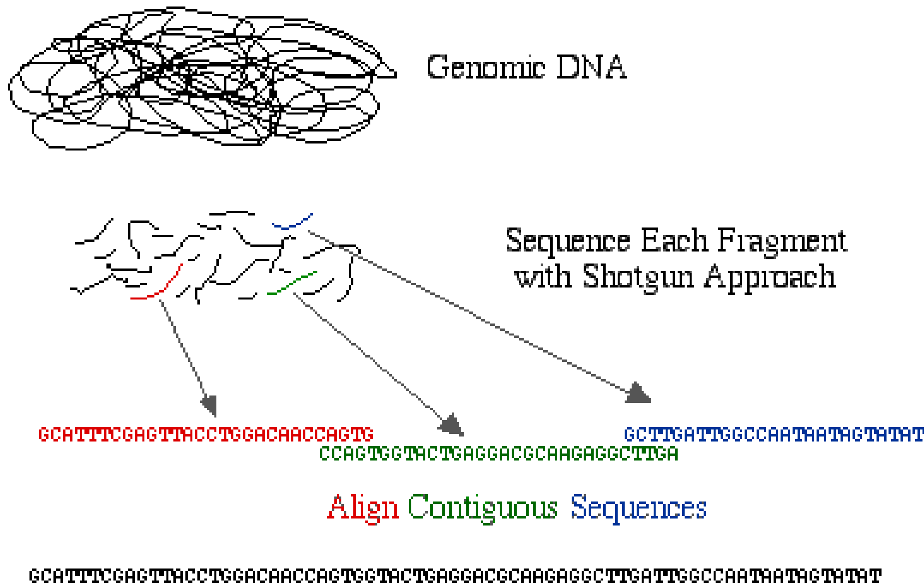
Bioinformatik

Suffixbäume
Suffixarrays

Ulf Leser

Wissensmanagement in der Bioinformatik

Problemdimension: Whole Genome Shotgun



- Zerbrechen von kompletten Genomen in Stücke 1KB-100KB
- Alle Stücke (an-) sequenzieren
- Celera:
  - Homo sap.: Genom: 3 GB, 28.000.000 Reads
  - Drosophila: Genom: 120 MB, 3.200.000 Reads

➢ Schnelle Algorithmen notwendig!

Exaktes Matching

- Gegeben: P (Pattern) und T (Text), $|P| \leq |T|$
- Gesucht: Sämtliche Vorkommen von P in T

Beispiel

Auffinden der Erkennungssequenzen von Restriktionsenzymen

Eco RV - GATATC

```
tcagcttactaattaaaaattctttctagtaagtgctaagatcaagaaaataaattaaaaataatggaacatggcacattttcctaaactcttcacagattgctaatga
ttattaattaaagaataaatgttataatttttatggtaacggaatttcctaaaatattaattcaagcaccatggaatgcaaataagaaggactctgttaattggtact
attcaactcaatgcaagtggaactaagttggtattaatactctttttacatatatatgtagttattttaggaagcgaaggacaatttcatctgctaataaagggattac
atatttattttgtgaatataaaaaatagaaagtatgttatcagattaaacttttgagaaaggtaagtatgaagtaaagctgtatactccagcaataagttcaaatagc
gaaaaacttttaataacaaagttaaataatcattttgggaattgaaatgtcaaagataattacttcacgataagtagttgaagatagtttaaattttcttttgtatt
acttcaatgaaggtaacgcaacaagattagagtatatatggccaataaggtttgctgtaggaaaattattctaaggagatacgcgagagggcttctcaaatttattcaga
gatggatgttttagatggtggtttaagaaaagcagtattaaatccagcaaaactagaccttaggtttattaaagcgaggcaataagttaattggaattgtaaaagatat
ctaattcttcttcatttgttggaggaaaactagttaacttcttaccccatgcagggccatagggtcgaatacgatctgtcactaagcaaaggaaaatgtgagtgtagact
ttaaaccattttattaatgactttagagaatcatgcatttgatgttactttcttaacaatgtgaacatatttatgcgattaagatgagttatgaaaaaggcgaatatat
tattcagttacatagagattatagctggtctattcttagttataggacttttgacaagatagcttagaaaataagattatagagcttaataaaagagaacttcttggaat
tagctgcctttggtgcagctgtaatggctattggtatggctccagcttactggttaggttttaatagaaaaattccccatgattgctaattatatctatcctattgagaa
caacgtgcgaagatgagtggcaaattggttcattattaactgctggtgctatagtagttatccttagaaagatatataaatctgataaagcaaaatcctggggaaaatat
tgctaactggtgctggtagggtttggggattggattatttcctctacaagaaatttggtgtttactgatatccttataaataatagagaaaaaattaataaagatgatat
```

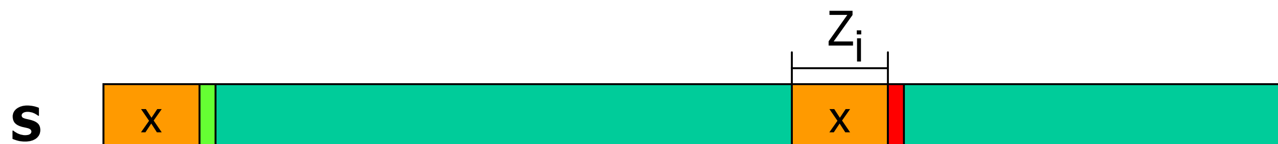
Naiver Algorithmus

- Exaktes Matching: P in T
- Naiver Algorithmus
  - Beginnt Vergleich von P an jedem Zeichen von T
  - Quadratische Komplexität

```
T    ctgagatcgcgta
P    gagatc
      gagatc
       gagatc
        gagatc
         gagatc
          gatatc
           gatatc
            gatatc
```

Z-Algorithmus: Preprocessing

- Im Folgenden: S statt P (Gusfield)
- Definition
  - *Sei i>1. Dann ist $Z_i(S)$ die Länge des größten Substrings x von S mit*
    - *x=S[i..i+|x|]* *(x startet an Position i in S)*
    - *S[i..i+|x|] = S[1..|x|]* *(x ist auch Präfix von S)*
  - *x heißt Z-Box von S an Position i mit Länge $Z_i(S)$*

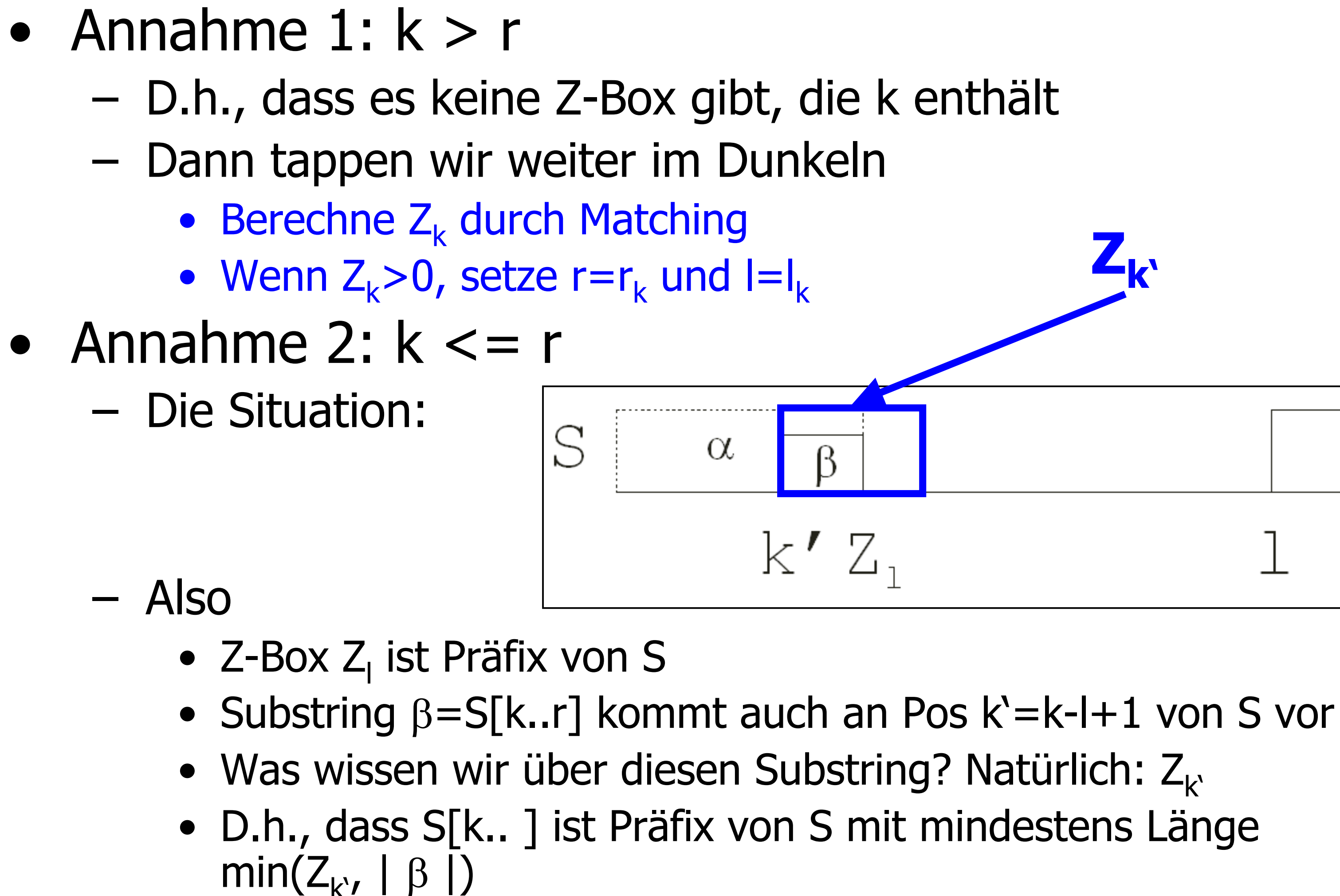


Lineare Berechnung der Z_i Werte

- Trick
  - Verwenden von bereits berechneten Z_i zur Berechnung von Z_k $(k > i)$
  - Lineares Durchlaufen des Strings
  - Kontinuierliches Vorhalten der Werte $l=l_{i-1}$ und $r=r_{i-1}$
  - Größe der Z-Box an Position i ergibt sich mit konstantem Aufwand
- Induktive Erklärung
  - Beginne mit i=2. Berechne Z_2. Wenn $Z_2 > 0$, setze $r=r_2$ und $l=l_2$, sonst r=l=0
  - Nun stehen wir an Position k>2 und wissen r, l und alle Z_j, j<k

Z-Algorithmus 1

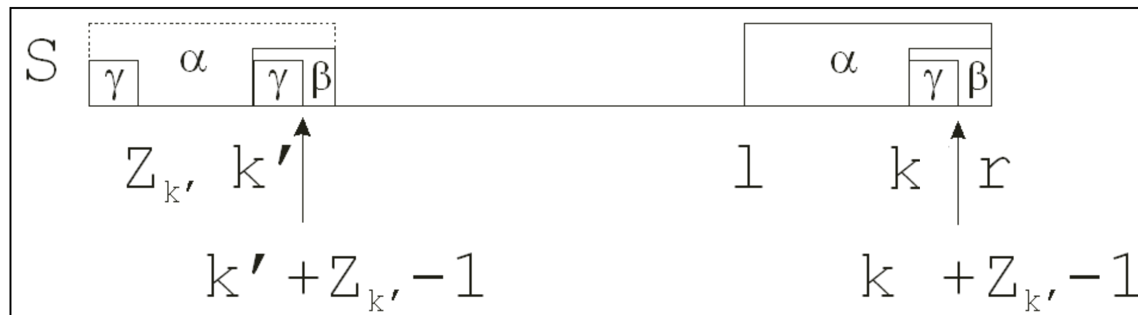
- Annahme 1: $k > r$
  - D.h., dass es keine Z-Box gibt, die k enthält
  - Dann tappen wir weiter im Dunkeln
    - Berechne Z_k durch Matching
    - Wenn $Z_k>0$, setze $r=r_k$ und $l=l_k$
- Annahme 2: $k <= r$
  - Die Situation:



  - Also
    - Z-Box Z_l ist Präfix von S
    - Substring β=S[k..r] kommt auch an Pos $k`=k-l+1$ von S vor
    - Was wissen wir über diesen Substring? Natürlich: $Z_{k`}$
    - D.h., dass S[k..] ist Präfix von S mit mindestens Länge $\min(Z_{k`}, |\beta|)$

Z-Algorithmus 2

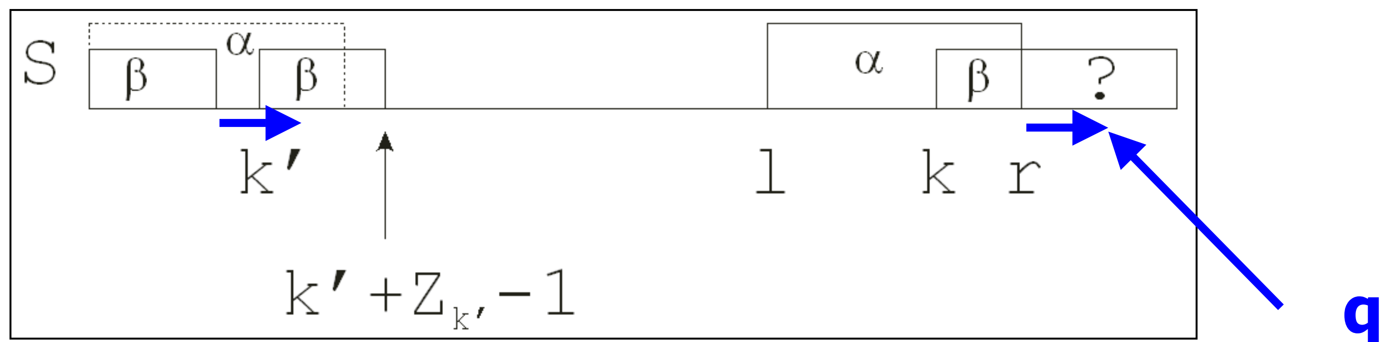
- Fallunterscheidung
  - $Z_{k'} < |\beta|$: Dann ist das Zeichen an $k'+Z_{k'}$ ein Mismatch bei der Präfixverlängerung. Dann ist das Zeichen $S(k+Z_{k'})$ der gleiche Mismatch. Also $Z_k=Z_{k'}$; r und l unverändert



Z-Algorithmus 3

- $Z_{k'} \geq |\beta|$: Dann ist β ein Präfix von S
  - ... dass sich vielleicht sogar verlängern lässt
  - ... denn i.A. ist $S(k'+Z_{k'}+1) \neq S(r+1) = S(k+|\beta|+1)$
  - ... und $S(|\beta|+1)$ ist noch nicht mit $S(r+1)$ gematched worden

Matche Substring S[r+1.. ?] mit S[|β|+1.. ?]
Sei der erste Mismatch an Position q
Dann: $Z_k=q-k$; $l=k$; $r=q-1$



Linearer Stringmatching Algorithmus

- Z-Boxen lassen sich in O(|S|) berechnen
- Verwendung der Z-Boxen für String Matching

```
S := P||'$'||T;                    // ($ ∉ Σ)
Berechne Z-Boxen von S;
for i = |P|+2 to |S|
        if (Z_i(S)=|P|) then
                print Z_i(S);  // P in T at position i
        end if;
end if;
```

- Komplexität
  - Schleife wird m-Mal durchlaufen => O(m)

Inhalt dieser Vorlesung

- Suche in Sequenzdatenbanken:
  Festes Template, wechselnde Suchsequenzen
- Suffixbäume
- Verwendung und Konstruktion
- Suffixarrays

Problemstellung

- Bisherige Algorithmen
  - Gegeben ein Template T (m) und ein Pattern P (n)
  - Suche alle Vorkommen von P in T
  - Dazu: Preprocessing von P in O(n), dann Suche in O(m)
- Jetzt betrachtetes Szenario
  - Gegeben sei eine Datenbank von Sequenzen (T)
  - Benutzer weltweit schicken kontinuierlich sich ändernde Sequenzstücke (P)
- Also: T (und nicht P) vorverarbeiten
- Für die Suche wird O(n) angestrebt
- Lösung: Suffixbäume

Weiteres Vorgehen

- Definition Suffixbaum
- Beispiele
- Einige Anwendungen
- Ein erster Konstruktionsalgorithmus

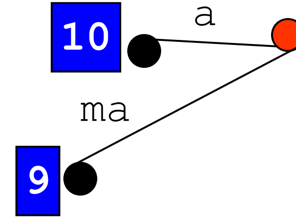
- Ab jetzt: Wir bauen einen Suffixbaum T für String S mit $|S|=m$

Suffixbäume

- Intuition: Kompakte Repräsentation aller Suffixe von S in einem Baum
- Definition: *Suffixbaum T für S ist ein Baum:*
  - *m Blättern beschriftet mit 1..,m*
  - *Jede Kante E ist mit einem Substring label(E)$\neq\emptyset$ von S beschriftet*
  - *Jeder interne Knoten K hat $\geq$2 Kinder. Die Label aller Kanten aus K beginnen mit unterschiedlichen Zeichen*
  - *Sei $\{K_1,K_2,...K_n\}$ ein Pfad von der Wurzel zu einem Blatt mit Beschriftung i. Dann ist die Konkatenation der Label des Pfades gleich S[i..m]*
    - Also das Suffix von S, das an Position i startet

Beispiel 1

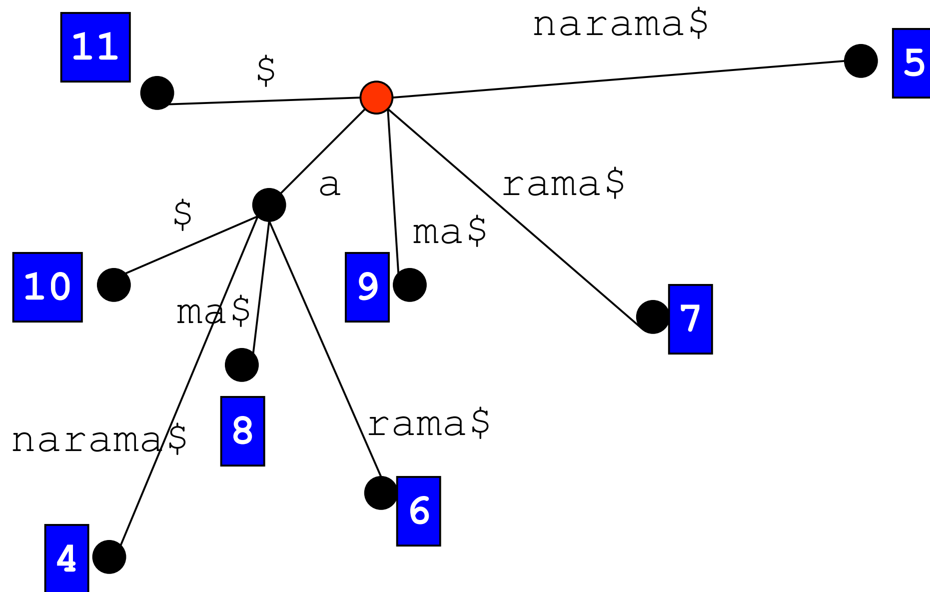
```
  1234567890
```
- S= BANANARAMA



- Problem: Wohin kommt „AMA“ ?
  - Verlängerung von „a“ verboten – 10 sonst kein Blatt
  - Neue Kante „ama“ verboten – zwei Pfade aus der Wurzel würden sonst mit gleichem Zeichen beginnen
  - ➢ Es gibt keinen Suffixbaum für BANANARAMA
  - ➢ Problem tritt auf, sobald ein Suffix Präfix eines anderen Suffix ist
- Trick: Wir betrachten „BANANARAMA$“
  - „$“ nicht Teil des Alphabets von S
  - Problemfall kann nicht mehr auftauchen, da $\$ \notin S$

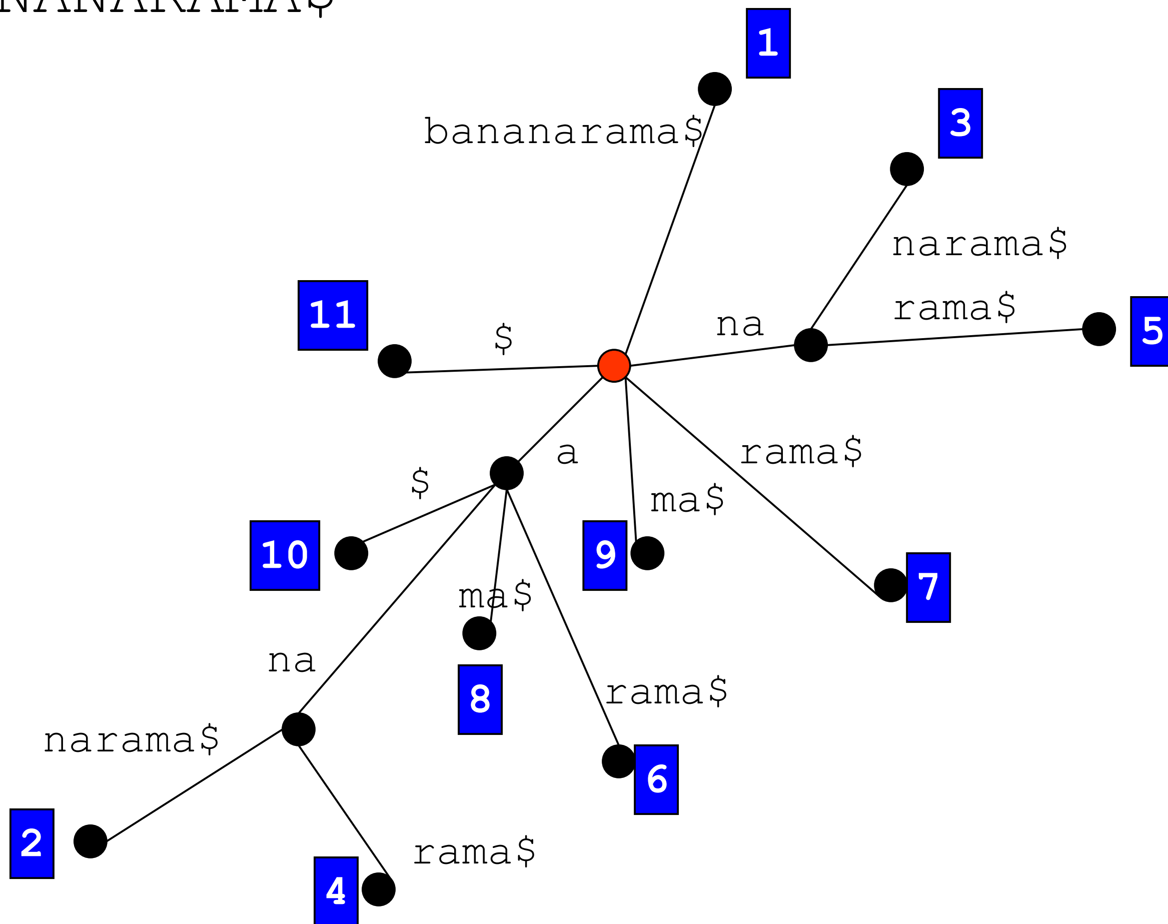
Beispiel 2

```
     12345678901
```
- S= BANANARAMA$



Beispiel 3

```
     12345678901
```
- S= BANANARAMA$



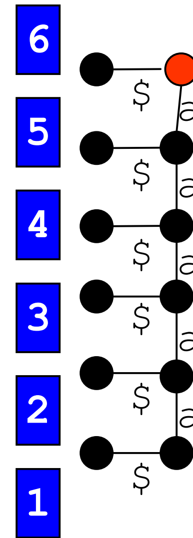
Humboldt Universität Informatik

Eigenschaften von Suffixbäumen

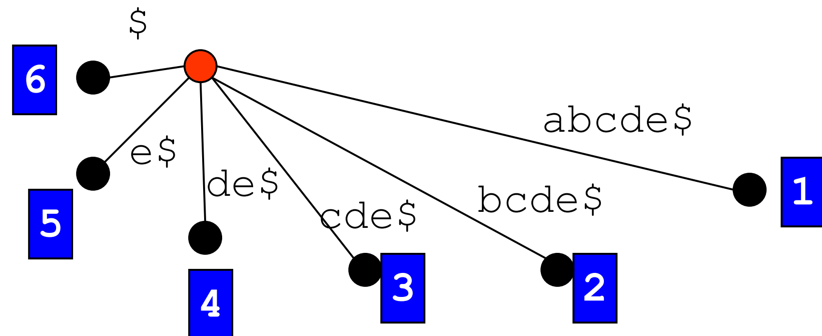
- Zu jedem String gibt es genau einen Suffixbaum
- Jeder Pfad von der Wurzel zu einem Blatt ist unterschiedlich
- Jede Verzweigung an einem inneren Knoten ist eindeutig bzgl. des nächsten Zeichens auf dem Pfad
- Gleiche Substrings können sehr wohl an mehreren Kanten stehen

Weitere Beispiele

- S= aaaaaa$



- S= abcde$



Suche mit Suffixbäumen

- Intuition: Jedes Vorkommen von P muss als Präfix eines Suffixes vorkommen
  - Und die haben wir alle auf Pfaden von der Wurzel aus
- Gegeben String S und ein Pattern P. Finde alle Vorkommen von P in S
  - Konstruiere den Suffixbaum T zu S
    - Das geht in $O(|S|)$, wie wir sehen werden
  - Matche P auf einen Pfad in S. Wenn das nicht geht, kommt P in S nicht vor. Sonst sei K der letzte Knoten des Pfades
    - Pfade sind eindeutig – das kostet nur $O(|P|)$
  - Alle unterhalb von K gelegenen Blätter in T sind Startpunkte von Vorkommen von P in S
    - Depth-First Durchlaufen

Beispiel: bananarama$

P = „na“

P = „ana“

Naive Konstruktion von Suffixbäumen

- Gegeben: String S. Gesuch: Suffixbaum T für S

- Bilde Baum T_0 mit Wurzelknoten und einer Kante mit Label „S$"zu einem Blatt mit Label 1
- Induktion: konstruiere T_{i+1} aus T_i wie folgt
  - Betrachte das Suffix S_{i+1} = S[i+1..]
  - Matche S_{i+1} in T_i so weit wie möglich
  - Wenn S_{i+1} auf einer Kante P mit Label L an Position j aufgebraucht ist, füge in P an Position j einen Knoten ein mit neuem Kind „$"
  - Wenn S_{i+1} auf einer Kante P mit Label L an Position j nicht mehr matched (der Mismatch in S_{i+1} sei j'), füge in P an Position j einen Knoten ein mit neuem Kind „S[j'..]$"

Beispiel

- „barbapapa“
- ...

Komplexität

- Komplexität
  - Jeder Induktionsschritt ist O(m)
  - Es gibt m-1 Induktionsschritte
  - Zusammen: $O(m^2)$
- Aber: O(m) Algorithmus von Ukkonen bekannt

Anwendungen

- Längster gemeinsamer Substring zweier Strings
- Längstes Palindrom

Längster gemeinsamer Substring

- Gegeben zwei Strings S_1 und S_2
- Gesucht: Längster gemeinsamer Substring s
- Vorschläge ?
- Lösung
  - Konstruiere Suffixbaum T für $S_1\$S_2\%$
  - Durchlaufe T depth-first und markiere alle internen Knoten mit 1, wenn im Baum darunter ein Blatt aus S1 kommt; markiere Knoten mit 2 für S2
  - Suche breadth-first den tiefsten Knoten mit Beschriftung 1 und 2
- Komplexität: $O(|S_1| + |S_2|)$

Beispiel

- S_1=main, S_2=kai
- ...

Längstes Palindrom

- Gegeben String S. Finde den längsten Substring s, der sowohl vorwärts als auch rückwärts in S vorkommt
- Ideen ?
- Lösung
  - Konstruiere Suffixbaum T für S$reverse(S)%
  - Suche längsten gemeinsamen Substring

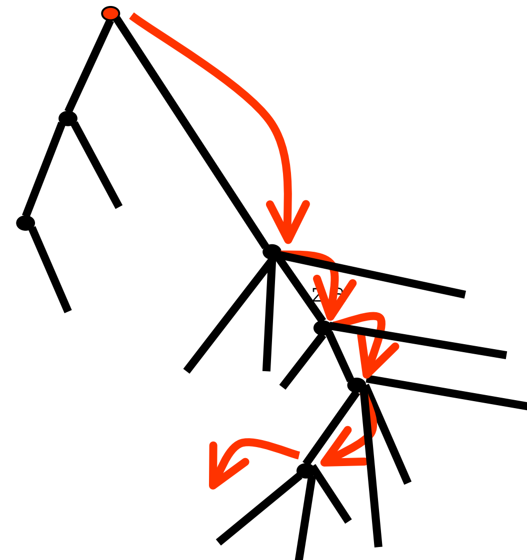
Zusammenfassung

- Bisherige Algorithmen sind exzellent für veränderliche Texte (Textverarbeitung etc.)
- Für Datenbanksuchen (Feste T, dauernd andere P) sind Suffixbäume besser
- Naive Konstruktion ist teuer
  - Aber lineare Algorithmen sind bekannt

- Ein Problem bleibt: Hoher Speicherverbrauch
- Deshalb: Suffixarrays

Suche in Suffixbäumen

- Matche Suchstring bis Erfolg oder Mismatch
  - Jeder Substring von S ist Präfix eines Suffix
  - An jedem Knoten Entscheidung unter allen ausgehenden Kanten basierend auf erstem Zeichen des Labels
- Wie trifft man diese Entscheidung?
  - Array in der Größe des Alphabets Σ; Zelle x mit Pointer auf Kante k, wenn label(k)[1]=x
    - Konstante Zeit pro Knoten
  - Verkettete Liste
    - Lineare Zeit pro Knoten
    - Mit wachsendem Array, Sortierung und binärer Suche: $\log(|\Sigma|)$



Trade-Off Zeit / Platzbedarf

- Arrayspeicherung
  - Das erfordert ein Array pro Knoten in der Größe des Alphabets Σ
  - Gesamtspeicherbedarf damit mindestens $m*|\Sigma|$
  - Enormer Speicherverbrauch bei allem außer DNA (z.B: Proteine, Restriktionmaps)
  - Aber nur dann hält die $O(n)$ Suchkomplexität
- Alternativen
  - Geringer Speicherverbrauch, aber $O(n*\log|\Sigma|)$ Suche
  - Hoher Speicherverbrauch, aber $O(n)$ Suche
- Oftmals besser: Suffixarrays
  - Viel geringerer Speicherverbrauch, $O(n+\log(m))$ Suche

Suffixarrays

- Definition
  - *Ein Suffixarray a für String S ist ein Integerarray der Länge |S|, in dem a[i] den Index des i-ten Suffix von S, sortiert nach lexikographischer Ordnung, enthält*
- Beispiel

```
12345678901
mississippi
```

```
mississippi    i             a[1]=11
ississippi     ippi          a[2]=8
ssissippi      issippi       a[3]=5
sissippi       ississippi    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
```

Konstruktion von Suffixarrays

- Behauptung: Suche nach P in a benötigt nur O(n+log(m))
  - Später
- Zunächst: Konstruktion eines Suffixarrays
  - In linearer Zeit aus Suffixbaum
  - Erinnerung: Im Suffixbaum gibt es ein Blatt pro Suffix
  - Wir suchen die Blätter – und zwar gleich in der richtigen Reihenfolge – und füllen dabei das Array von vorne nach hinten
    - Irgendwie suchen und sortieren wäre schlechter: O(m*log(m))

Trick zur linearen Konstruktion

- Ablauf in „lexikographischer“ Depth-First Suche
  - Wir laufen den Suffixbaum Depth-First ab
  - An jedem Knoten wählen wir die Kinder in der lexikographischen Reihenfolge der Kantenlabel
  - Mitzählen
    - Erstes Blatt mit Beschriftung $i_1 \Rightarrow a[1] = i_1$
    - Zweites Blatt mit Beschriftung $i_2 \Rightarrow a[2] = i_2$
  - [„$“ gilt dabei als kleiner als alle anderen Zeichen]
- Komplexität: $O(m)$
  - Depth-First ist abhängig von Anzahl Knoten
  - Wenn die Kinder als sortierte verkettete Liste oder als Array gespeichert sind ist jede Entscheidung in konstanter Zeit möglich

Suche mit Suffixarrays

- Ideen?
- Suche alle Vorkommen von P=„ssi“
  - Erinnerung: Jeder Substring ist Präfix (mindestens) eines Suffix
  - P liegt also am Anfang eines Suffix (wenn P in S)
  - Suffixe liegen alle sortiert vor
  - ➢ Also: Binäre Suche im Suffixarray

```
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
```

Suchalgorithmus

```
l:=1; r:= m;
Let f(i) abbreviate S[a[i]..a[i]+n]
while l<r do
    z = l+(l-r)/2;                         // Middle of interval
    if (f(z) > P) then r = z;              // Go left
    else if f(z) < P then l = z;           // Go right
    else                                   // Found one occurence
        z‘:=z;
        while (f(z‘)=P & z‘>0) do          // Search „smaller“ occurences
            report a[z‘];
            z‘ := z‘-1;
        end while;
        z‘:=z+1;
        while (f(z‘)=P & z‘<=m) do         // Search „bigger“ occurences
            report a[z‘];
            z‘ := z‘+1;
        end while;
        break;
    end if;
end while;
```

Beispiel

- Suche alle Vorkommen von P=„ssi“

```
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
```

Beschleunigung

- Durch die binäre Suche sind wir bei $O(n*\log(m))$
  - Sehr pessimistisch – jeder Vergleich müsste n-1 Matches haben, dann 1 Mismatch
- Verbesserung ist möglich
  - Man muss nicht immer 1..n Zeichen vergleichen
  - Während Suche merken: p_l (p_r) ist die Länge des matchenden Präfix von P und f(z) an der linken (rechten) Grenze des Suchintervalls
  - Vergleiche müssen immer erst ab Position $p=\min(p_r, p_l)$ durchgeführt werden – die Zeichen 1..p sind im Intervall identisch
  - Heuristik - keine veränderte Worst-Case Komplexität, aber gute Ergebnisse im Average Case
- Algorithmen mit garantierter $O(n + \log(m))$ bekannt (Gusfield, p. 152-154)
- Suffixarrays kann man direkt bauen, Umweg über Suffixbaum ist nicht zwingend