Bioinformatik

2 Probleme mit Suffixbäumen

2 Lösungen

Ulf Leser

Wissensmanagement in der Bioinformatik

Überblick

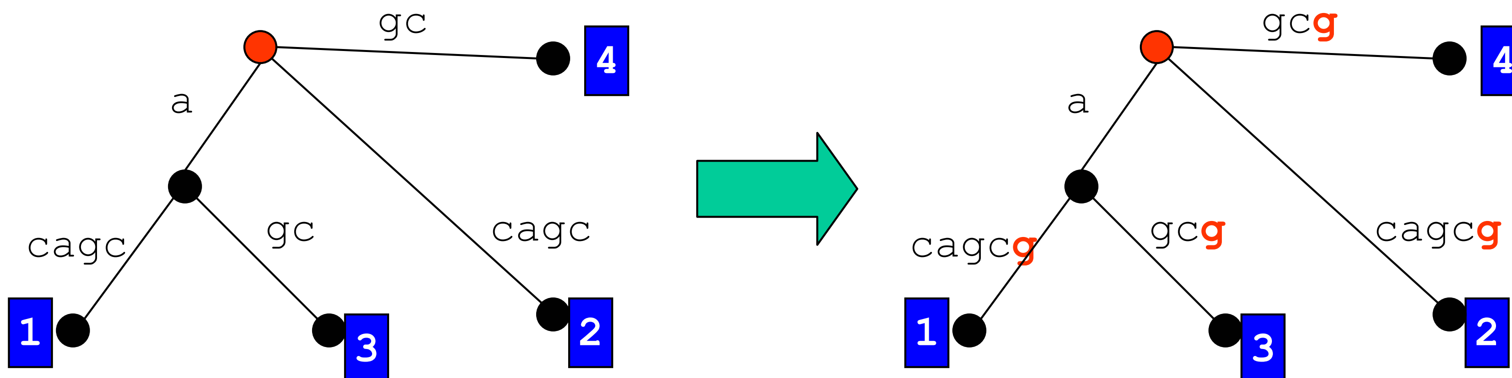
- Voraussetzungen
- High-Level: Phasen und Extensionen
  - Führt leider zu $O(m^3)$
- Verbesserungen
  - Suffix-Links
  - Skip/Count Trick
  - Noch zwei Tricks
- Alles zusammen: $O(m)$

Grundaufbau Ukkonen‘s Algorithmus

- Induktive Konstruktion impliziter Suffixbäume für jedes Präfix von T
  - Wir konstruieren alle T_i, d.h., implizite Suffixbäume für S[1..i]
  - Startpunkt T_1: Wurzel und ein Knoten mit Kantenlabel S[1]
  - Phasen (Induktionsschritte): Konstruktion von T_{i+1} aus T_i
  - Abschluss: Transformation von T_m in den Suffixbaum T
- Jede der m-1 Phasen besteht aus Extensionsschritten
  - Phase i hat i Extensionsschritte
  - Jeder Schritt verlängert ein Suffix von S[1..i] um das Zeichen S[i+1]
  - Der letzte Schritt jeder Phase verlängert das leere Suffix – einfügen von S[i+1]
  - Reihenfolge der Schritte: von links nach rechts (S[1..i], S[2..i], ...)
- Wie wird verlängert?
  - Drei Extensionsregeln

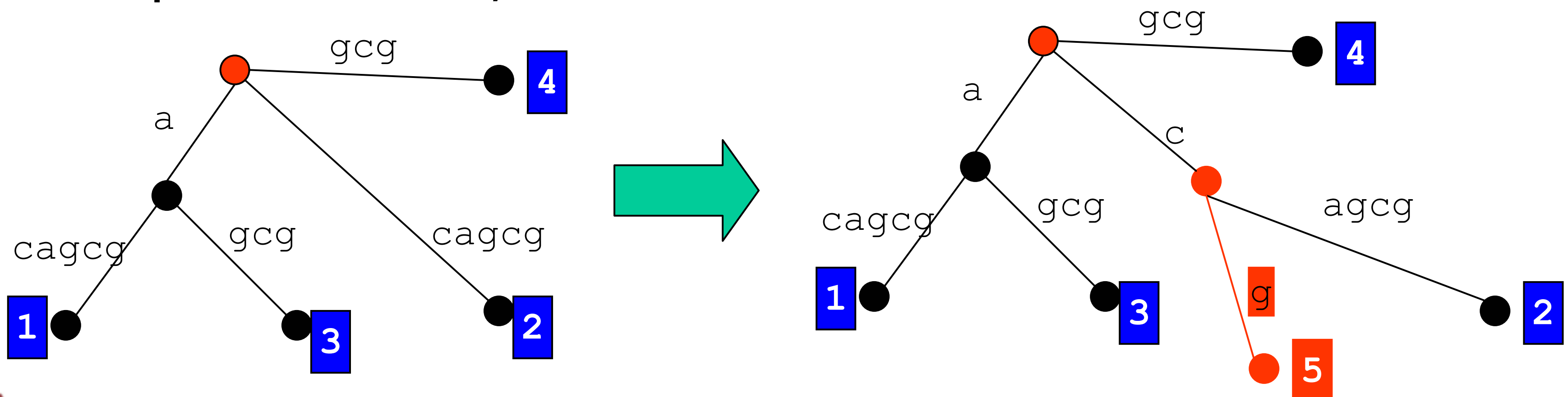
Extensionsregel 1

- Matche b in T_{i+1}. Das geht bis ...
  - Regel 1: b endet in einem Blatt
  - ➢ Erweitere das Label der letzten Kante um S[i+1]
- Beispiel (wir hängen „g" an Suffixe von „acagc")
  - Erweiterung von „acagc", „cagc", „agc", „gc"
  - [es bleiben Schritt 6 „" und Schritt 5 „c"]



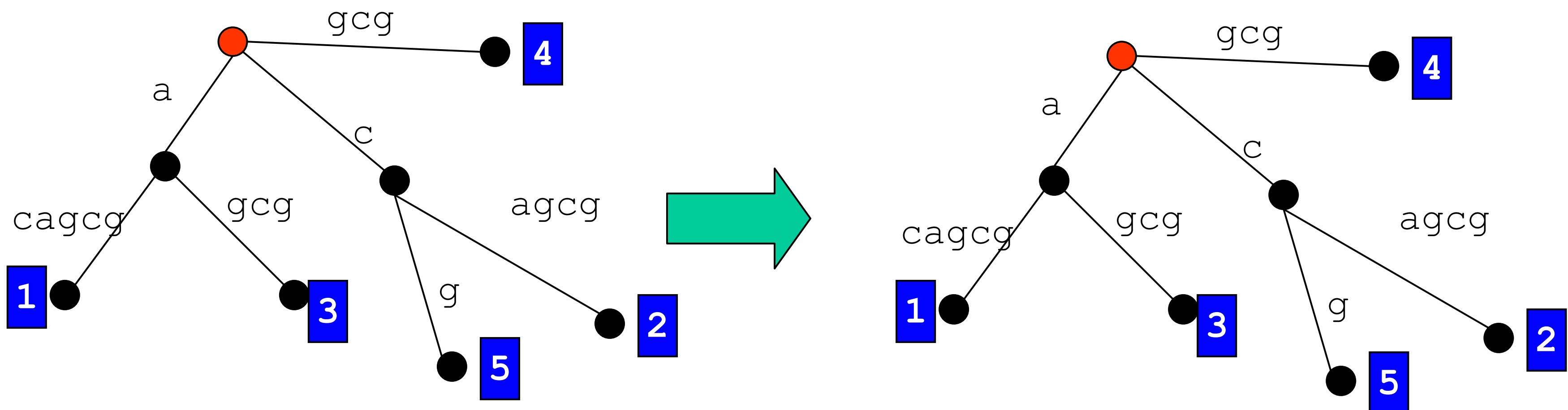
Extensionsregel 2

- Matche b in T_{i+1}. Das geht bis ...
  - Regel 2: b endet an einem inneren Knoten oder in einer Kante, und kein weiterer Pfad beginnt mit S[i+1]
    - Innerer Knoten: Neues Blatt unterhalb dieses Knotens mit Kantenlabel S[i+1]; markiere Blatt mit „j"
    - In einer Kante: Neuer innerer Knoten, der diese Kante teilt; neues Blatt wie oben
- Beispiel: Schritt 5, „c"



Extensionsregel 3

- Matche b in T_{i+1}. Das geht bis ...
  - Regel 3: b endet an einem inneren Knoten oder in einer Kante, und einer der weiteren Pfade beginnt mit S[i+1]
    - ➢ Tue gar nichts
- Beispiel: Schritt 6, „“



Algorithmus und Komplexität

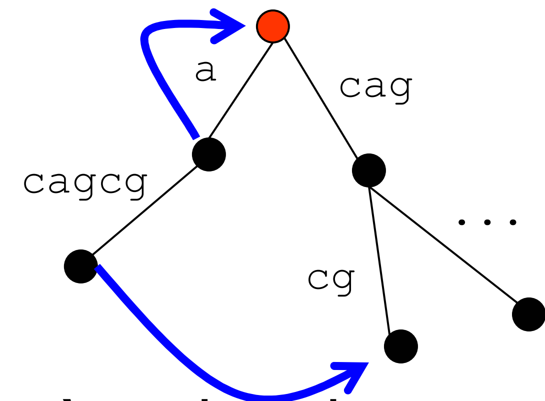
```
construct T1;
for i=1 to m-1                          // m-1 phases
    Ti+1 = Ti;
    for j = 1 to i+1                    // i+1 extensions, left-right
        match S[j..i] in Ti+1;
        extend Ti+1 with S[i+1];        // Using one of the 3 rules
    end for;
end for;
```

- Die zwei Schleifen sind $O(m^2)$
- Finden der Suffixe b in T_{i+1} ist $O(m)$
- Extension ist konstant
- Zusammen: $O(m^3)$

➢ Da kann noch nicht alles gewesen sein ...

Suffix-Links formal

- Definition
  - *Sei k ein innerer Knoten des impliziten Suffixbaums T‛ für S*
  - *Sei label(k)=„xa", wobei |x|=1, |a| beliebig (auch 0)*
  - *Sei k‛ ein innerer Knoten von T‛ mit label(k‛)=a*
  - *Der Pointer (k,k‛) heißt Suffix-Link*



- Spezialfall für |a|=0
  - Der Suffix-Link geht dann zur Wurzel
- Wir zeigen, dass jeder innere Knoten in T‛ nach jeder Phase von Ukkonen Algorithmus einen Suffix-Link hat
- Diese Links werden wir dann in späteren Phasen entlang springen

Verwendung der Suffix-Links

- Während einer Phase sucht man nacheinander die Enden von S[1..i], S[2..i], S[3..i] etc.
  - Sprich: xyz..., yz..., z... – genau das Suffix-Link Szenario
- Wenn wir in Schritt j das Ende von S[j..i] gefunden haben und zu Schritt j+1 übergehen
  - Suche den inneren Knoten k über dem Ende von S[j..i]
  - Wenn k die Wurzel ist
    - Matche wie bei naivem Algorithmus
  - Sonst ist k ein innerer Knoten
    - Folge dem Suffix-Link von k zu Knoten k�much
    - Das Ende von S[j+1..i] muss unter k�much liegen (aber wo?)
    - Das Präfix von S[j+1..i] oberhalb von k�much müssen wir nicht mehr beachten, sondern nur das Suffix unterhalb von k�much
- Anfang in jeder Phase
  - Wir merken uns immer einen Pointer auf Blatt 1
  - Mit dem fängt man immer an – längstes Suffix

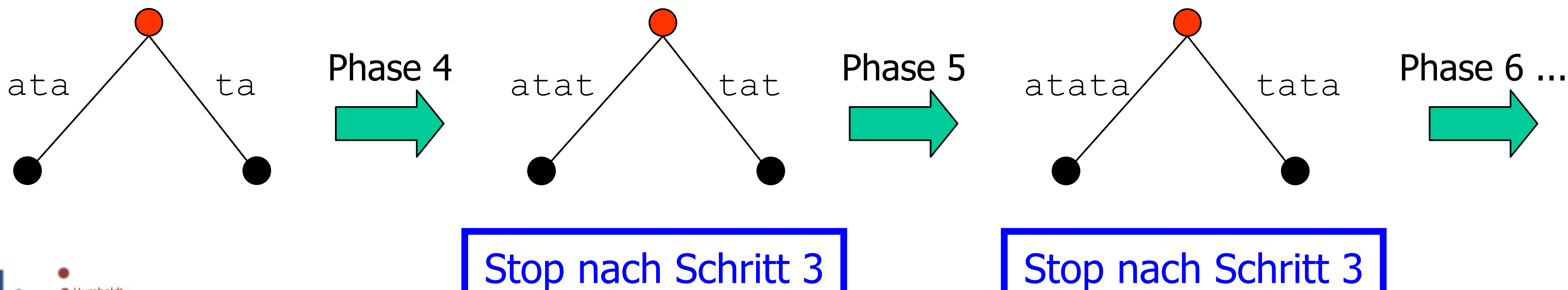
Nutzen bisher?

- Noch keiner ...
  - Unterhalb von k‵ müssen wir wieder Zeichen für Zeichen matchen
  - Das ist O(m) pro Extensionsschritt
- Noch ein Trick: Skip/Count
  - Wir wissen etwas über die Länge von S[j+1..i]
  - Wir können uns auch die Länge der Kantenlabel merken
    - Konstante Zeit zur Aktualisierung während des Aufbaus
  - Wir kennen damit die Länge des Präfix oberhalb von k‵
  - Wir können damit auch die Länge des Suffix unterhalb von k‵ berechnen
  - Damit können wir von Knoten zu Knoten hüpfen ...

Extensionsregeln – auf den zweiten Blick

- Beobachtung: Was passiert, wenn Regel 3 das erste Mal in einer Phase greift?
  - Wir verlängern ein Suffix S[j..i] um S[i+1]
  - Regel 3: Also gibt es das Suffix S[j..i+1] schon (kam schon in einer früheren Phase vor)
  - Dann gibt es auch S[j+1..i+1], S[j+2..i+1], ...
  - ➢ Wir können die Phase beenden – in dieser Phase wird nur noch Regel 3 greifen, und die ändert nichts am Baum

- Beispiel: „atatatatatc", Phase ...



Extensionsregel, Abkürzung 2

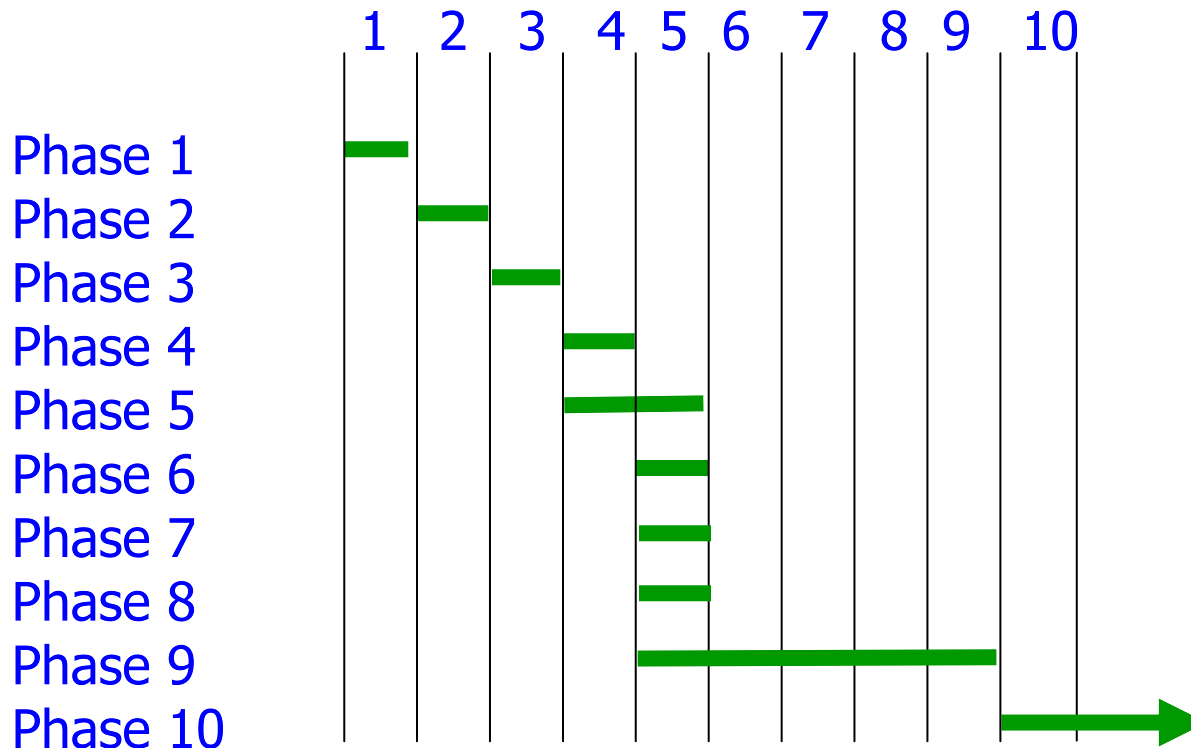
- Beobachtung
  - Es gibt keine Regel zur Umwandlung von Blättern
  - ➢ Blätter bleiben immer Blätter
- Was passiert in den Schritten
  - Regel 1: Verlängerung des Labels einer Blattkante
  - Regel 2: Neues Blatt oder: neuer Knoten und neues Blatt
  - Regel 3: Nichts, Abbruch der Phase
- Jede Phase läuft also ab ala 1,2,2,2,1,2,1,3
  - Erst einige Anwendungen von 1 oder 2, dann einmal 3
  - Sei j` die letzte explizite Extension in Phase i
- Bei jeder Anwendung von 1 oder 2 wird das Label einer Blattkante verlängert oder ein Blatt+Kante geschaffen
  - Alle Schritte bis j` sind nach Phase i durch Blätter repräsentiert
  - ➢ In Phase i+1 verlängern die Schritte 1...j` nur Blätter

Extensionsregel, Abkürzung 2

- Diese Schritte können wir uns schenken
  - Blattkanten erhalten statt (p,q) die Beschriftung (p,E)
  - E steht für „Bis zum Ende“
    - Am Ende wird jede Blattkante bis zum Ende von S gehen (denn Blättern repräsentieren schließlich Suffixe von S)
- Damit: Eine Phase im einzelnen
  - Überspringe alle Schritte bis j‵ der letzten Phase
  - Führe explizite Extensionen aus, entweder bis i+1 oder bis zur ersten Anwendung von Regel 3
    - Hier werden neue Blätter / innere Knoten geschaffen
    - Das kann auch eine Blattkante unterbrechen – p anpassen
  - Neues j‵ merken und nächste Phase starten

Komplexität

- Welche Schritte haben wir ausgeführt?



- Schritte markieren einen Pfad von links oben nach rechts unten
- Das können zusammen höchstens 2*m Schritte sein

Inhalt dieser Vorlesung

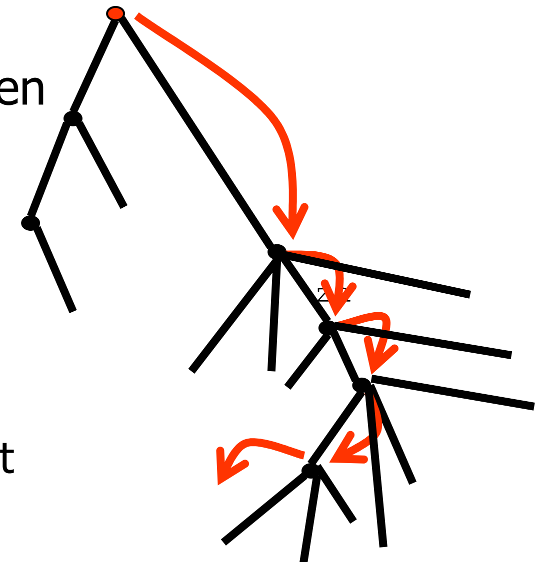
- Implementierung von Suffixbäumen: Zwei Probleme

- Problem 1: Speicherbedarf ist groß
- Lösung: Suffixarrays

- Problem 2: Schlechtes Laufzeitverhalten auf Sekundärspeichern
- Lösung: Inkrementeller Konstruktionsalgorithmus

Suche in Suffixbäumen

- Matche Suchstring bis Erfolg oder Mismatch
  - An jedem Knoten Entscheidung unter allen ausgehenden Kanten basierend auf erstem Zeichen des Kantenlabels
  - Alle Kantenlabel beginnen mit verschiedenen Zeichen
- 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
    - Pointer auf Kinderkanten sind untereinander verkettet
    - Lineare (in |Σ|) Zeit pro Knoten
  - Wachsendes, sortiertes Array
    - Pointer auf Kinderkanten sind alphabetisch sortiert und direkter Zugriff auf Pointer
    - Mit binärer Suche: log(|Σ|)



Problem 1: Trade-Off Zeit / Platzbedarf

- Arrayspeicherung
  - Erfordert pro Knoten ein Array in der Größe des Alphabets Σ
  - Gesamtspeicherbedarf damit O(m*|Σ|)
  - Enormer Speicherverbrauch bei allem außer DNA
    - z.B: Proteine, Restriktionmaps (später)
  - Also: Hoher Speicherverbrauch, aber O(n) Suche
- Alternativen
  - Sortiertes, wachsendes Array
  - Geringer Average-Case Speicherverbrauch
  - Aber O(n*log(|Σ|)) Suche
- Oftmals besser: Suffixarrays
  - Geringer Speicherverbrauch, O(n+log(m)) Suche

Motivation

- Performance (Zeit / Speicherverbrauch) von Suffixarrays unabhängig von der Größe des Alphabets
  - Warum ist das wichtig?
- DNA/RNA: $|\Sigma|=4$
  - Platzbedarf der Suffixbäume vertretbar
- Proteine: $|\Sigma|=20$
  - Anwendungsabhängige Entscheidung
- [Chinesisch: ~ 2000 Zeichen]
- Aber es gibt auch ganz andere „Alphabete"
  - Suffixbäume haben so schnelle Algorithmen, dass es sich oftmals lohnt, ein Problem als String zu modellieren
  - Beispiel: Genomkarten

Vergleich von Genomkarten

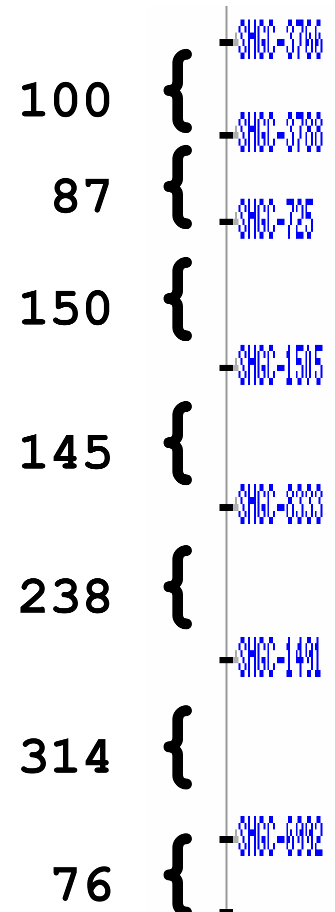
- Situation
  - Chromosom X ist sequenziert
  - Schnittstellen des Restriktionsenzyms Y auf X bekannt
  - Labor unternimmt „Position Cloning" auf der Suche nach einem Gen
  - Ergebnis ist ein Clone Z, der das Gene enthält
  - Frage: Liegt dieser Clone auf Chromosome X?
- Sehr billige Möglichkeit
  - Schnittstellen von Y auf Z bestimmen
  - Mit den Schnittstellen auf X vergleichen
  - Das Muster der Stücklängen von Z muss irgendwo in X liegen

Modellierung als Stringmatching

- Restriktionskarten als Strings
  - Wichtig sind die Abstände der Schnittpunkte (also die Länge der Fragmente)
- Chromosome X
  - X‘=„100,87,150,145,238,314,76,...“
- Clone Z
  - Z‘=„145,238,314,76,...“

➢ Z liegt auf X wenn Z‘ $\subseteq$ X‘

- Wie groß ist der Alphabet?
  - Sehr groß ...
  - Abhängig vom Enzym (Häufigkeit der Bindungsstelle)



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, 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))
  - Wie? 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
    - Suffixe in beliebiger Reihenfolge aufzählen und dann sortieren wäre schlechter: O(m*log(m))
    - Aber Suffixbäume sind ja in gewisser Weise „sortiert“

Trick zur linearen Konstruktion

- Ablauf in „lexikographischer" Depth-First Ordnung
  - 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 größer 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 konstant

Suche mit Suffixarrays

- Ideen?
- Suche alle Vorkommen von P=„ssi" in „mississippi"
  - Erinnerung: Jeder Substring ist Präfix (mindestens) eines Suffix
  - P liegt also am Anfang eines Suffix (wenn P in S)
  - Suffixe liegen alle sortiert vor
  - ➢ 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; n=|P|;
deffunc f(i):=S[a[i]..a[i]+n];
while l<r do
    z = floor(l+(r-l+1)/2);           // Middle of interval
    if (f(z) > P) then r = z;         // Go up
    else if f(z) < P then l = z;      // Go down
    else                              // Found one occurrence
        z`:=z;
        while (f(z`)=P & z`>0) do     // Search smaller occurrences
            report a[z`];
            z` := z`-1;
        end while;
        z`:=z+1;
        while (f(z`)=P & z`<=m) do    // Search bigger occurrences
            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
```

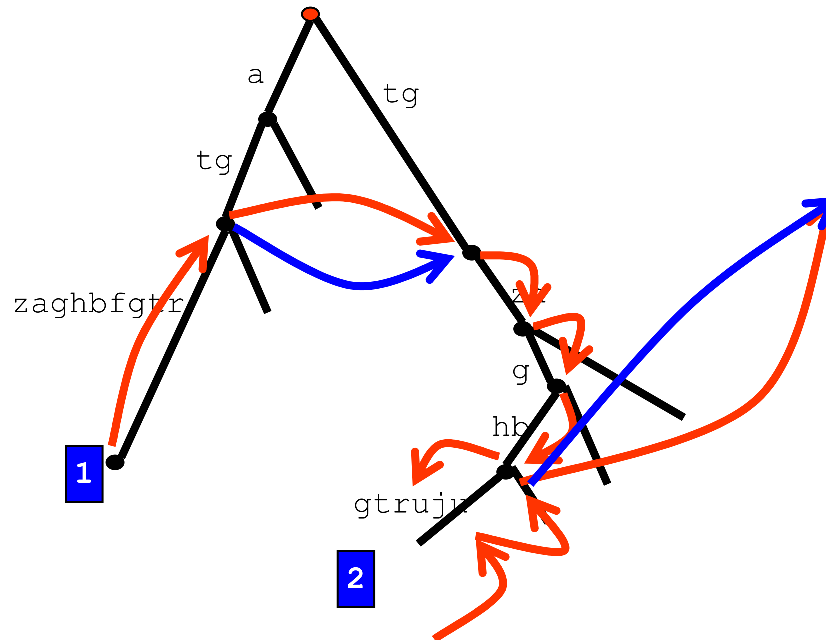
Beschleunigung

- Durch die binäre Suche sind wir bei O(log(m)*n)
  - 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 aktuellen Suchintervall bei allen Suffixen 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)

Was ist gewonnen?

- Muss man zuerst den (großen) Suffixbaum bauen, um dann das (kleinere) Suffixarray zu bauen
  - Stimmt nicht ... direkte Konstruktionen bekannt
- Nicht alles, was mit Suffixbäumen geht, geht mit Suffixarrays
- Suffixbäume bleiben wichtig
  - Und die haben noch Problem 2: Konstruktion mit Sekundärspeicher kaum möglich

Konstruktion von Suffixbäumen mit Ukkonen's Algorithmus



- Baum wird beim Konstruieren auf zwei Arten traversiert
  - Verfolgen von Suffix-Links – Sprünge in andere Teilbäume
  - Knotenhüpfen – Abstieg in einen Teilbaum
- Kein Problem beim Aufbau im Hauptspeicher, aber ...

Problem 2: Suffixbäume auf Sekundärspeichern

- Blöcke müssen auf / von Disk geschrieben / gelesen werden
- Keine lineare Anordnung der Blöcke möglich, da zwei Ordnungen vorhanden
  - Suffix Links und Pfade ab Wurzel zu Blatt
- Random Access Zugriff auf Datenblöcke – teure IO
- Sehr schlechtes Laufzeitverhalten

➢ Suffixbäume gelten (galten?) als unbenutzbar für große Strings

- Lösung: Anderer Konstruktionsalgorithmus

Veränderter Konstruktionsalgorithmus

- Hunt, E., Atkinson, M. and Irving, R. W. "A Database Index to Large Biological Sequences". VLDB 2001.
- Beobachtung
  - Platz-effizienteste Implementierung von Suffixbäumen für DNA brauchen ca. 13 Byte/Base [Kurz et al. 1999]
    - Humanes Genom mit 3GB: ~ 13*3= 39 GB
  - Suffixbäume auf Sekundärspeicher schwierig zu konstruieren
    - Zwei Traversierungsmuster: Suffix-Links und Knoten-Hüpfen
    - Auch Blöcke mit inneren, hohen Knoten werden immer wieder verändert (Regel 2)
    - Sehr viel Paging, sehr ineffizientes IO-Verhalten
    - Algorithmen degradieren signifikant, wenn Hauptspeicher erschöpft ist

Idee

- Algorithmus
  - Keine Verwendung von Suffix-Links
  - Multi-Pass Algorithmus: Komplette Sequenz muss mehrmals gelesen werden
  - In jedem Lauf werden nur Suffixe mit einem bestimmten Präfix behandelt
  - Die liegen alle im gleichen Teilbaum
- Ziel: Komplette Teilbäume im Hauptspeicher bauen
  - Keine Sprünge zwischen Teilbäumen durch Suffix-Links zulassen
  - Keine Änderungen in anderen Teilbäumen durch gemeinsames Suffix-Präfix
  - Teilbaum wird von Disk gelesen, im Hauptspeicher gebaut, am Ende geschrieben und nie mehr geladen

Komplexität

- Suffixbäume kann mit in O(m) konstruieren
  - Benötigt Suffixlinks und andere Tricks
- Bei Konstruktion von Partitionen muss der naive Algorithmus ($O(m^2)$) verwendet werden
  - Denn Suffixe in Partition bilden zusammen keinen echten String
  - Bedingungen von Ukkonen treffen nicht zu
- Komplexität bei k Partitionen
  - k so wählen, dass Teilbaum im Hauptspeicher gebaut werden kann
  - Sequenz wird k-mal gelesen
  - Partition hat (m/k) Elemente
  - Zusammen: damit $O(k * (m/k)^2)$

Partitionen

- Zerlegung der Menge aller Suffixe in Partitionen
  - Partition ist charakterisiert durch gemeinsames Präfix
  - Unterschiedliches Präfix – unterschiedlicher Teilbaum
  - Präfixe entweder exakt bestimmen
    - Häufigkeiten von q-Grammen mit steigenden q zählen
    - q-Gramm X befindet sich an Position i -> Suffix mit Präfix q beginnt an Position i
    - Optimale q-Gramme so bestimmen, dass jeweilige Anzahl Suffixe exakt in den Hauptspeicher passt
    - Sehr teuer – viel Lesen des Strings, bevor überhaupt die Konstruktion beginnt
  - ... oder schätzen
    - DNA ist nahezu zufällig verteilt
    - Alle q-Gramme gleich häufig
    - Nur die Länge q abschätzen

Algorithmus

```
chose prefixes for partitioning;
construct suffixtree T for prefixes;  // of all partitions
for i in partitions do
    k := find_node(T, prefix(i));     // Find start of subtree
    for j = 0 to m do                 // Scan sequence on disk
       if S[j..j+|prefix(i)|]=prefix(i) then
           insert S[j..m] under k;    // Using naive construction
       end if;
    end for;
    write k to disk;                  // Never touched again
end for;
write T to disk;                      // Bookkeeping
```

- Kein paging
  - (Wenn Partitionen richtig gewählt)
- Subtrees liegen hintereinander auf der Platte
  - Gut für Suche – sequentielle Reads, kein Random Access
- Sehr gut parallelisierbar

Was gibt es noch?

- Wichtig für viele Erweiterungen
  - „Least common ancestor" (lca) für zwei Blätter ist in Suffixbäumen in konstanter Zeit lösbar (nach linearer Vorverarbeitung) [Gusfield, Kapitel 8]
- Damit: Suffixbäume und unscharfes Matching
  - Matching mit k Wildcards: O(k*m)
    - Trick: Ähnlich Keywordtrees mit Wildcards
  - K-Mismatch Problem: O(k*m)
    - Trick: Ebenso
  - Beide benutzen lca um in konstanter Zeit das längste Präfix zweier Suffixe zu finden
  - Details: [Gusfield, Kapitel 9.1, 9.3]

State of the Art

- Indexierung von DNA ist sehr schwierig
  - Keine Wörter, keine Trennzeichen
  - Suffixbäume so ziemlich der einzige Vorschlag bisher
- Beispiele aus [HAI01]
  - BLAST auf 4 Prozessoren: 99 Pattern mit Länge 400-6000 gegen 3 humane Chromosome (Zusammen 300 MB): 62 Stunden
  - NCBI/EBI haben große Serverfarmen – unmöglich für viele Firmen
- Viele Probleme benötigen die Flexibilität von BLAST nicht
  - Clustering = Suchen in sehr ähnlichen Sequenzen
  - Arbeiten mit Seeds oder k-Mismatches
  - Finden von Repeats
  - ...
- Suffixarrays/bäume zunehmend populär in Bioinformatik
  - Aber Sekundärspeicherproblem gilt als noch nicht gelöst

Zusammenfassung

- Suffixbäume sind extrem effiziente Hilfsmittel für viele Stringprobleme

- Kontinuierliche Weiterentwicklung
  - Suffixarrays: sehr speicherplatzeffizient
    - [Suffixarrays auf Sekundärspeichern?]
  - Partitionierte Konstruktion: verbessertes Sekundärspeicherverhalten
    - Einzelne Erweiterungen bekannt (Caching, Sortieren der Suffixe)
    - [Zusammenspiel mit unscharfem Matching?]

Aber ...

- Exakte Suchalgorithmen sind ja gut und schön ...
  - Z-Box, Boyer-Moore, Knuth-Morris-Prath, Aho-Corasick, Suffixbäume, Suffixarrays
- ... aber häufiger braucht man unscharfe Vergleiche
  - Mismatches, Deletions, Insertions
- Das kommt im nächsten Block