Suffix Arrays – Eine Datenstruktur für Stringalgorithmen

Karsten Klein

Vorlesung

Algorithmen und Datenstrukturen

WS 08/09 – 11.November 2008

TU Dortmund, Fakultät Informatik, LS11 Algorithm Engineering

Überblick

- Suffix Array – Anwendungsfelder, Struktur
- Aufbau:

  Algorithmus von Kärkkäinen, Sanders, Burkhardt 2004
- Suchen in Suffix Arrays

  Naiv, Beschleunigung
- Anwendungen

Problemstellung

- Suche von Zeichenketten in Text: „String-Matching“
- In Bioinformatik:
  - Proteinidentifikation (20 Zeichen Alphabet)
  - Genom Alignment (4 Zeichen Alphabet)
  - Repeatsuche
- Kompression: Lempel-Ziv (GIF, PDF,…), Burrows-Wheeler (BZIP2)
- Circular String Linearization in der (Bio)Chemie
- Web Suche, Wissensnetze

TAVIKKLKQINEEQREGLMDDLRGVNLSKF

INEEQREG ?

Problemstellung

- Hürde: Riesige Datenmengen (menschl. Genom ca. 3 Mrd. Basenpaare)
- Quadratisch viele Substrings – nicht explizit speicherbar
- Klassische Algorithmen überfordert (Suchzeit linear)

„Lösung“

Suffix Array:

- Platzsparende Datenstruktur
- Schneller Aufbau (amortisiert über Suchen)
- Schnelle Suche (sublinear)
- Gut für External Memory Nutzung
- Flexibel auch als Ersatz z.B. für Suffix Tree und viele spezielle Fragestellungen

Engineering Aspekte

- In Praxis: Komplexität kritisch
- Zwar: Speicherplatzbedarf geringer als bisherige Lösungen und asympt. „schnell“
- Aber: Schon Konstanten kritisch ➔ Rallye nach impl. Verbesserungen für
  - Speicherplatz (Aufbau)
  - Laufzeit (Aufbau)
  - Laufzeit (Suche), Halten in Hauptspeicher
- Prakt. Effizienz hängt ab von Textlänge, Musterlänge, Alphabet, Anzahl Suche/Repeats

Lösungsansatz

Gegeben String *T* der Länge *n* über Alphabet Σ

- Jeder Teilstring ist Präfix eines Suffix

INEEQREG

TAVIKKLKQINEEQREGLMDDLRGVNLSKF$

Es gibt nur *n* Suffixe!

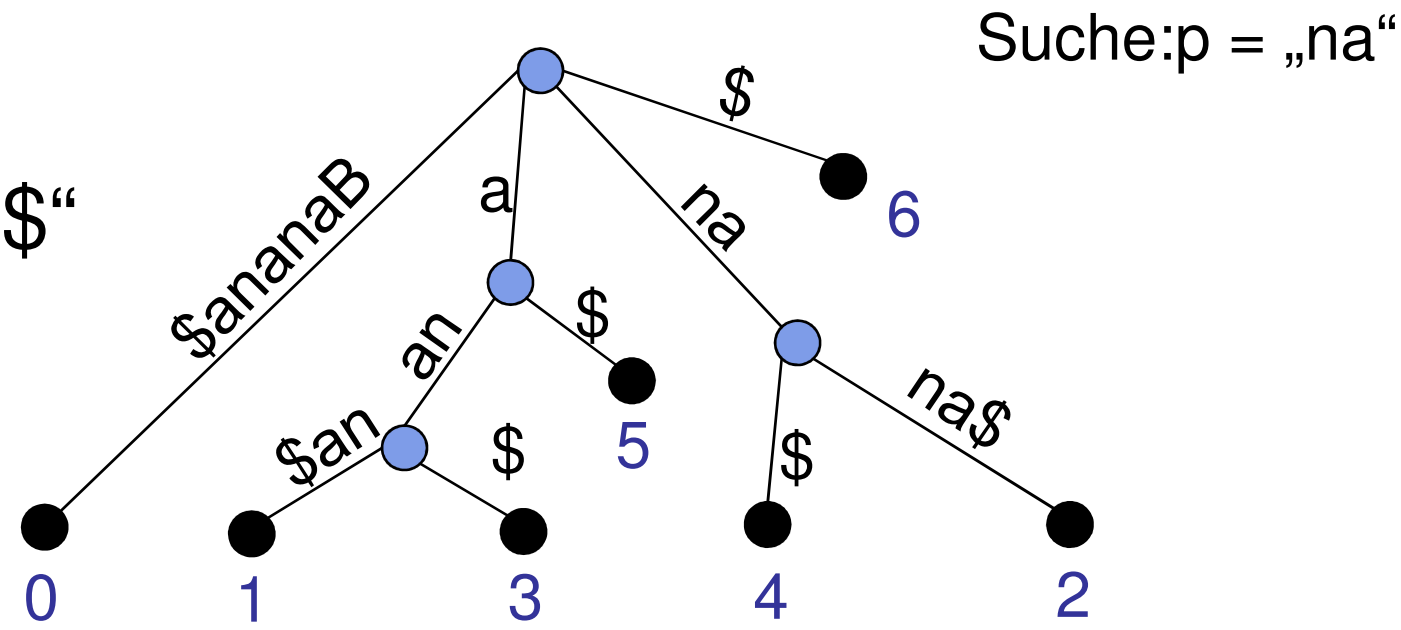
- ⇒ Nur Suffixe anschauen, Präfix testen
- Eindeutigkeit?
  ⇒ Kein Suffix als Präfix von Suffix: Sentinel $

Exkurs:Suffix Tree (McCreight/Weiner 73)

Idee: Darstellung aller Suffixe als Baum

„Banana$“

Suche:p = „na“



Bei endl. Alphabet Suche in *O(|p|+#Treffer)*
(Konkrete Impl. siehe z.B. [Kurtz ´99])

Suffix Tree – Vor-/Nachteile

- Sehr gut untersucht, sehr viele Verbesserungen
- Zeit für Bau/Suche linear, abhängig von Alphabetgröße $O(n \log |\Sigma|)$ bzw. $O(|p| \log |\Sigma|)$
- Platzbedarf abhängig von Alphabet, 10-20 Bytes pro Zeichen
- Schlecht für Externspeicher anpassbar (Lokalität!)
- Implementierung (relativ) „aufwendig“

$\Rightarrow$ Suffix Array

Alles auch in gleicher asympt. Laufzeit möglich [Abouelhoda et al. 2004]

Suffix Array (Manber/Myers 91)

- Repräsentiert sortierte Liste der Suffixe in einem Integer Array (Bottleneck: Berechnung der Sortierung)
- Entwicklungsziel: Platzsparend für extrem grosse Datenmengen
- Aufbauzeit damals: *O(nlogn)* (direkt)
- Heute: Zeit linear, kleiner Faktor (große Menge verschiedener Algorithmen in kurzer Zeitspanne)
- Für Praxis besser als Suffix Tree, unabhängig von Alphabet, deutlich kleiner
- Suche sublinear
- Auch: PAT-Array (Gonnet et al.)

Sortierung von String-Suffixen

Ordnung auf Alphabet $\Rightarrow$ lexikographisch Sortieren

Banana
0 1 2 3 4 5

0	Banana	5	a
1	anana	3	ana
2	nana	1	anana
3	ana	0	Banana
4	na	4	na
5	a	2	nana
		SA	

SA[i]: An welcher Position beginnt i-ter Suffix in Sortierung?

Sortierung von String-Suffixen

Ordnung auf Alphabet $\Rightarrow$ lexikographisch Sortieren

Banana
0 1 2 3 4 5

0	Banana	5	a
1	anana	3	ana
2	nana	1	anana
3	ana	0	Banana
4	na	4	na
5	a	2	nana
			SA

Suffix Array SA von s: Integerarray mit

SA[i] = j : Suffix S_j hat Rang i in lexikogr. Ordnung

Suffix Array - Beispiel

Ind	0	1	2	3	4	5	6	7	8	9	10
s	M	I	S	S	I	S	S	I	P	P	I

SA	10	7	4	1	0	9	8	6	3	5	2

Reverse Array R: SA[R[i]] = i

R	4	3	10	8	2	9	7	1	6	5	0

Suffix Arrays

- Textindex durch Suffixsortierung
- Speicherung als Integerarray
- |int|n bytes, Praxis: ~ 4n bytes + n bytes Text
- Größe unabhängig von Alphabetgröße

Algorithmen:

- Aufbau
- Suche
- Anwendungen: Repeat Suche, Kompression,...

Aufbau von Suffix Arrays

Aufbauvarianten

- Traversal des Suffix Tree: Aufwendig, Platzbedarf ≥15n, Alphabetabhängig, Zeit *O(n)*
- Einfacher Aufbau durch Sortierung
  Achtung: Sortierung von Strings!
  ⇒Ternary (Multikey) Quicksort oder Bucketsort mit Tricks, Worst-case Zeit *O(nlogn)* (Theor.)
- „Geschicktere“ Sortierung in Linearzeit?

Viele Variationen unter Ausnutzung der Suffixeigenschaft (Larsson-Sadakane, Itoh-Tanaka, Seward, Burkhardt-Kärkkäinen, ...)

Aufbau

Divide & Conquer / Rekursion

Linearzeitalgorithmus *DC3* von Kärkkäinen, Sanders, Burkhardt 2004

Schema DC3

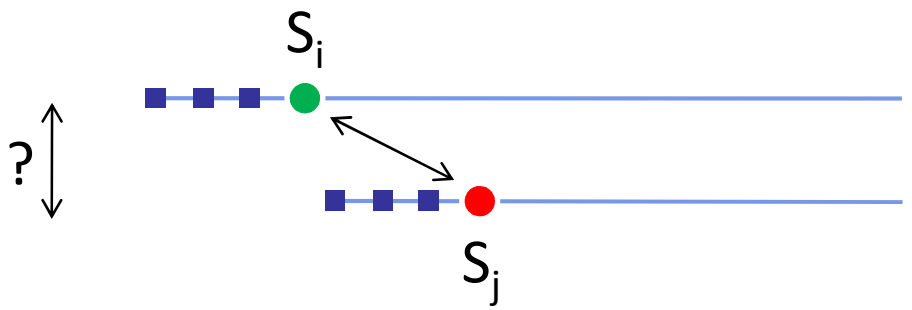
1. Partitioniere Suffixe in Auswahlmenge („*Sample*“) und Rest
2. Erzeuge Sortierung für Sample
3. Sortiere Rest
4. Merge die erzeugten Arrays

Welche Partitionen?
Wie kann man Merge durchführen?

Ideen

- Anforderung an Sample und Mergen:
  1. Sample ist leicht zu sortieren ($\Rightarrow$ nicht zufällig)
  2. Sample nützlich für Restsortierung:
     Suffixstruktur + bekannte Teilsortierung ausnutzen

$S_0 = t_0t_1t_2t_3\ldots$
$S_1 = \quad t_1t_2t_3\ldots$
$S_2 = \quad\quad t_2t_3\ldots$
$S_3 = \quad\quad\quad t_3\ldots$



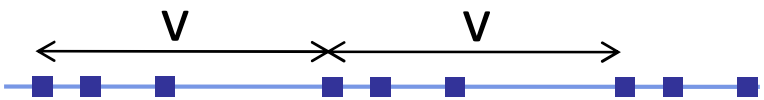
Difference Cover

- $D \subseteq [0, v)$ *Difference cover* modulo v wenn

  $$\{ (i - j) \bmod v \mid i, j \in D \} = [0, v)$$

  „Erzeugt [0,v) durch Differenzen aus D“
- C *v-Periodisches Sample* von [0, n] mit Periode D:

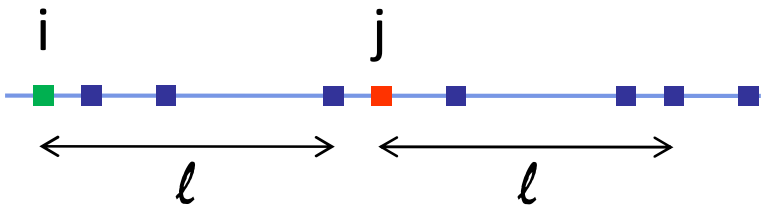
  $$C = \{ i \in [0, n] \mid i \bmod v \in D \}$$



Difference Cover

Difference cover, v-Periodisches Sample?

- Falls die Periode des Sample ein Difference Cover ist, bedeutet das: Für beliebige i,j existiert ein $\ell \in [0, v)$ mit $(i + \ell) \bmod v$ *und* $(j + \ell) \bmod v$ in D,
  also Sample Positionen in kurzem Abstand.
- Für S_i, S_j: $\exists$ innerhalb Abstand $\ell \leq v$ Paar S_{i+k},S_{j+k} dessen Vergleich bekannt ist $\Rightarrow$ Nur $\leq \ell$ Zeichen vergleichen.



Aufbau

Linearzeitalgorithmus DC3

1. *Sortiere* Suffixe S_i mit: $i \bmod 3 \in \{1, 2\}$
   Reduziere Problem damit auf Suffixsortierung für Stringlänge $\frac{2}{3}n$
   Rekursion auf reduziertem Problem
2. *Sortiere* Suffixe S_i mit: $i \bmod 3 = 0$
   Basierend auf $i \bmod 3 = 1$ Sortierung
3. *Merge* die beiden Mengen
   Nutze vergleichsbasiertes Mergen mit Suffixvergleich in konstanter Zeit

Aufbau

Linearzeitalgorithmus DC3

1. Auswahlmengen : Dreiteilung der Indizes

   $B_k = \{i \in [0,..,n] \mid i \bmod 3 = k\}$ für $k = 0,1,2$

   Samplepositionen $C = B_1 \cup B_2$

   Samplesuffixe S_C

Beispiel

- Samplepositionen $i \bmod 3 \neq 0$:

mississippi

1: ississippi
2: ssissippi
4: issippi
5: ssippi
7: ippi
8: ppi
10: i

Samplepositionen $C = \{1,2,4,5,7,8,10\}$

Aufbau

Linearzeitalgorithmus

2. Erzeuge Sortierung für C (= $\frac{2}{3}n$)
3. Sortierung des Restes B_0 (mit $i \bmod 3 = 0$):

   Für Suffix S_i gilt $S_i = t_i S_{i+1}$ $\quad$ $i+1 \bmod 3 = 1$

$\Rightarrow$Statt Stringvergleich nutze bekannten Sortierungsrang für Suffixe S_{i+1}

Also für $i,j \in B_0$ Vergleich

$$S_i \leq S_j \Leftrightarrow (t_i, \text{Rang}(S_{i+1})) \leq (t_j, \text{Rang}(S_{j+1}))$$

Beispiel

Text s = mississippi

S_C

1: ississippi	4
2: ssissippi	
4: issippi	3
5: ssippi	
7: ippi	2
8: ppi	
10: i	1

S_{B0}

0: m-ississippi	(m,4)
3: s-issippi	(s, 3)
6: s-ippi	(s, 2)
9: p-i	(p, 1)

Ergebnis 0,9,6,3

Aufbau

Berechnung für Menge C

C: Positionen i mod 3 = 1,2 TRICK!

Rechnung modulo 3 ➔ Zeichentripel $[t_{i,i+1,i+2}]$

Tripel beginnen mit Zeichen mit gleichem mod-Rest!

$R_k = [t_{k,k+1,k+2}] \ldots [t_{maxB_k,maxB_{k+1},maxB_{k+2}}]$, k= 1,2

R_1 = iss iss ipp i$$

R = R1 ⊕ R2 = iss iss ipp i$$ ssi ssi ppi

Suffixe in S_C und R entsprechen sich!

S_i entspricht $[t_{i,i+1,i+2}][t_{i+3,i+4,i+5}]\ldots$

Berechnung für Menge C

Einfacher? Sortiere Tripel und benenne sie mit Rang

Eindeutig? ➔ Suffixsortierung nach erstem Tripel

Nicht eindeutig? ➔ Rekursiver Aufruf mit Größe $\frac{2}{3}$

3 3 2 1 5 5 4

iss iss ipp i$$ ssi ssi ppi

➔Rang für Suffixe rang(S_i) für R bzw. S_C

Laufzeit? $T(n) = T(\frac{2}{3}n)+O(n) \Rightarrow O(n)$

Aufbau

Linearzeitalgorithmus

2. Erzeuge Suffixarray für Positionen $i \bmod 3 \neq 0$
   Rekursiv mit Aufteilung in Problem der Größe 2/3
3. Erzeuge Suffixarray für $i \bmod 3 = 0$ unter Benutzung des Arrays aus 1.
4. Merge die zwei Mengen

Wie Merge ausführen?

Aufbau

Merge

- Standardmerge mit Durchlauf der Arrays
- Vergleiche $S_i \in S_C$ mit $S_j \in S_{B_0}$
    1. $i \in B_1$: $(t_i, \text{rang}(S_{i+1})) \leq (t_j, \text{rang}(S_{j+1}))$
    2. $i \in B_2$: $(t_i, t_{i+1}, \text{rang}(S_{i+2})) \leq (t_j, t_{j+1,} \text{rang}(S_{j+2}))$

Allgemeiner Ansatz

- DC3 ist ein Spezialfall:
  - Auswahl von C beruht auf „Difference Cover“ modulo v=3
  - Beliebiges v aus [1, √(n)] möglich
- DC erlaubt Zeit-Platz Tradeoff für Wahl von v:
  - Zeit O(vn)
  - Platz O(n / √(v)) ohne Eingabe / SA
- Samplegröße O(n/ √(v))

Aufbau

- Aufbau in *O(n)* Zeit und Platz
- Mehrere Algorithmen mit untersch. Eigenschaften bzgl. Platz/Zeit je nach Eingabe
- Aufbauzeit kann über mehrere Suchen amortisiert werden

- External Memory Algorithmen verfügbar: Crauser/Ferragina, Dementiev et al., …
- Komprimierte/Kompaktierte Suffix Arrays („succinct data structures“)

Suche?

Literatur

- U. Manber, G. Myers: Suffix Arrays: *A new method for on-line string searches*, SIAM J. Comp. 22 (5) 1993
- S. Burkhardt, J. Kärkkäinen, P. Sanders: *Linear work suffix array construction*, J. ACM 53 (6) 2006
- M. I. Abouelhoda, S. Kurtz, E. Ohlebusch: *Replacing suffix trees with enhanced suffix arrays*, J. Disc. Alg. 2 (1), 2004
- S. Kurtz: *Reducing the space requrements of suffix trees*, Soft. Prac. Ex 29 (13), 1999
- S. J. Puglisi, W.F. Smyth, A.H. Turpin: *A Taxonomy of suffix array construction algorithms*, ACM Comp. Sur. 39 (2), 2007