\$
2	abab\$
5	b\$
3	bab\$
1	babab\$

Stringsuche via Suffix Array

Finde alle Positionen k in S in denen $Q[1 \ldots q] = S[k \ldots k+q-1]$, also alle Infixe Q in S:

Finde $Q =$`ba` in $S=$`babab$`:

`babab$` $\ldots S$

`bab$` $\ldots$ 2. Suffix beginnend mit `ba` $\ldots$

`ba` $\ldots Q$

- Binäre Suche um ersten Match zu finden $O(\log n * q)$
- Lineare Traversierung aller weiteren k Matches $O(kq)$
- $O(q)$ für jeden Stringvergleich
- Gesamt: $O(q(\log n + k))$

6	`$`
4	`ab$`
2	`abab$`
5	`b$`
3	`bab$`
1	`babab$`

Stringsuche via Suffix Array

Finde alle Positionen k in S in denen $Q[1 \ldots q] = S[k \ldots k+q-1]$, also alle Infixe Q in S:

Finde $Q =$`ba` in $S=$`babab$`:

`babab$` $\ldots S$

`bab$` $\ldots$ 2. Suffix beginnend mit `ba` $\ldots$

`ba` $\ldots Q$

- Binäre Suche um ersten Match zu finden $O(\log n * q)$
- Lineare Traversierung aller weiteren k Matches $O(kq)$
- $O(q)$ für jeden Stringvergleich
- Gesamt: $O(q(\log n + k))$

6	`$`
4	`ab$`
2	`abab$`
5	`b$`
3	`bab$`
1	`babab$`

Stringsuche via Suffix Array

Finde alle Positionen k in S in denen $Q[1 \ldots q] = S[k \ldots k+q-1]$, also alle Infixe Q in S:

Finde $Q =$`ba` in $S=$`babab$`:

`babab$` $\ldots S$

`bab$` $\ldots$ 2. Suffix beginnend mit `ba` $\ldots$

`ba` $\ldots Q$

- Binäre Suche um ersten Match zu finden $O(\log n * q)$
- Lineare Traversierung aller weiteren k Matches $O(kq)$
- $O(q)$ für jeden Stringvergleich
- Gesamt: $O(q(\log n + k))$

6	$
4	ab$
2	abab$
5	b$
3	bab$
1	babab$

Stringsuche via Suffix Array

Finde alle Positionen k in S in denen $Q[1 \ldots q] = S[k \ldots k+q-1]$, also alle Infixe Q in S:

Finde $Q =$`ba` in $S=$`babab$`:

`babab$` $\ldots S$

`bab$` $\ldots$ 2. Suffix beginnend mit `ba` $\ldots$

`ba` $\ldots Q$

- Binäre Suche um ersten Match zu finden $O(\log n * q)$
- Lineare Traversierung aller weiteren k Matches $O(kq)$
- $O(q)$ für jeden Stringvergleich
- Gesamt: $O(q(\log n + k))$

6	$
4	ab$
2	abab$
5	b$
3	bab$
1	babab$

Enhanced Suffix Arrays: *longest common prefix*

- Lineare Traversierung aller weiteren k Matches $O(kq)$
- $O(q)$ für jeden Stringvergleich

. . . besser . . .

- Speichere in $LCP[i]$ die Länge des längsten gemeinsamen Prefix von $SA[i]$ und $SA[i-1]$
- Speicher: n Byte + $O(z)$ Byte für alle LCP größer 2^7 in spezieller Datenstruktur (zB Hashtable)

Start	Suffix	LCP
6	$	-
4	ab$	0
2	abab$	2
5	b$	0
3	bab$	1
1	babab$	3

Enhanced Suffix Arrays: Suffix Links

- verknüpft Suffix uv in $SA[i]$ mit Suffix v in $SA[k]$
  $sfl[i] = k$
- erlaubt Finden von v, sobald u bekannt ist in: $O(1)$, statt $O(\log n)$

i	Start	Suffix	LCP	Suffix Link
1	6	$	-	-
2	4	ab$	0	4
3	2	abab$	2	5
4	5	b$	0	1
5	3	bab$	1	2
6	1	babab$	3	3

Substrings der Länge ℓ der k-mal auftaucht

Sei $\ell = 2$, finde k's:

Start	Suffix	LCP	Länge	Anzahl
6	$	-	1	0
4	ab$	0	2	1
2	abab$	2	2	2 (ab, 2×)
5	b$	0	2	1 (b$, 1×)
3	bab$	1	2	1
1	babab$	3	2	2 (ba, 2×)

1. Traversiere Suffix Array von oben nach unten: $k \leftarrow 1 \ldots n$
2. Wenn $SA[k]$ mit $length(SA[k]) < i$ dann: gebe Treffer für $SA[k-1]$ aus; setze $Anzahl[k] = 0$
3. Wenn $LCP[k] < \ell$ dann: gebe Treffer aus, $Anzahl[k] = 0$
4. Sonst $Anzahl[k] = Anzahl[k-1] + 1$

Zusammenfassung

- Die lexikografisch sortierte Liste aller Suffixe eines Strings ist ein *Suffix array*
- Speicherbedarf: String $S = s_1 \dots s_n$: zn Byte, mit z zB 4 oder 8 (32 / 64 Bit Rechner)
- extrem einfache Konstruktion in: $O(n^2 \log n)$
- einfache Konstruktion in: $O(n)$ Zeit (Kärkkäinen & Sanders, 2003, *Simple linear work Suffix Array construction*)
- sehr Effiziente Konstruktion: *Suffix-Array-Induced-Sorting* Ngong *et al* 2011 in $O(n)$
- *lcp*-Array speichert für zwei aufeinander folgende Strings die Länge des längsten gemeinsamen prefix
- Speicherbedarf lcp: $n + O(z)$ Byte, wobei $O(z)$ insgesamt für alle zu langen lcp anfallen
- (alle) Problem auf Suffixbäumen können mit gleicher Zeitkomplexität auch auf Suffix arrays gelöst werden . . . (unter zu Hilfenahme von Zusatzstrukturen wie *lcp*)

Hinweise zur PRÜFUNG

Zeit und Ort:

- Dienstag, 06.02.2018 von 13:00 - 14:00
- drei Hörsäle: Audimax, Hörsaal 3 und Felix-Klein-Hörsaal (Paulinum)
- Bitte 15 Min. vorher ankommen, um ggf. Hörsaal wechseln zu können

Letzte Vorlesung am 30.1.: Fragestunde

- Fragen werden diskutiert und gemeinsam beantwortet.
- Bitte Fragen vorher einreichen an: adshelp@informatik.uni-leipzig.de
- Bis Ende der Woche: 27.1.2018

Prüfungsvorleistung: 50% der Punkte aus den Übungsaufgaben
Sobald alle Übungsaufgaben korrigiert sind, gibt es auf http://adsprak.informatik.uni-leipzig.de/ eine Liste mit Matrikelnummern und Punktzahl. Zusätzlich gibt es eine Liste derjenigen, die die 50% nur knapp verfehlt haben. Auf Wunsch (mitgeteilt an adshelp@informatik.uni-leipzig.de) werden diese Studierenden zu einem Extraseminar um den 30.01. eingeladen und können sich nachträglich qualifizieren.
Bei Unklarheiten bitte E-Mail an adshelp@informatik.uni-leipzig.de.