

Bioinformatik

Suffixbäume auf Sekundärspeicher



Ulf Leser

Wissensmanagement in der
Bioinformatik



Ukkonen Überblick

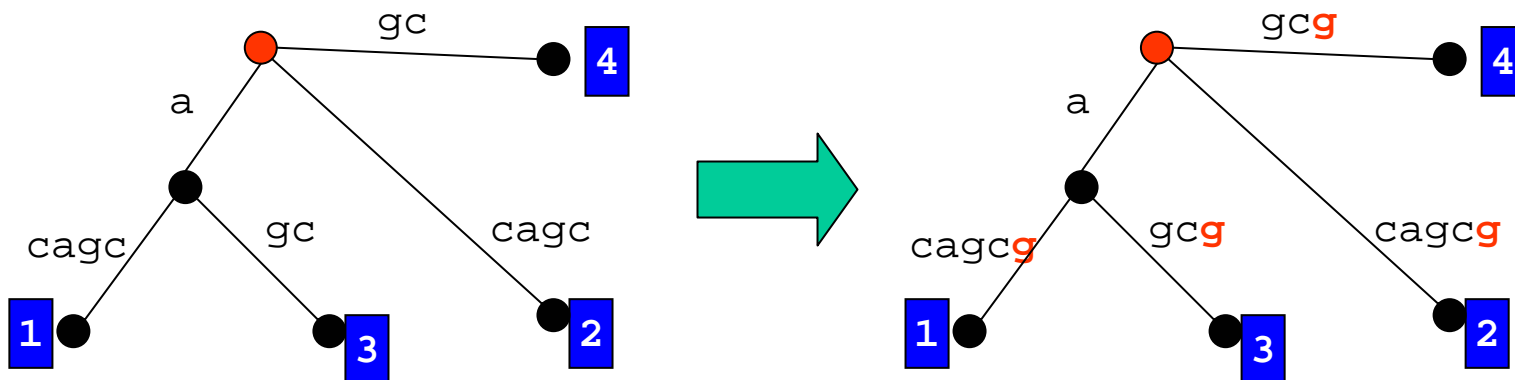
- High-Level: Phasen und Extensionen
 - Führt leider zu $O(m^3)$
- Verbesserungen
 - Suffix-Links
 - Skip/Count Trick
 - Noch zwei Tricks
- Alles zusammen: $O(m)$

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**

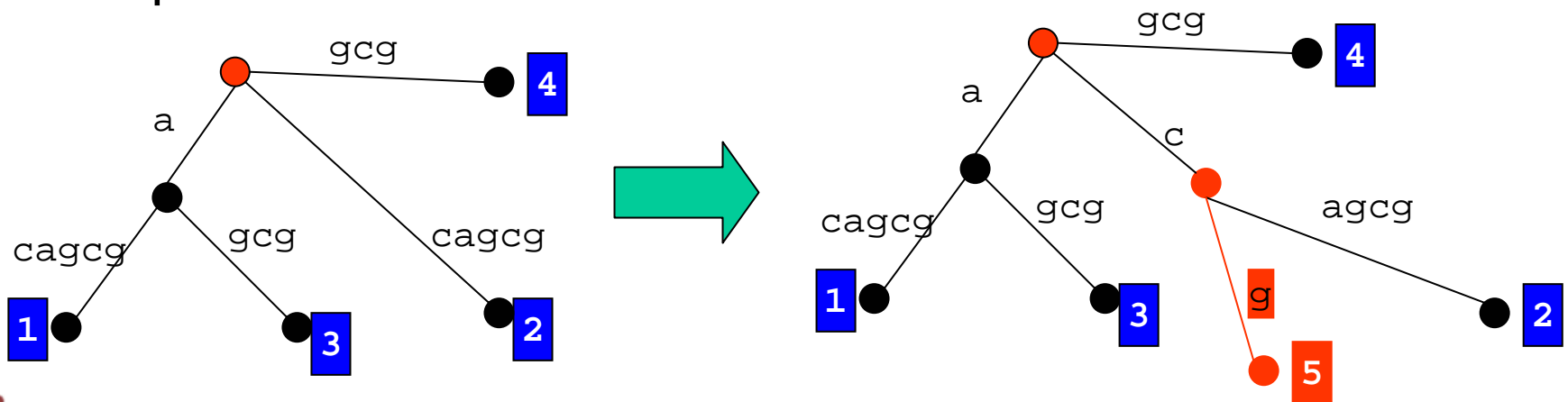
Extensionsregel 1

- 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“]



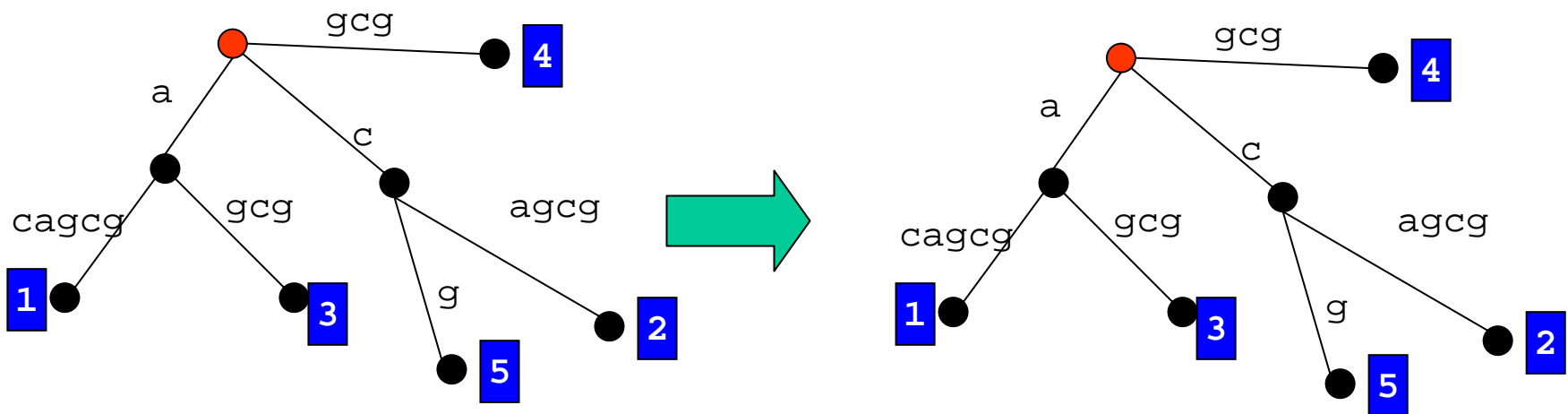
Extensionsregel 2

- 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 ...

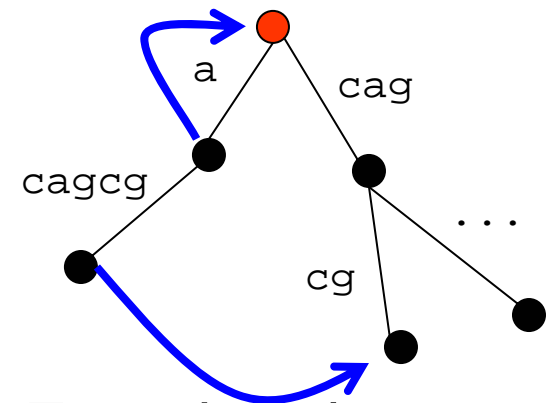
Suffix-Links formal

- Definition

- Sei k ein innerer Knoten des impliziten Suffixbaums T' für S
- Sei $\text{label}(k) = „xa“$, wobei $|x|=1$, $|a|$ beliebig (auch 0)
- Sei k' ein innerer Knoten von T' mit $\text{label}(k') = a$
- 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: $xyz\dots$, $yz\dots$, $z\dots$ – 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'
 - **Das Ende von $S[j+1..i]$ muss unter k' liegen (aber wo?)**
 - **Das Präfix von $S[j+1..i]$ oberhalb von k' müssen wir nicht mehr beachten, sondern nur das Suffix unterhalb von k'**
- Anfang in jeder Phase
 - Wir merken uns immer einen Pointer auf Blatt 1
 - Mit dem fängt man immer an – längstes Suffix

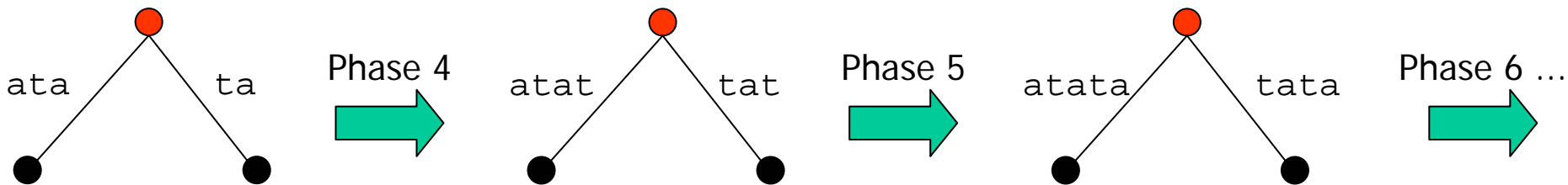


Nutzen bisher?

- 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 ...



Stop nach Schritt 3

Stop nach Schritt 3

Extensionsregel, Abkürzung 2

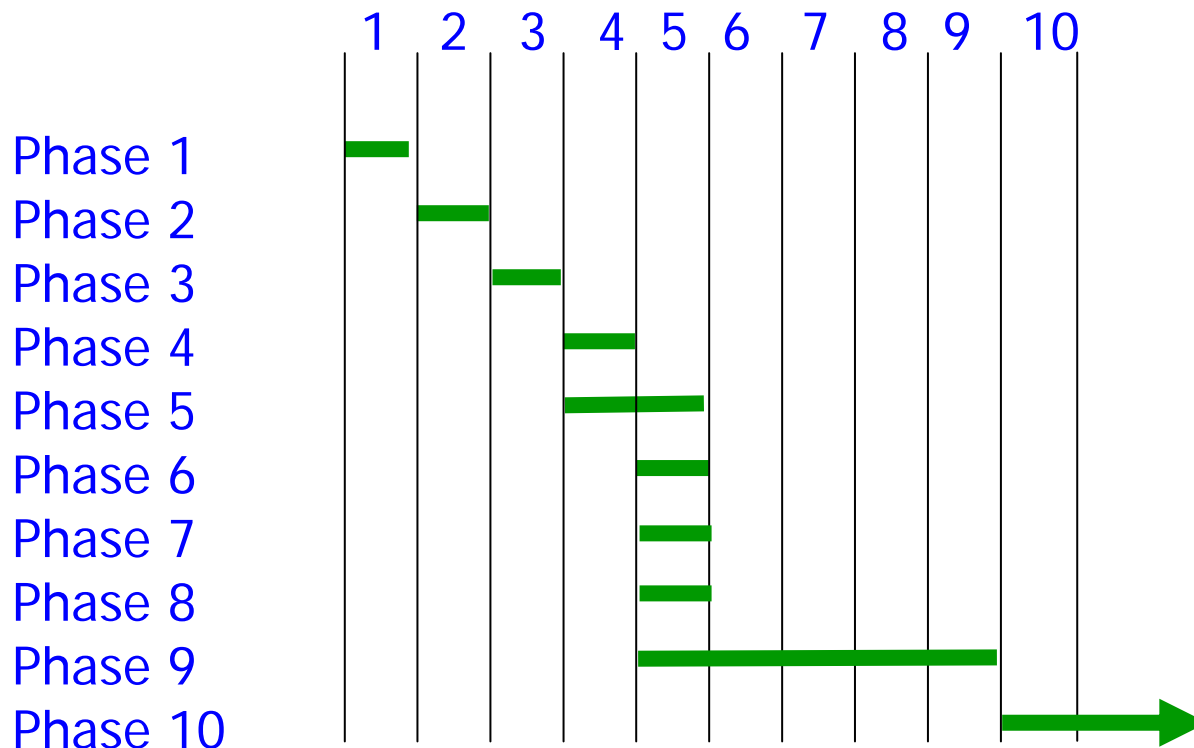
- Beobachtung
 - Es gibt keine Regel zur Umwandlung von Blättern
 - Blätter bleiben immer Blätter
- Was passiert in den Schritten
 - Regel 1: Verlängerung des Labels einer Blattkante
 - Regel 2: Neues Blatt oder: neuer Knoten und neues Blatt
 - Regel 3: Nichts, Abbruch der Phase
- Jede Phase läuft also 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 $1\dots j'$ 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?



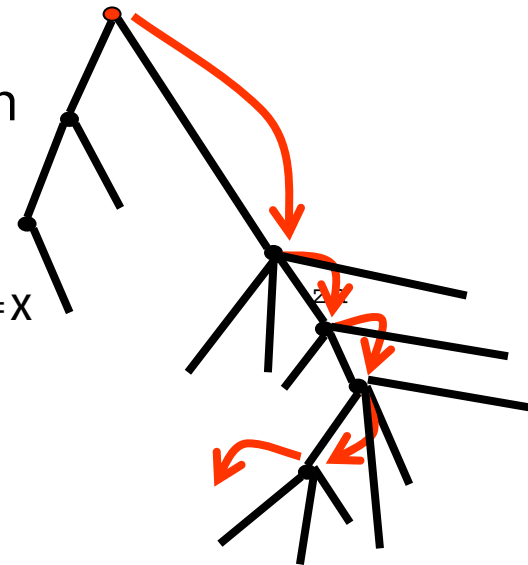
- Schritte markieren einen Pfad von links oben nach rechts unten
- Das können zusammen **höchstens 2^*m Schritte** sein

Inhalt dieser Vorlesung

- Implementierung von Suffixbäumen
- Problem: Größe
 - Schlechtes Laufzeitverhalten auf Sekundärspeichern
- Lösung: Spezielle Konstruktionsalgorithmen

Suche in Suffixbäumen

- Matche Suchstring bis Erfolg oder Mismatch
 - An jedem Knoten **Entscheidung** basierend auf erstem Zeichen des Kantenlabels
 - 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 $\text{label}(k)[1]=x$
 - **Konstante Zeit** pro Knoten
 - **Verkettete Liste**
 - Pointer auf Kinderkanten sind verkettet
 - Reihenfolge der Kinder wie Einfügung (unsortiert)
 - **Lineare (in $|\Sigma|$) Zeit** pro Knoten
 - **Wachsendes, sortiertes Array**
 - Pointer auf Kinderkanten sind alphabetisch sortiert und direkter Zugriff auf Pointer
 - Mit binärer Suche: **$\log(|\Sigma|)$**



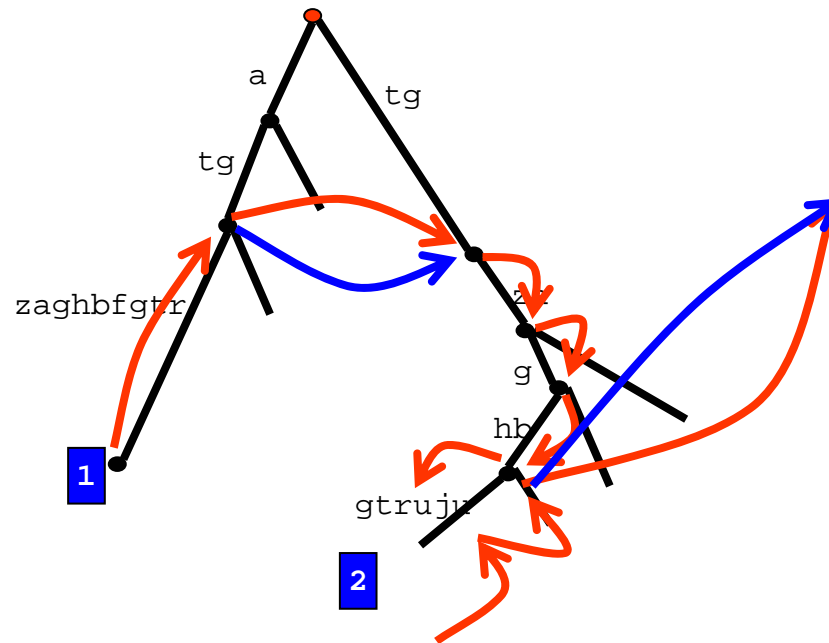
Problem 1: Trade-Off Zeit / Platzbedarf

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

Große Suffixbäume

- Beste Implementierung brauchen ca. 15 Byte / Base für Suffixbäume auf DNA
 - Hängt von der Anzahl innerer Knoten ab
 - Die ist sehr schwer vorherzusagen
- Für Chromosom 1 des Menschen (250 MB) also 3,75 GB
- Speicherbedarf für das komplette humane Genom: 39 GB
- Bei der Berechnung braucht man außerdem Suffix-Links
- Konstruktion wird **im Hauptspeicher** nicht gelingen
 - Aber beachte: 64 Bit Prozessoren!
- Also muss man beachten, dass Teile des (wachsenden) Baumes ab und an auf Platte ausgelagert werden müssen
 - Aber welche?

Konstruktion 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 ...

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 durch Paging
- Suffixbäume gelten /galten als unbenutzbar für große Strings

Scalability

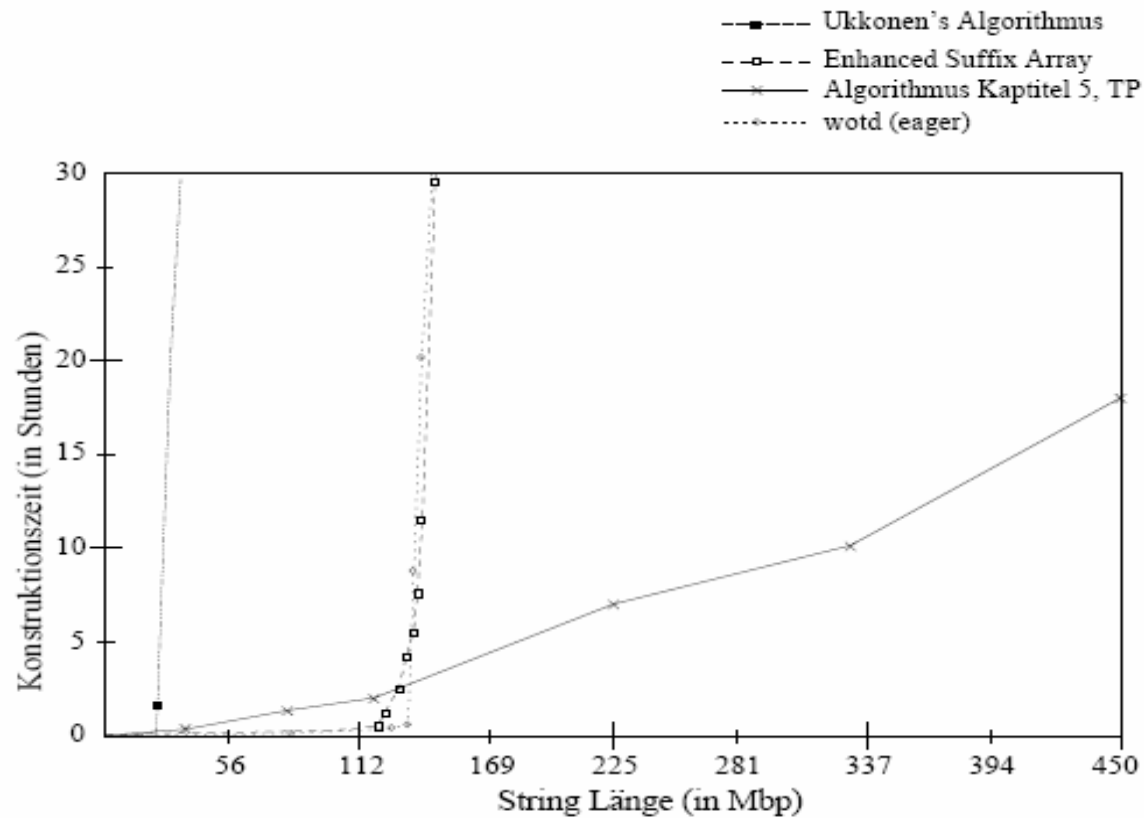


Abbildung 6.3: Entwicklung der Laufzeiten im Vergleich zu anderen Algorithmen

Quelle: G. Witterstein, Dissertation, 2005

Idee: Konstruktionsalgorithmus anpassen

- Hunt, E., Atkinson, M. and Irving, R. W. "A Database Index to Large Biological Sequences". VLDB 2001
- Idee
 - Keine Verwendung von Suffix-Links
 - Zerlege den Suffixbaum in Partitionen
 - Suffixe mit gleichem Präfixen liegen in einem Teilbaum
 - Wähle Präfixe so, dass jeder Teilbaum garantiert in den Hauptspeicher passt
 - Konstruiere jeden Teilbaum naiv
 - Quadratischen Algorithmus benutzen
 - Denn: Suffix-Links gibt es nicht, würden aus Teilbaum hinausführen
 - Kompletten Baum aus (disjunkten) Teilbäumen zusammensetzen

Vergleich zu Ukkonen

- Vorteile
 - Suffixe des Teilbaums wird von Disk gelesen, im Hauptspeicher gebaut, am Ende geschrieben und nie mehr geladen
 - **Sehr wenig IO**
- Nachteile
 - **Multi-Pass Algorithmus**
 - Komplette Sequenz muss pro Partition einmal gelesen werden
 - **Quadratische Komplexität** der Teilbaumkonstruktion
 - Präfixe sorgfältig wählen – sonst doppelte Bestrafung (quadratischer Algorithmus und Paging)
 - Führt zu konservativen Schätzungen
- Beobachtung: IO ist so viel teurer als HS Operationen, dass es sich lohnt

Partitionen

- Präfixe kann man **exakt bestimmen**
 - Häufigkeiten von q -Grammen mit steigenden q zählen
 - Suffix mit Präfix X beginnt an Position i gdw. q -Gramm X sich an Position i befindet
 - Menge der q -Gramme so bestimmen, dass alle Suffixe mit diesem Präfix optimal in den Hauptspeicher passt
 - Präfixe müssen nicht gleich lang sein
 - Erfordert Abschätzen der Größe von Teilsuffixbäumen
 - Sehr teuer
 - Viel Lesen des Strings, bevor überhaupt die Konstruktion beginnt
 - Optimale Lösung ist vermutlich SET-COVER
- ... **oder schätzen**
 - DNA ist nahezu zufällig verteilt
 - Alle q -Gramme gleich häufig
 - Nur die Länge q abschätzen

Komplexität

- Suffixbäume kann man in $O(m)$ konstruieren
- 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
 - Sequenz wird k -mal gelesen
 - Partition hat (m/k) Elemente
 - Zusammen: damit $O(k * (m/k)^2) = O(m^2/k)$

Algorithmus

```
choose set P of prefixes;
construct suffixtree T for P;           // of all partitions
for prefix in P do
    k := find_node(T, prefix);         // Find start of subtree
    for j = 0 to m do                 // Scan sequence on disk
        if S[j..j+|prefix|-1]=prefix 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;                       // Head of tree
```

- Kein paging
- Subtrees liegen hintereinander auf der Platte
 - Gut für Suche – sequentielle Reads, kein Random Access
- Sehr gut **parallelisierbar**

Neuere Verfahren

- Hunt: Konstruktion eines Suffixbaums für 50MB in vertretbarer Zeit mit 1GB Hauptspeicher
- TOP-Q Algorithmus
 - Bedathur, S. J. and Haritsa, J. R., "Engineering a Fast Online Persistent Suffix Tree Construction", ICDE 2004.
 - Verwendung von Ukkonen's Algorithmus
 - Optimierung des Cachemanagements
 - Viel weniger Paging
 - Konstruktion von Bäumen für 70MB

Bisher bestes Verfahren

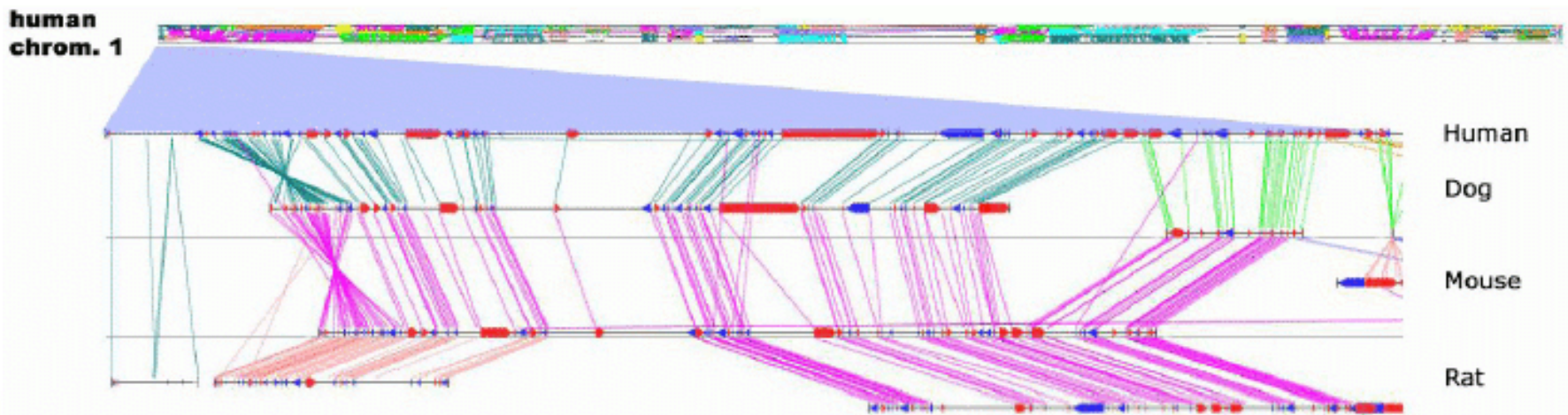
- TDD – Top-Down, Disc-Based
 - Tata, Hankins, Patel: „Practical Suffix Tree Construction“, VLDB 2004
 - Partitionierung ähnlich Hunt
 - Sequenz wird gecached – Vermeidung der k Scans
 - In einer Partition
 - Alle Suffixe bestimmen
 - **Sortieren** (man braucht nur Pointer in die Originalsequenz)
 - In sortierter Reihenfolge Suffixtree bauen
 - Mit einigen Tricks kriegt man $O(n \cdot \log(n))$ Average Case hin
 - Welche Datenstruktur bekommt wie viel Speicher?
 - Sequenz, sortierte Liste, wachsender Partitionsbaum
- Ergebnis: 30h für 3GB
- Auch im **Main-Memory besser als Ukkonnen**
 - Failure Links verursachen Cache Misses im CPU Cache
- Man kennt auch schon garantiert lineare Secondary-Memory Algorithmen, sind aber wohl langsamer als TDD

MUMmer

- Eine Anwendung für große Suffixbäume
- Whole Genome Alignment
- MUMmer: Finding Maximally Unique Matches
 - Delcher, Phillippy, Carlton, Salzberg, „Fast algorithms for large-scale genome alignment and comparison“, NAR 1999

Whole Genome Alignment

- Genome unterliegen Evolution
- Neben Punktmutationen treten großräumige Veränderungen auf
 - Duplikation von Chromosomen
 - Duplikation von langen DNA Abschnitten
- Duplikate unterliegen dann unabhängiger Evolution
- In nahe verwandten Organismen (z.B. Säugetiere) sind Genome „durcheinander geschüttelte“, approximative Kopien voneinander



Finden von MUMs

- Beobachtung
 - Evolutionärer Abstand in nahe verwandten Spezies ist klein
 - Trotz unabhängiger Evolution findet man **genügend lange, identische Sequenzabschnitte**
 - Also: Anwendung für exaktes Stringmatching
- Aufgabe
 - Gegeben zwei Chromosome X, Y (Mensch und Maus)
 - Finde **Bereiche von X, die identisch in Y** vorkommen
- Genauer: Finde **längste identische** Teilsequenzen über einer Mindestlänge
- Außerdem: Um klare Zuordnungen zu ermöglichen, beschränkt man sich auf **eindeutige Teilsequenzen**
 - Dürfen in X und Y nur jeweils einmal vorkommen
- Wir wollen alle MUMs zwischen Maus und Menschen finden
 - Vorschläge?

Möglichkeit 1

- Bilde **alle Kombinationen $S=X\$Y\%$**
 - X ein Chromosom des Menschen
 - Y ein Chromosom der Maus
- Baue den Suffixbaum T für S; markiere Knoten mit 1,2
- Suche alle inneren Knoten, die
 - Genügend tief liegen
 - Also die Mindestlänge haben
 - Keinen Kindknoten haben, der mit 1 und 2 markiert sind
 - Sonst sind es keine maximalen gemeinsamen Strings
 - Nur ein Kind haben, dass mit 1, und eines, das mit 2 markiert sind
 - Sonst ist der Substring nicht eindeutig

Probleme mit Möglichkeit 1

- S kann sehr groß sein ($>500\text{MB}$)
 - Kann man nur „experimentell“ bauen
 - Dauert lange
- Wir brauchen c^2 solcher Strings
 - c : Anzahl Chromosomen Menschen / Maus
- Seien alle Chromosomen m Zeichen lang
 - Wir bauen Suffixbäume in $O(2^m)$
 - In einem Paar findet man in $O(2^m)$ alle MUMs
 - Das machen wir c^2 mal
 - Zusammen: $O(c^2 * 2^m)$

Möglichkeit 2

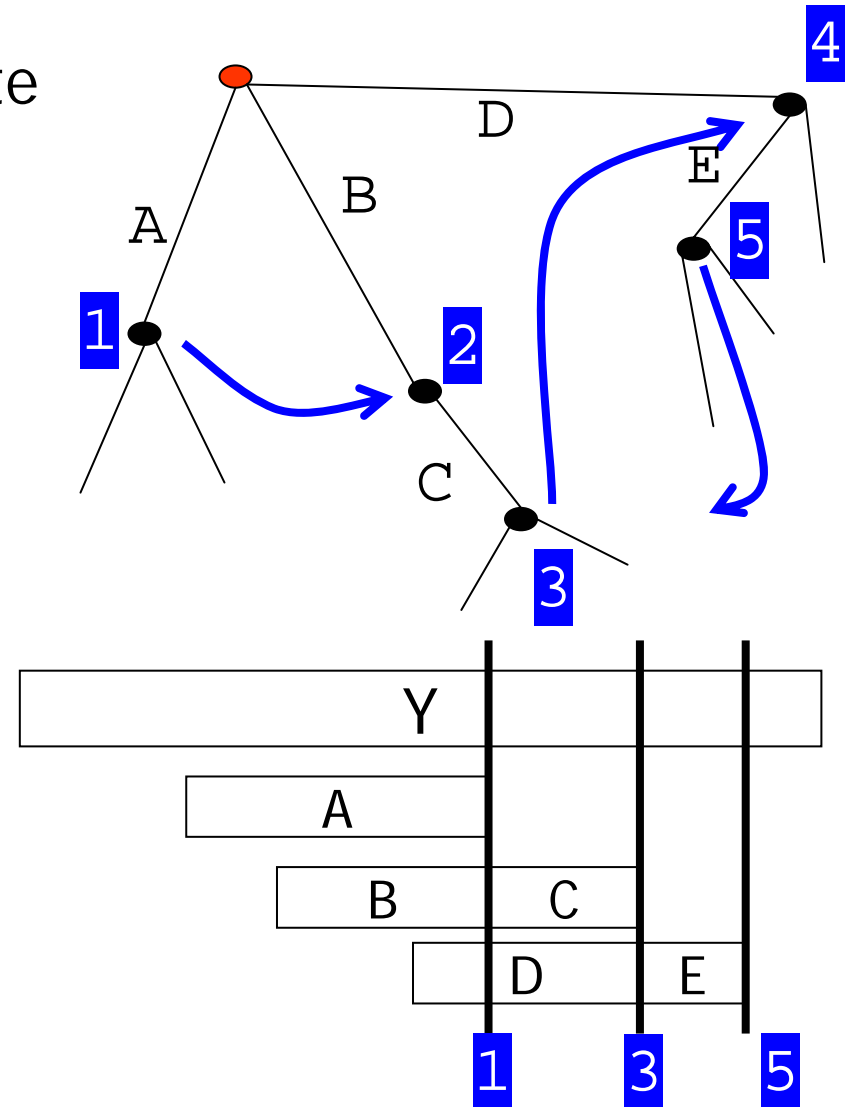
- Idee nach [Chang and Lawler](#); verwendet in MUMmer 2
 - Wir bauen Suffixtrees nur für ein Chromosom X
 - [Suffix-Links](#) werden behalten
 - Alle Chromosomen der anderen Spezies „streamen“ wir gegen X
- Komplexität
 - Man baut $2c$ „kleinere“ Suffixbäume in jeweils $O(m)$
 - Gegen jeden streamen wir c mal in $O(m)$
 - Zusammen: $O(2c*m) + O(c^2*m)$
- Hauptvorteil: IO vernachlässigbar
 - Suffixbäume kann man im Hauptspeicher bauen
 - Vergleichssequenz wird sequentiell gelesen

„Streamen“ gegen einen Suffixbaum

- Sei T der Suffixbaum für Chromosom X , Y das andere Chromosom
- Wir matchen Y mit T
 - Wir schieben einen **Pointer** durch Y und einen durch T
 - Bei Matches verschieben wir beide Pointer
 - Wenn Mismatch an Position i in Y und unter Knoten k in T
 - Überprüfe Tiefe (Minimaltiefe erreicht?)
 - Überprüfe Kinder (mehr als 1 Kindknoten – Match ist nicht eindeutig)
 - Ggf: Gib MUM aus
 - Es seien l Zeichen zwischen k und dem Mismatch
 - Mismatches können auf Kantenlabels sein
 - Folge dem **Suffix-Link zu Knoten k'**
 - Den String von Root zu k' haben wir in T und P schon gesehen
 - Matche ab **Position $i-l$ in P** weiter

Illustration

- Vereinfachte Annahme: Letzte Matches immer in Knoten
- Mismatch in Knoten 1
 - mit Label A
- SL zu Knoten 2
 - mit Label B
 - B ist Suffix von A
- Weiter in X matchen bis Mismatch in Knoten 3
 - mit Label BC
- SL zu 4
 - mit Label D
 - D ist Suffix BC
- Weiter in Y matchen bis ...
- ...



Fast richtig

- Wenn man ab k' weiter matched
 - Und einen MUM findet
 - Dann ist der nicht notwendigerweise maximal
 - Das rechte Ende ist klar
 - Der aktuelle Mismatch
 - Aber das **linke Ende nicht**
- Linke Enden müssen explizit geprüft werden
 - In Y und in T vor dem Suffix nach links vergleichen bis Mismatch
- Damit matched man **Zeichen in Y mehrmals**
 - Außerdem matched man die Zeichen zwischen k und einem Mismatch zweimal
- Aber: Wenn minimale MUM Länge sinnvoll groß, gibt es nur wenige MUMs – **praktisch lineare** Laufzeit (in $|Y|$)

Zwei weitere Tricks

- Unsere Matches sind bisher nur eindeutig in T
 - Auch in die Gegenrichtung suchen (Suffixtree für Y)
 - Man muss wieder die gleichen max. Substrings finden
 - Nur übernehmen, wenn wieder Unique
- Es gibt viel zu viele MUMs
 - Man sucht Cluster– viele MUMs in der gleichen Reihenfolge hintereinander
 - Zwischen zwei MUMs liegen nicht konservierte Regionen
 - Da kann auch ein drittes MUM liegen
 - Neues Problem: Gegeben zwei Positionsarrays von MUMs, finde die **längste gemeinsame Subsequenz**

Suffixbäume: 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: $O(k \cdot m)$
 - Trick: Ähnlich Keywordtrees mit Wildcards
 - K-Mismatch Problem: $O(k \cdot m)$
 - Trick: Ebenso
 - Beide benutzen lca um in konstanter Zeit das längste Präfix zweier Suffixe zu finden
 - Details: [Gusfield, Kapitel 9.1, 9.3]

Zusammenfassung

- Suffixbäume sind ein extrem effizientes Hilfsmittel für viele Stringprobleme
 - Lineare Suche ist berücksend
 - Viele schicke Tricks: Maximale Substrings, maximale Repeats, Matching mit k-Mismatches, ...
- Aber sie brauchen viel Platz
 - Spezielle Verfahren zur Konstruktion großer Suffixbäume
 - Alternative: [Suffix-Arrays](#)