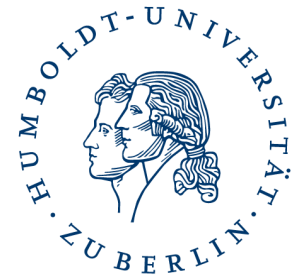


# Algorithmische Bioinformatik

## Suffixarrays

Ulf Leser

Wissensmanagement in der  
Bioinformatik



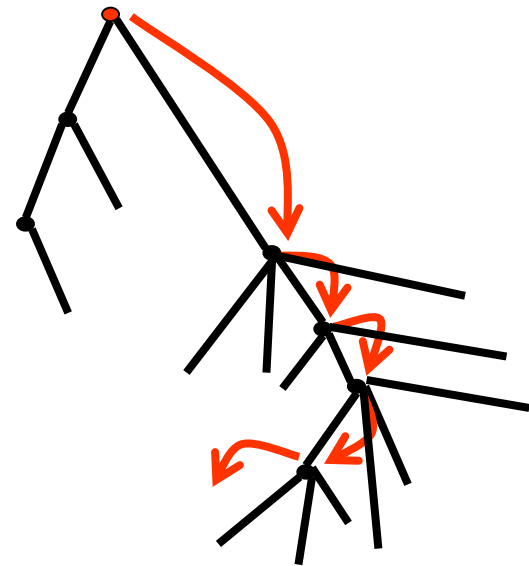
# Inhalt dieser Vorlesung

---

- Suffixarrays
  - Idee und Suche
  - Konstruktionsalgorithmus nach Manber / Myers
- Enhanced Suffix Arrays (Sketch)

# Suche in Suffixbäumen

- Matche Pattern bis Erfolg oder Mismatch
  - An jedem Knoten **Entscheidung** basierend auf erstem Zeichen des Kantenlabels
- Wie findet man die richtige Kante?
  - Array in der Größe des Alphabets  $\Sigma$ 
    - Zelle  $x$  mit Pointer auf Kante:  $O(1)$
  - Verkettete Liste
    - Pointer auf Kinder sind verkettet:  $O(|\Sigma|)$
  - Sortiertes (wachsendes) Array
    - Pointer auf Kinder alphabetisch sortiert:  $O(\log(|\Sigma|))$
    - Und mehr Arbeit beim Einfügen



# Verbesserungsbedarf

---

- Nachteile von Suffixbäumen
  - Sehr **hoher Platzverbrauch**
    - Viele Pointer, Arrays in Knoten etc.
  - Schlechte Lokalität / Ausnutzung Cache
  - Richtig schnell nur, wenn Kantenwahl in konstanter Zeit erfolgt
- DNA:  $|\Sigma|=4$ , Proteine: 20, Englisch: 23, Deutsch: 28
- Suffixbaum für **humanes Genom**:  $\sim 40\text{GB}$  Hauptspeicher

# Suffixarrays

- Definition

- Das *Suffixarray*  $A$  für String  $S$  ist ein Integerarray der Länge  $|S|$ , in dem  $A[i]$  die Startposition des  $i$ -ten Suffix von  $S$  enthält, sortiert nach *lexikographischer Ordnung*

- Beispiel

12345678901
mississippi

mississippi	i	$A[1]=11$
ississippi	ippi	$A[2]=8$
ssissippi	issippi	$A[3]=5$
sissippi	issippi	$A[4]=2$
issippi	mississippi	$A[5]=1$
ssippi	pi	$A[6]=10$
sippi	ppi	$A[7]=9$
ippi	sippi	$A[8]=7$
ppi	sissippi	$A[9]=4$
pi	ssippi	$A[10]=6$
i	ssissippi	$A[11]=3$

# Suche mit Suffixarrays

- Finde **alle Vorkommen** eines Pattern P im Suffixarray A für String S
- Idee?
- Suche alle Vorkommen von P=„ssi“ in „mississippi“

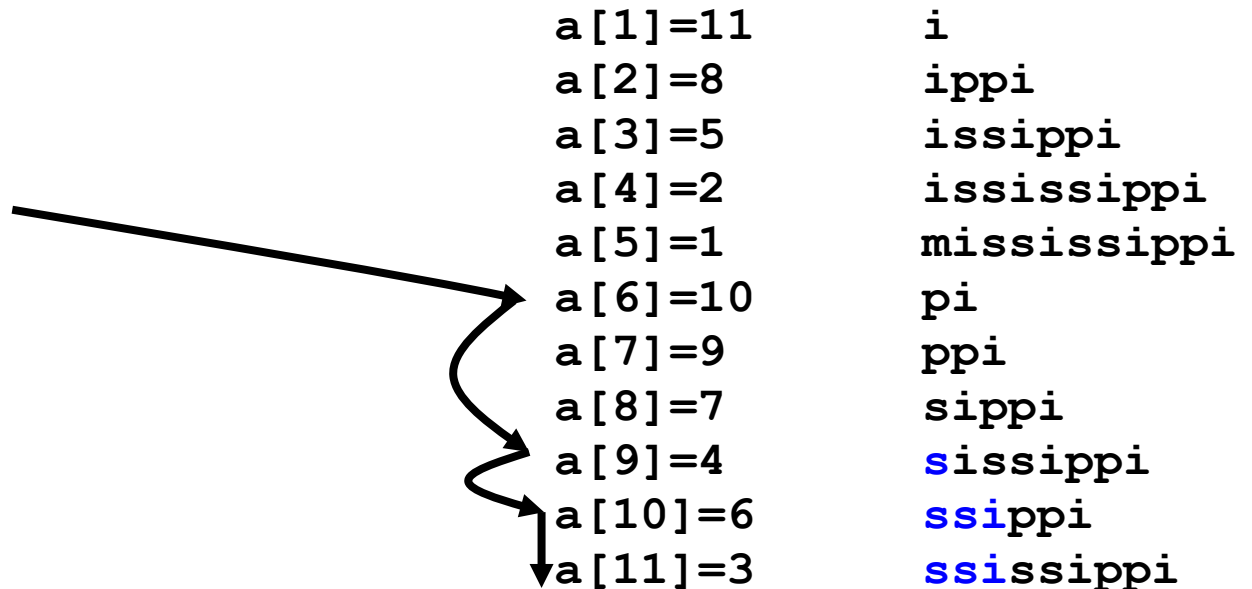
- Wenn P in S, dann muss P Präfix mindestens eines Suffix sein
- Suffixe sind in A sortiert
- **Binäre Suche** in A

a [ 1 ] = 11	i
a [ 2 ] = 8	ippi
a [ 3 ] = 5	issippi
a [ 4 ] = 2	ississippi
a [ 5 ] = 1	mississippi
a [ 6 ] = 10	pi
a [ 7 ] = 9	ppi
a [ 8 ] = 7	sippi
a [ 9 ] = 4	sissippi
a [ 10 ] = 6	ssippi
a [ 11 ] = 3	ssissippi

# Beispiel

---

- Suche alle Vorkommen von  $P = \text{„ssi“}$



# Suchalgorithmus

```
l:=1; r:=m; n=|P|; suffixarray A von S;
defunc f(i):=S[A[i]..A[i]+n-1];
while l<r do
  z := floor((r-1)/2);           // Middle of interval
  if (f(z) > P) then r := z-1;   // Go up
  else if f(z) < P then l := z+1; // Go down
  else                            // Found one occurrence
    z`:=z;
    while (f(z`) = P & z` > 0) do // Search smaller occ's
      report a[z`];
      z` := z`-1;
    end while;
    z`:=z+1;
    while (f(z`) = P & z` <= m) do // Search larger occ's
      report a[z`];
      z` := z`+1;
    end while;
    break;
  end if;
end while;
```



# Komplexität

---

- Wir machen  $O(\log(m))$  Sprünge im Array
- Jeder Test kostet im Worst-Case  $n$  Zeichenvergleiche
- Zusammen:  $O(n \cdot \log(m))$  (für ein Vorkommen)
  - Pessimistisch – jeder Vergleich müsste  $n$  Zeichenvergleiche brauchen
- Finden **aller Vorkommen**
  - $O(n \cdot \log(m) + k \cdot n)$
  - $k$ : Anzahl Vorkommen von  $P$  in  $T$

# Beschleunigung

---

- Man muss nicht immer 1..n Zeichen vergleichen
- Fortlaufend zwei Informationen merken
  - $p_l$ : Länge des längsten gemeinsamen Präfix von P und A[l]
  - $p_r$ : Länge des längsten gemeinsamen Präfix von P und A[r]
- Beim Vergleich mit dem mittleren Element des Intervalls
  - Sei  $p = \min(p_r, p_l)$
  - String [1..p] ist **gemeinsames Präfix aller Suffixe im Intervall**
  - Vergleiche Zeichen erst ab Position p+1
- Gewinn
  - Keine veränderte Worst-Case Komplexität
  - Aber sehr gute Ergebnisse (quasi-linear) **im Average Case**
- Suche geht auch in  $O(n + \log(m))$  (Gusfield, p. 152-154)

# Inhalt dieser Vorlesung

---

- Suffixarrays
  - Idee und Suche
  - Konstruktionsalgorithmus nach Manber / Myers
- Enhanced Suffix Arrays (Sketch)

# Konstruktion von Suffixarrays

---

- Erste Möglichkeit: Alle Suffixe aufzählen und **sortieren**
  - Braucht  $O(m \cdot \log(m))$  String-Vergleiche, die jeweils bis zu  $m$  Zeichen vergleichen – schlecht
- Zweite Möglichkeit: **Aus dem Suffixbaum**
  - Konstruktion des Suffixbaums
  - Lexikographisches Depth-First: Kinder in **lexikographischer Reihenfolge** der Kantenlabel ablaufen
    - „\$“ gilt als kleiner als alle anderen Zeichen
  - Blatt  $i$  wird als  $k$ -tes Blatt erreicht  $\Rightarrow A[k] = i$
  - Komplexität:  $O(m)$
  - Aber: Wir haben unser ursprüngliches **Platzproblem** nicht gelöst

# Konstruktion von Suffixarrays

---

- Wir besprechen den **originalen  $O(m \cdot \log(m))$**  Algorithmus
  - Manber, U. and Myers, G. (1993). "Suffix arrays: a new method for on-line string searches." *SIAM Journal of Computing* **22**(5)
- Erste **direkte, lineare Konstruktionsalgorithmen** seit 2003
  - Z.B. Kärkkäinen and Sanders: "Simple Linear Work Suffix Array Construction". 30th Int. Col. on Automata, Languages and Programming (ICALP'03)
  - Die haben aber oft hohe konstante Faktoren und sind in der Praxis nicht unbedingt schneller
  - Heute gibt es einen ganzen Zoo an Verfahren (später)

# Grundkenntnisse

---

- Aufgabe: Sortiere Menge von Strings **nach erstem Zeichen**
  - **Bucketsort** macht das in  $O(m)$ 
    - Durch alle Strings iterieren
    - Funktion mapped erstes Zeichen auf Bucket
    - Die Buckets sind nach dem ersten Zeichen sortiert
    - Vorsicht vor Platzbedarf
      - Naiv:  $|\Sigma|$  leere Buckets der Größe  $m$  erzeugen
      - Braucht  $O(m * |\Sigma|)$  Platz; geht aber besser (siehe Alg&DS)
- **Stabiles sortieren**
  - Sei  $S$  die Eingabemenge,  $S'$  die sortierte Menge
  - Ein **Sortierverfahren ist stabil**, wenn  $\forall a, b$  mit  $a=b$  und  $\text{pos}(a) < \text{pos}(b)$  in  $S$ , auch  $\text{pos}(a) < \text{pos}(b)$  in  $S'$
  - Merke: Bucketsort ist stabil

# Grundaufbau 1

---

- Erzeugen Array A mit **Startpositionen** aller Suffixe der Länge nach sortiert (1,2,3,4...)
- Sortiere A (stabil) mit Bucketsort nach erstem Zeichen
- Wir müssen noch **innerhalb jedes Buckets** sortieren
- Sortieren nach den zweiten Zeichen in einem Bucket B
  - Jedes zweite Zeichen ist natürlich das erste Zeichen des um eins längeren Suffix
  - **Nach denen haben wir schon sortiert** (im ersten Lauf)
  - Ausnutzen – erste Sortierung nachschlagen statt neu vergleichen
    - Lohnt sich jetzt noch nicht - Geduld

# Grundaufbau 2

---

- Damit haben wir neue (kleinere) Buckets, die nach den ersten zwei Zeichen sortiert sind
- Im nächsten Schritt sortieren wir nach dem **3. und 4. Zeichen** gleichzeitig unter Benutzung der vorhandenen Sortierung
- Dann nach dem 5-8, 9-16 ...
- Ende, wenn alle Buckets nur noch ein Suffix enthalten
- Komplexität
  - Es kann maximal  $\log(m)$  Durchläufe geben
  - In jedem Durchlauf läuft man im Worst-Case durch ganz A
  - Damit:  $O(m \cdot \log(m))$ 
    - [MM93] zeigen  $O(m \cdot \log(\log(m)))$  im Average Case



# Details

---

- Array mit allen Suffixstartpositionen von  $S$  der Länge nach absteigend initialisieren:  $A[i] = i$
- Nach dem ersten Zeichen stabil sortieren
- Sortierung nach dem **nächsten Zeichen**
  - Seien  $S[i..] = s_i \dots s_m\$$  und  $S[j..] = s_j \dots s_m\$$  in einem Bucket ( $s_i = s_j$ )
  - Man müsste nun  $s_{i+1}$  mit  $s_{j+1}$  vergleichen
  - Das muss **dasselbe Ergebnis haben** wie der Vergleich von  $S[i+1..] = s_{i+1} \dots s_m\$$  mit  $S[j+1..] = s_{j+1} \dots s_m\$$
  - Den Vergleich haben wir schon gemacht, müssten aber das Ergebnis erst noch in  $A$  „finden“
  - Besser: Wir laufen **einmal durch ganz  $A$** , womit wir an jedem zweiten Zeichen in richtiger Reihenfolge vorbeikommen, und erledigen dabei alle Sortierungen nach dem zweiten Zeichen

# Verfahren

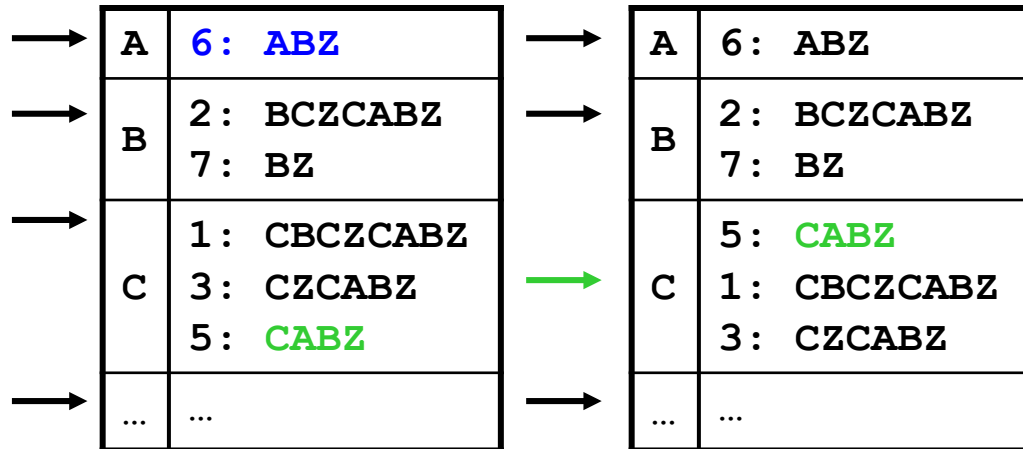
123456789012
MISSISSIPPI\$

\$	12: \$
I	2: ISSISSIPPI\$ 5: ISSIPPI\$ 8: IPPIS\$ 11: IS\$
M	1: MISSISSIPPI\$
P	9: PPI\$ 10: PI\$
S	3: SSISSIPPI\$ 4: SSISSIPPI\$ 6: SSISSIPPI\$ 7: SIPPIS\$

- Setze in jedem Bucket B einen Pointer  $P^B$  auf das **erste Element**
- Wir gehen einmal durch A (Variable i)
  - Sei  $A[i] = j$ . Dann ist  $S[j]$  **das zweite Zeichen von dem Suffix an Position j-1**
  - Sei j-1 in Bucket B mit aktuellem Pointer  $P^B$ 
    - B müssen wir finden – man braucht eine „Backpointer-“Liste
  - Da wir A sortiert durchlaufen, ist das Suffix ab Position j-1 das **nächst größere** in B
  - Also **schieben wir j** an die Stelle  $P^B$  in seinem Bucket und inkrementieren  $P^B$

# Beispiel

12345678  
CBCZCABZ...



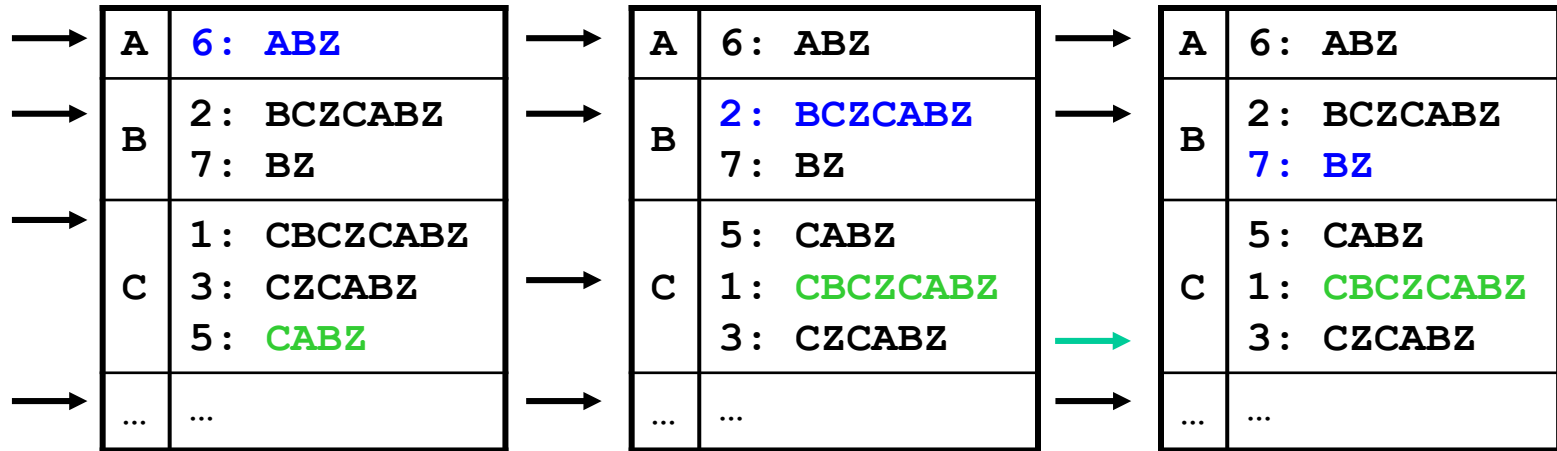
$i=1; A[1]=6; S[5..6]=CA$

Innerhalb des C-Buckets ist das Suffix  $S[5..6]$  also das kleinste, denn seine Verlängerung 'A' haben wir als erstes gefunden, wenn wir alle Suffixe lexikographisch durchlaufen (i)

$S[5..6]$  in seinem Bucket nach vorne schieben

# Beispiel

12345678  
CBCZCABZ...



$i=2; A[2]=2;$   
 $S[1..6]=CBCZCABZ$   
Keine Verschiebung,  
Pointer wandert

# Beispiel 2

---

123456789012
MISSISSIPPI\$

\$	12: \$
I	2: ISSISSIPPI\$ 5: ISSIPPI\$ 8: IPPI\$ 11: I\$
M	1: MISSISSIPPI\$
P	9: PPI\$ 10: PI\$
S	3: SSISSIPPI\$ 4: SSISSIPPI\$ 6: SSIPPI\$ 7: SIPPI\$

# Beispiel 2

- In jedem Bucket wird nach dem zweiten Zeichen sortiert
- Durch das Array laufen. Sei  $A[i]=j$ . Dann schiebe den Index  $j-1$  an den Anfang in seinem Bucket
- Alle Buckets werden in einem Durchlauf nach dem 2ten Zeichen sortiert
- Oftmals entstehen neue Buckets

A	1	2 3 4 5	6	7 8	9 10 11 12
	12	2 5 8 11	1	9 10	3 4 6 7
12		11			
2			1		4
5					4 7
8				10	
11		11 8			
1				10 9	
9		11 8 2			4 7 3
10					
3		11 8 2 5			4 7 3 6
4					
6					
7					
	12	11 8 2 5	1	10 9	4 7 3 6
	\$	I I I I	M	P P	S S S S
		\$ P S S	I	I P	I I S S
		P S S	S	\$ I	S P I I

# Sortierung nach 3. und 4. Zeichen

---

- Alle Elemente eines B haben dasselbe 2-buchstabige Präfix
- Nächster Schritt
  - Seien  $S[i..] = s_i \dots s_m\$$  und  $S[j..] = s_j \dots s_m\$$  wie vorher
  - Man müsste nun  $s_{i+2}$  mit  $s_{j+2}$  vergleichen
  - Das ist das gleiche wie  $S[i+2..] = s_{i+2} \dots s_m\$$  mit  $S[j+2..] = s_{j+2} \dots s_m\$$
  - Die  $S[i+2..]$  und  $S[j+2..]$  haben wir schon sortiert, und zwar **nach den ersten beiden Zeichen**
- Wir sortieren simultan nach dem 3. und 4. Zeichen
  - Wir gehen wieder durch das Array
  - Sei  $A[i] = j$ . Dann ist  $S[j]$  **das dritte Zeichen von  $S[j-2 \dots]$**
  - Schiebe  $j-2$  an Start seines Buckets und verschiebe Pointer um eins
- Dann nach 5-8. Zeichen etc.

# Beispiel

- Wert  $j-2$  verschieben
- Ob neue Buckets entstehen, kann man zur Laufzeit bemerken
- Aufhören, wenn alle Buckets nur noch ein Suffix enthalten

A	1	2	3	4	5	6	7	8	9	10	11	12
	12	11	8	2	5	1	10	9	4	7	3	6
12							10					
11								9				
8											6	
2											6	3
5												
1												
10			8									
9									7			
4				2								
7				2	5							
3						1						
6									7	4		
	12	11	8	2	5	1	10	9	7	4	6	3
	\$	I	I	I	I	M	P	P	S	S	S	S
		\$	P	S	S	I	I	P	I	I	S	S
			P	S	S	S	\$	I	P	S	I	I
			I	I	I	S		\$	P	S	P	S
			\$	S	P	I			I	I	P	S



# Andere Konstruktionsalgorithmen

- Puglisi, Smyth, Turpin (2007). "A taxonomy of suffix array construction algorithms." *ACM Computing Surveys* **39**(2).
- In der **Praxis schnellste Algorithmen** haben keinen linearen WC
  - „in practise, the behaviour of all the algorithms is linear“
- Platzverbrauch weniger als **halb so groß** als bei Suffix-Trees
- Indexierung von  $\sim 100\text{MB}$  möglich in  $\sim 50\text{-}60$  sec

Algorithm	Worst Case	Time	Memory
Prefix-Doubling			
MM [Manber and Myers 1993]	$O(n \log n)$	30	$8n$
LS [Larsson and Sadakane 1999]	$O(n \log n)$	3	$8n$
Recursive			
KA [Ko and Aluru 2003]	$O(n)$	2.5	$7 - 10n$
KS [Kärkkäinen and Sanders 2003]	$O(n)$	4.7	$10\text{-}13n$
KSPF [Kim et al. 2003]	$O(n)$	–	–
HSS [Hon et al. 2003]	$O(n)$	–	–
KJP [Kim et al. 2004]	$O(n \log \log n)$	3.5	$13\text{-}16n$
N [Na 2005]	$O(n)$	–	–
Induced Copying			
IT [Itoh and Tanaka 1999]	$O(n^2 \log n)$	6.5	$5n$
S [Seward 2000]	$O(n^2 \log n)$	3.5	$5n$
BK [Burkhardt and Kärkkäinen 2003]	$O(n \log n)$	3.5	$5\text{-}6n$
MF [Manzini and Ferragina 2004]	$O(n^2 \log n)$	1.7	$5n$
SS [Schürmann and Stoye 2005]	$O(n^2)$	1.8	$9\text{-}10n$
BB [Baron and Bresler 2005]	$O(n\sqrt{\log n})$	2.1	$18n$
M [Maniscalco and Puglisi 2006a]	$O(n^2 \log n)$	1.3	$5\text{-}6n$
MP [Maniscalco and Puglisi 2006b]	$O(n^2 \log n)$	1	$5\text{-}6n$
Hybrid			
IT+KA	$O(n^2 \log n)$	4.8	$5n$
BK+IT+KA	$O(n \log n)$	2.3	$5\text{-}6n$
BK+S	$O(n \log n)$	2.8	$5\text{-}6n$
Suffix Tree			
K [Kurtz 1999]	$O(n \log \sigma)$	6.3	$13\text{-}15n$

# Zusammenfassung

---

- Suffixarrays
  - Relativ **wenig Speicher: ~6m Bytes**
  - In der Praxis lineare Konstruktion
  - Schnelle Suche (aber nicht linear)
- Aber: Keine Informationen zu „Least common ancestors“
  - Längste gemeinsame Präfixe verschiedener Suffixe
  - Das sind innere Knoten im Baum – gibt's nicht im Suffixarray
  - Viele Anwendungen von Suffixbäumen nicht übertragbar (z.B. longest common substring)
- Möglichkeit: **Enhanced Suffixarrays**

# Inhalt dieser Vorlesung

---

- Suffixarrays
  - Idee und Suche
  - Konstruktionsalgorithmus nach Manber / Myers
- Enhanced Suffix Arrays (Sketch)

# Enhanced Suffixarrays

---

- Abouelhoda, Kurtz, Ohlebusch (2004). "Replacing suffix trees with enhanced suffix arrays." *Journal of Discrete Algorithms* **2**(1): 53-86
  - Auch: Enno Ohlebusch: „Bioinformatics Algorithms“, Olhebusch Verlag, 2015
- Suche in  $O(n * |\Sigma|) \sim O(n)$
- Platzverbrauch:  $\sim 8m$
- Genauso mächtig wie Suffixbäume
- Nicht unkompliziert – wir kratzen nur die Oberfläche

# Begriffe

---

- Definition

*Gegeben ein Suffixarray  $S$ . Ein Intervall  $[i, j]$  heißt lcp-Intervall mit lcp-Wert  $\alpha$  wenn gilt:*

- *Alle Suffixe, die zwischen den Indexpositionen  $i$  bis  $j$  in  $A$  liegen, haben ein **gemeinsames Präfix der Länge  $\alpha$***
- *$i$  und  $j$  sind die maximalen Werte, für die das gilt*
  - *Also:  $A[i][1..\alpha] \neq A[i+1][1..\alpha]$  und  $A[j][1..\alpha] \neq A[j+1][1..\alpha]$*
- *$\alpha$  lässt sich nicht vergrößern, ohne die Bedingung zu verletzen*
  - *Also:  $\exists k, i+1 < k < j: A[k][1..\alpha+1] \neq A[k-1][1..\alpha+1]$*

- Bemerkungen

- lcp-Intervalle kodieren die längsten gemeinsamen Präfixe zweier oder mehrerer Suffixe – also die **inneren Knoten** im Suffixbaum
- Wir schreiben das als  $\alpha-[i, j]$

# Beispiel

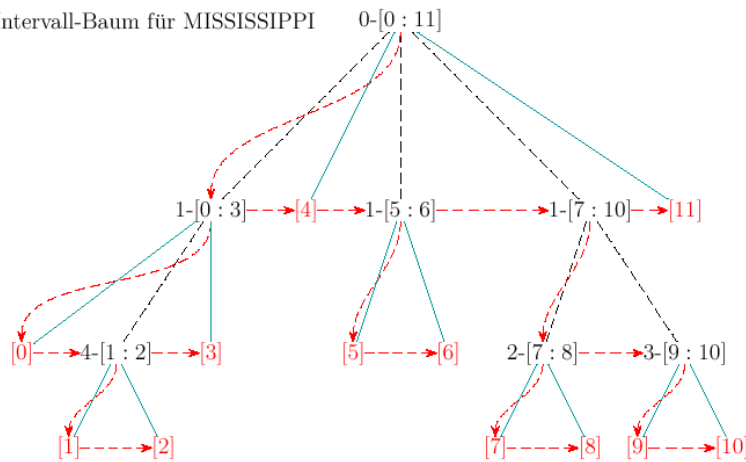
$A[0]$	=	08	$\hat{=}$	IPPI\$		
$A[1]$	=	05	$\hat{=}$	ISSIPPI\$		
$A[2]$	=	02	$\hat{=}$	ISSISSIPPI\$	4-[1 : 2]	1-[0 : 3]
$A[3]$	=	11	$\hat{=}$	I\$		
$A[4]$	=	01	$\hat{=}$	MISSISSIPPI\$		
$A[5]$	=	10	$\hat{=}$	PI\$	1-[5 : 6]	
$A[6]$	=	09	$\hat{=}$	PPI\$		
$A[7]$	=	07	$\hat{=}$	SIPPI\$	2-[7 : 8]	
$A[8]$	=	04	$\hat{=}$	SISSIPPI\$		1-[7 : 10]
$A[9]$	=	06	$\hat{=}$	SSIPPI\$	3-[9 : 10]	
$A[10]$	=	03	$\hat{=}$	SSISSIPPI\$		
$A[11]$	=	12	$\hat{=}$	\$		

Quelle: [AKO04]

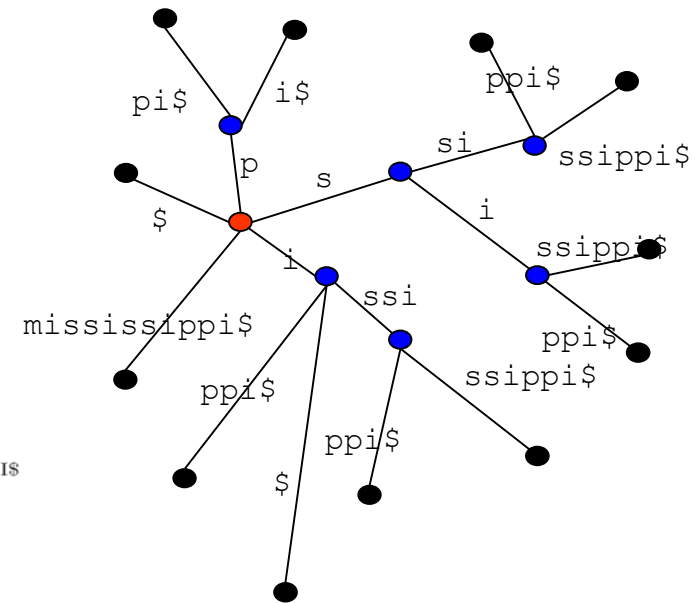
# Lcp-Baum

- Offensichtlich enthalten sich lcp-Intervalle gegenseitig
  - Innere Intervalle haben größere  $\alpha$ -Werte
- Gemeinsam bilden alle **lcp-Intervalle einen Baum**
  - Entspricht dem **Suffixbaum ohne Blätter**

LCP-Intervall-Baum für MISSISSIPPI



A[0]	=	08	≙	IPPI\$
A[1]	=	05	≙	ISSIPPI\$
A[2]	=	02	≙	ISSISSIPPI\$
A[3]	=	11	≙	I\$
A[4]	=	01	≙	MISSISSIPPI\$
A[5]	=	10	≙	PI\$
A[6]	=	09	≙	PPI\$
A[7]	=	07	≙	SIPPI\$
A[8]	=	04	≙	SISSIPPI\$
A[9]	=	06	≙	SSIPPI\$
A[10]	=	03	≙	SSISSIPPI\$
A[11]	=	12	≙	\$



# Noch ein Array

---

- Den lcp-Baum wollen wir nicht explizit bauen
  - Sonst kann man gleich den Suffix-Tree bauen
- ESA verwenden stattdessen ein Array: `child[1...m]`
  - Sei  $\alpha$ -[i,j] das längste lcp-Intervall, das an Position j endet
  - In `child[j].down` merken wir uns die Position des ersten Kind-Intervalls
  - ... und von jedem Kind-Intervall in `child[...].next` die **Position seines lexikographisch nächsten Geschwisters**
- Folgerung: Über `child[...].down` und `child[...].next` können wir **alle Kind-Intervalle eines lcp-Intervalls** traversieren
  - Beachte: Da die ersten Suffixe eines Kind-Intervalls nur einen Präfix der Länge  $\alpha$  mit ihrem Vorgänger gemeinsam haben, beginnen die Suffixe (nach  $\alpha$ ) aller Kindintervalle mit einem anderen Zeichen



# Child-Array

---

- Klar ist: Es gibt weniger als  $m$  lcp Intervalle
  - Jedes entspricht einem inneren Knoten im Suffixbaum
  - Jeder Baum mit  $\text{Grad} \geq 2$  hat höchstens so viele innere Knoten wie Blätter
- Alle child-Infos passen also in ein  $|m|$  Array - nur wie?
  - Man muss `child.down` und `child.next` in konstanter Zeit finden
  - An einer Position können mehrere Kindintervalle beginnen, aber wir wollen nicht mehrere next-Werte speichern
  - Das Child-Array muss in linearer Zeit gebaut werden
  - ...
- Details: Siehe [AKO04]

# Pattern-Matching mit ESA ( $|P|=m$ )

```
c := 0
queryFound := True
(i, j) := getInterval(0, n, P[c])
while (i, j) ≠ ⊥ and c < m and queryFound = True
  if i ≠ j then
    ℓ := getlcp(i, j)
    min := min{ℓ, m}
    queryFound := S[suftab[i] + c..suftab[i] + min - 1] = P[c..min - 1]
    c := min
    (i, j) := getInterval(i, j, P[c])
  else queryFound := S[suftab[i] + c..suftab[i] + m - 1] = P[c..m - 1]
if queryFound then
  Report(i, j) /* the P-interval */
else print "pattern P not found"
```

Springt Kinder entlang

$O(|P|)$  (nicht  $|S|$ )

Vergleicht nur  
„neue“ Zeichen

„Implizite Antwort“:  
Alle Matches liegen  
A zw i und j

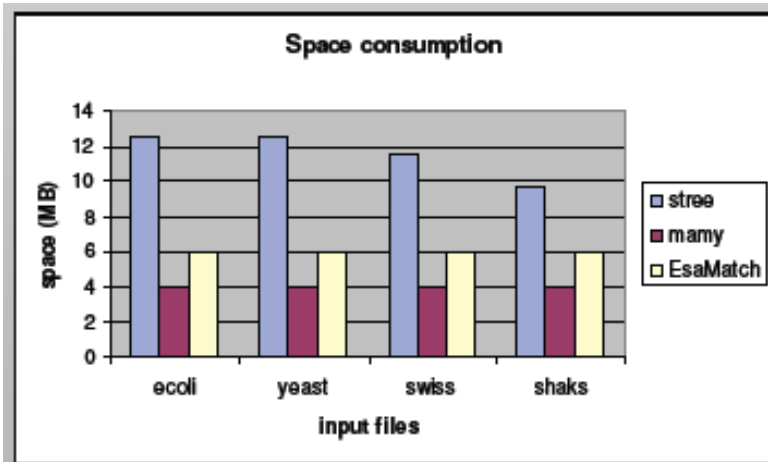
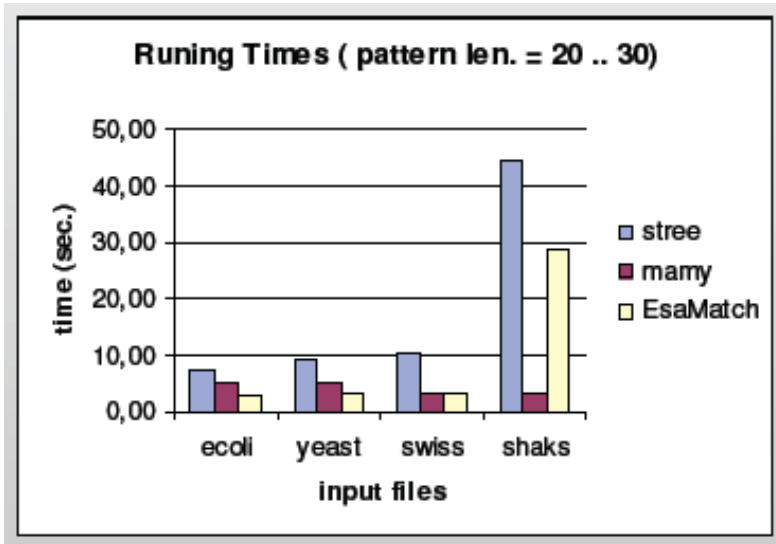
- $\text{getInterval}(i, j, c)$ : Finde das Kindintervall von  $\alpha$ - $[i, j]$ , dass an Position  $\alpha+1$  mit Zeichen  $c$  fortsetzt
- $\text{getlcp}(i, j)$ : Finde den  $\alpha$ -Wert für das Intervall  $\alpha$ - $[i, j]$

# Komplexität

```
(i, j) := getInterval(0, n, P[c])
while (i, j) ≠ ⊥ and c < m and queryFound = True
  if i ≠ j then
    ℓ := getlcp(i, j)
    min := min{ℓ, m}
    queryFound := S[suftab[i] + c..suftab[i] + min - 1] = P[c..min - 1]
    c := min
    (i, j) := getInterval(i, j, P[c])
  else queryFound := S[suftab[i] + c..suftab[i] + m - 1] = P[c..m - 1]
if queryFound then
```

- Man muss höchstens  $|P|$ -mal im Intervallbaum absteigen
  - Bei jedem Abstieg (getInterval) matched man  $\geq 1$  Zeichen von  $P$
- getInterval() muss höchstens  $|\Sigma|$  mal child[].next aufrufen
  - Start: Aufruf von child[].down
  - Traversiere alle weiteren Kinder
  - Jedes „beginnt“ mit einem anderen Zeichen (nach Position  $\alpha$ )
- Zusammen:  $O(|P| * |\Sigma|) = O(|P|)$  (bzw.  $O(|P| + k)$ )

# Experimente [AKO04]



- Eine Million Anfragen, zufällige Query der Länge 30-40, Laufzeit in Sekunden
- ESA schlechter als stree/mamy für große Alphabete
  - stree: Suffixtree
  - mamy: Suffixarray Variante
- ESA **schneller als Suffixarray/-baum** für kurze Alphabete
- Platzverbrauch zwischen Suffixarray und Suffixtree
  - SA: Ca. 4m (4 byte integer array)

# Selbsttest

---

- Wie viel Platz braucht ein Suffixarray?
- Suchkomplexität im Suffixarray?
- Warum ist es beim Manber/Mayr Algorithmus wichtig, dass Bucket-Sort ein stabiles Suchverfahren ist?
- Was macht das child-Array im Enhanced Suffixarray?
- Wie kann ein  $O(n \cdot \log(n))$  Algorithmus in der Praxis schneller als ein  $O(n)$  sein?
- Kann man Suffixarrays in-place bauen? Wie?