

Suffix Arrays

Eine kleine Wiederholung

Suffix Arrays

- Datenstruktur für Stringalgorithmen
- Aufbauzeit $O(n)$ Stringlänge n
- Platzbedarf $O(n)$, $4n$ in Praxis
- Suchzeit $O(m + \log n)$ für Suchwort der Länge m

Suffix Arrays

Aufbau

- Aufbaualgorithmus: D&C
- Dreiteilung durch modulo Operation
- Zwei Sortiermengen S_C und S_0
- „Geschicktes“ Sortieren
- Einfaches Mergen

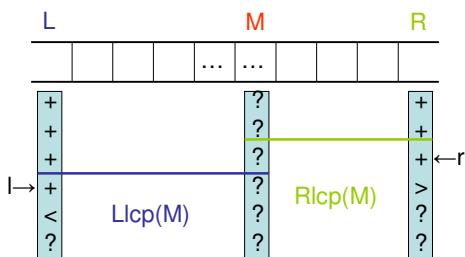
Suffix Arrays

Suche

- $O(m \log n)$ naiv - Binärsuche
- $O(m + \log n)$ mit Preprocessing:
Longest common Prefix $Llcp(M)$, $Rlcp(M)$
Übereinstimmung der Mitte mit den Rändern

Suche

Beschleunigung II



$lcp(p, S_{SA[M]})?$

Suche links oder rechts von M?

Suche

Beschleunigung II

Statische Vorberechnung der $Llcp/Rlcp$ -Werte

Wie berechnet man $Llcp$, $Rlcp$?

n verschiedene M , dabei L, R fix

Suche

Beschleunigung II

Berechnung von Lcp, Rlcp

$lcp = lcp(i, i+1)$

$lcp(i,j) = \min\{lcp(k-1, k) : k \in [i+1,j]\}$

				L ₂	M ₃	L ₁ /M ₂	M ₁	R ₁ /R ₂
--	--	--	--	----------------	----------------	--------------------------------	----------------	--------------------------------

Suche

Beschleunigung II

				M	...		M'	...	R
--	--	--	--	---	-----	--	----	-----	---

$Rlcp(M) = \min\{Llcp(M'), Rlcp(M')\}$

Llcp analog

Bottom-up traversal der lcp-Werte

➔ Vorberechnung in Linearzeit wenn $lcp(i-1, i)$ bekannt

LCP-Tabelle

Berechnung der speziellen lcp-Tabelle L

$L[i] = lcp(i-1, i)$ für $i \in [1..n]$

R[i]-1: a [shaded] [] S_{SA[R[i]-1]}

R[i]: a [shaded] [] S_i

lcp

R[i+1]-1: [shaded] [light blue] [] S_{SA[R[i+1]-1]}

R[i+1]: [shaded] [light blue] [] S_{i+1}

lcp

SA	11	8	5	2	1	10	9	7	4	6	3
----	----	---	---	---	---	----	---	---	---	---	---

Unterschied: $lcp = k$

i++

LCP-Tabelle

Vorgängerbeziehung:

$lcp(R[i+1], R[i+1] - 1) \geq lcp(R[i], R[i] - 1) - 1$

k = 0;

for i = 1 .. n

if R[i] > 0

j = SA[R[i] - 1]; //S_j lex. direkt vor S_i

while (t_{i+k} = t_{j+k}) k++; //Übereinstimmung

L[R[i]] = k;

k = max(0, k-1); //Formel

Suche

Beschleunigung II

Fazit: Preprocessing mit Berechnung von

- lcp-Wert Array L
- Llcp- und Rlcp-Werten

ist in Linearzeit möglich!!

➔ Suche in $O(m + \log n)$ mit linearem Aufbau

Suffix Arrays

Wir haben:

- Aufbau in Linearzeit
- Suche in Linearzeit

Wir brauchen:

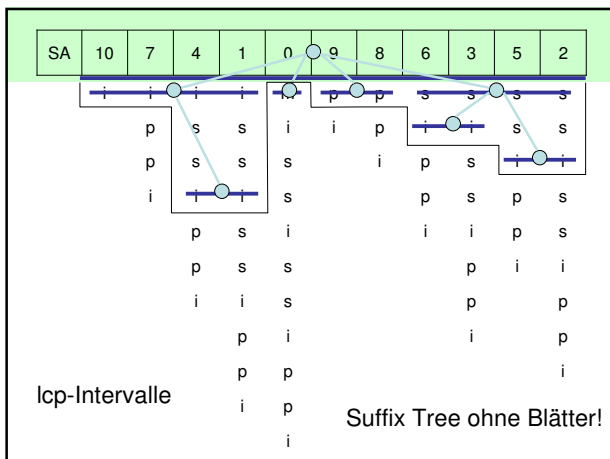
- Praktische Anwendung

Problem:

- Viele gute Algorithmen baumbasiert, z.B. mit bottom-up, top-down traversal

Enhanced Suffix Arrays

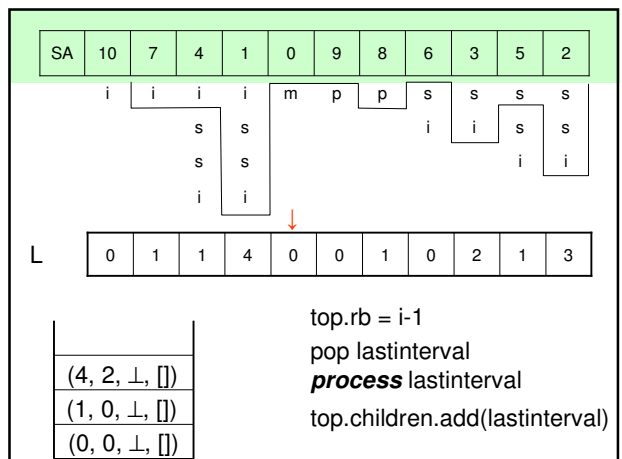
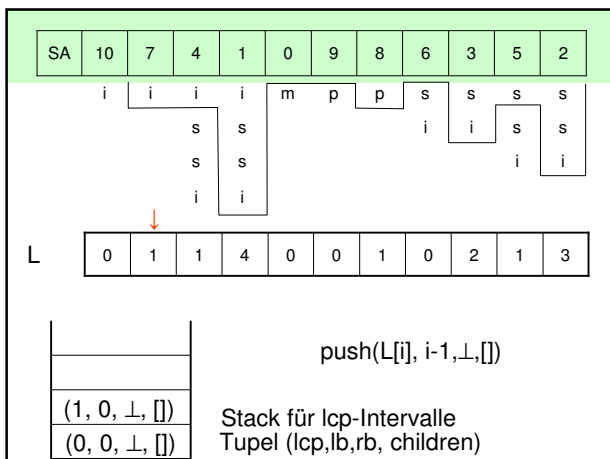
- Idee: Baumverfahren wie bottom-up-traversal in Suffix Arrays abbilden
- Speichern lcp-Tabelle L mit
- Immer noch Platzvorteil



Enhanced Array

Bausteine

- Suffix Array SA
- Lcp Tabelle L
- Lcp Intervall Baum IB (konzeptuell)
- Traversal des IB ohne Aufbau



LCP Tabelle

- Problem: Verdopplung des Speicherplatzbedarfs
- Idee: In Praxis $lcp \ll \maxInt$

L	L+
17	(2, 328)
94	(4, 455)
255	
23	
255	

$n + k$ bytes

Zugriff in $\log(|L+|)$
für random access,
sonst konstant

Fallstudie: Repeat Analyse

- Repeats: Musterwiederholung
- Genomanalyse: ~ 50% Repeats bei Mensch
- Nötig für Genomvergleich, Alignment
- Vermutungen über funkt. oder evol. Rolle

dfgdds**saard**wiensa**saare**mscherprim**s**

Repeat Analyse

Grundlagen

- Repeat: $(i_1, j_1) \neq (i_2, j_2)$, aber $S(i_1..j_1) = S(i_2..j_2)$
 $(i_1, j_1), (i_2, j_2)$ *repeated pair*
- Maximal Repeat: Kein Teil eines größeren Repeat

dfgdds**saard**wiensa**saare**mscherprim**s**

Repeat Analyse

Grundlagen

Berechnung maximal repeat pairs:

- Algorithmus von Gusfield
- Laufzeit $O(kn+z)$, $k = |\Sigma|$, z Anzahl Repeats
- Laufzeit optimal!
- Erste Implementierung in *REPuter* (Suffix Trees)

Repeat Analyse

Bausteine für Verbesserung

- Suffix Array SA
- Lcp Array L
- Burrows and Wheeler Transformation Array BW
- Lcp Intervall Baum-Konzept
- Nicht Inputstring!!

Idee: Sequentieller Arraydurchlauf

➔ Laufzeitverbesserung trotz Platzersparnis

Repeat Analyse

Bausteine für Verbesserung

Burrows and Wheeler transformation array:

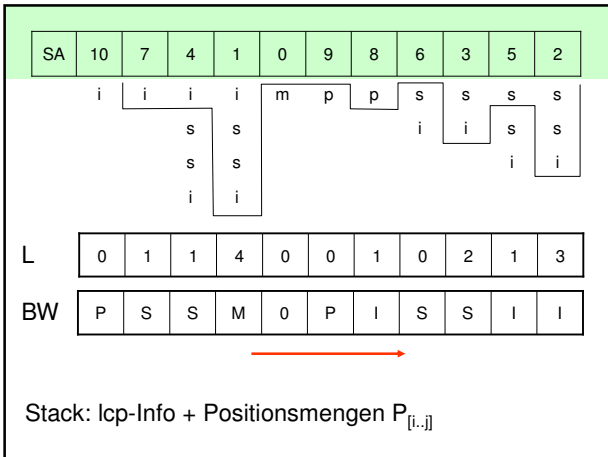
- $BW[i] = T[SA[i]-1]$, falls $SA[i] \neq 0$
- Speicherung in n bytes (Character)
- Aufbau in $O(n)$ mit Lauf über SA

$[i..j]$ l-Intervall, u zugehöriger lcp-String der Länge l

$P_{[i..j]} = \{SA[r] \mid i \leq r \leq j\}$ Anfangspos. mit Präfix u

$P_{[i..j]}(a) = \{p \in P_{[i..j]} \mid T[p-1] = a\}$, $a \in \Sigma$ (disjunkt)

Basisfall $i=j$: $P_{[i..j]}(a) = p$ mit $T[p-1] = a$



Repeat Analyse

Schema

process-Funktion:

- Alle Kind-Positionsmengen bekannt! → Durchlauf
- Sei $P_{[i..j]}^k(a)$ Teilmenge nach k Kindern
- k+1tes Kind $[i'..j']$:

1. Maximal Repeated Pairs ausgeben
2. $P_{[i..j]}^{k+1}(a)$ für alle a berechnen

Repeat Analyse

Schema

- Maximal Repeated Pairs ausgeben:
Kombiniere Pos. $p = P_{[i..j]}^k(a)$ mit $p' = P_{[i'..j']}^l(b)$
für $a \neq b$
- u kommt an p und p' vor (Def.)
Davor steht $a \neq b$
Dahinter Unterscheidung, da aus versch. Kindern
- $P_{[i..j]}^{k+1}(a)$ berechnen:
 $P_{[i..j]}^{k+1}(a) = P_{[i..j]}^k(a) \cup P_{[i'..j']}^l(a)$
- k Unions pro Intervall, $O(n)$ Intervalle,
 $O(z)$ Kombinationen → $O(kn+z)$

Repeat Analyse

Vorteile

- Speicherung in Secondary Memory – Kein Random Access nötig!!
- Verbesserung der Laufzeit (Cache- Verhalten!)
- Verbesserung des Platzbedarfs
- Vereinfachung der Implementierung

Repeat Analyse

Laufzeit

l	#reps	REP	SA	Platz
20	7799	3.28	0.79	REP
23	5206	3.28	0.78	61 Mb
27	3569	3.31	0.79	
30	2730	3.30	0.80	SA
40	840	3.29	0.79	31 Mb
50	607	3.29	0.79	

Escherichia coli, DNA-Länge 4.639.221, $\alpha = 4$

Kompression

Lempel-Ziv

- Gruppe verlustfreier Kompressionmethoden
- *Substituierende* Kompression:
Ersetze Repeats durch Verweis auf voriges Vorkommen
- *Dekomposition* zur Erkennung von Repeats

Lempel-Ziv

- Sei l_i Länge des längsten Präfix von S_i , der schon vorkommt
- Position s_i : Erstes solches Vorkommen
- Dekomposition: Erzeuge solche Präfixe durch Blockdekomposition
- Folge von Anfangsindizes i_1, \dots, i_k
Induktiv: $i_1 = 0, i_{B+1} = i_B + \max\{1, l_{i_B}\}$
- $T[i_B..i_{B+1}-1]$ heißt B-ter Block der Ziv-Lempel Dekomposition

Lempel-Ziv

- Dekomposition durch bottom-up traversal des lcp-Intervall Baums
- Neuer Stackwert: min
- **process**-Funktion für l-Intervall $[i..j]$:
Seien min_1, \dots, min_k Werte für schon bearbeitete Kinder
 $M = \{min_1, \dots, min_k\} \cup \{SA[q] \mid q \in [i..j] \text{ und nicht in Intervallen } 1..k\}$
 $min = \min M$
 $\forall q \in M, q \neq min: s_q = min, l_q = l$
 An Wurzel $[0..n]: \forall q \in M s_q = 0, l_q = 0$

Lempel-Ziv

T	a	c	a	a	a	c	a	t	a	t	\$
i	0	1	2	3	4	5	6	7	8	9	10
s_i	0	0	0	2	0	1	0	0	6	7	0
l_i	0	0	1	2	3	2	1	0	2	1	0

B	1	2	3	4	5	6	7	8
i_B	0	1	2	3	5	7	8	10
Block	a	c	a	aa	ca	t	at	\$

Hauptpunkte

- Suffix Array ist lexikographisch geordneter Array der Suffixe eines Strings
- Aufbau in Linearzeit
- Platzeffiziente Datenstruktur (~ 4Byte)
- Anwendung: Stringvergleiche, Suche
- Suche in Zeit $O(m + \log n)$
- Amortisierung des Aufbaus über viele Suchen (Indexierung)

Ende!