

Algorithmische Bioinformatik

Suffixbäume auf Sekundärspeicher

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: $O(m)$

Voraussetzungen

- Definition:

*Seit T ein Suffixbaum für S . Der **implizite Suffixbaum T'** entsteht aus T durch*

- *Entferne alle Vorkommen von „\$“ aus allen Labels*
- *Entferne alle Kanten ohne Label*
- *Entferne alle inneren Knoten mit weniger als 2 Kindern; konkateniere die Label der eingehenden und der ausgehenden Kante zu dem der neuen (durchgehenden) Kante*

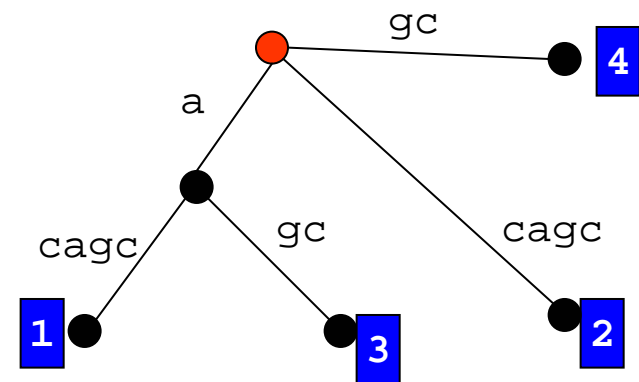
- Das ist die Definition; wir werden ganz anders vorgehen

Grundaufbau Ukkonen's Algorithmus

- Induktive Konstruktion impliziter Suffixbäume für jedes Präfix von S
 - Wir konstruieren nacheinander alle T_i , d.h., implizite Suffixbäume für $S[1..i]$
 - **Startpunkt** T_1 : Wurzel und ein Knoten mit Kantenlabel $S[1]$
 - **Phasen**: Konstruktion von T_{i+1} aus T_i
 - **Abschluss**: Transformation von T_m in den „echten“ 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 $S[i+1]$
 - Der letzte Schritt jeder Phase verlängert das **leere Suffix**
 - Entspricht dem Einfügen von $S[i+1]$
 - Reihenfolge der Schritte: von links nach rechts ($S[1..i]$, $S[2..i]$, ...)
- Wie wird verlängert?
 - Drei **Extensionsregeln**

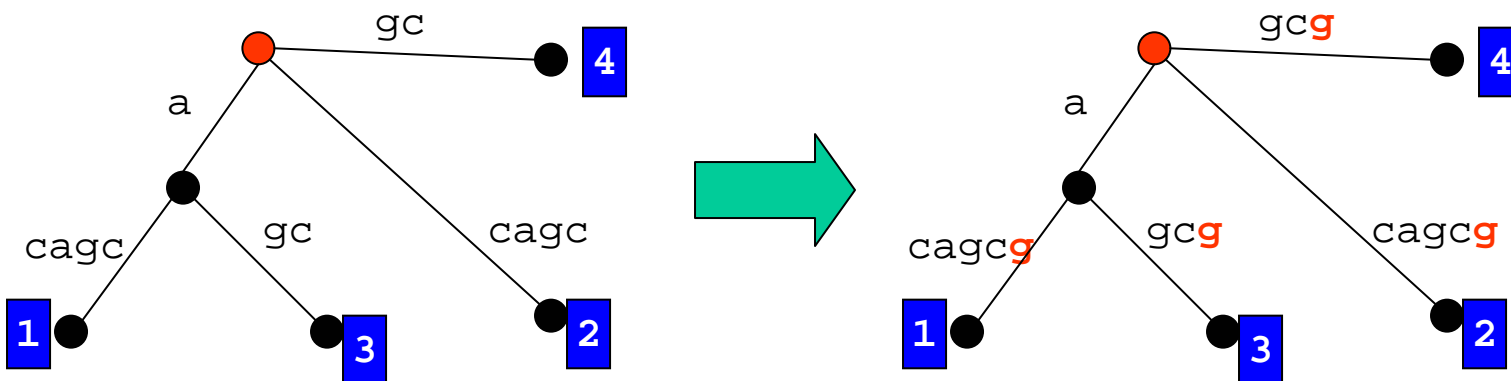
Extensionsregeln

- In Extensionsschritt j in Phase $i+1$ verlängern wir $S[j..i]$ um das Zeichen $S[i+1]$
 - Sei $b=S[j..i]$
- Matche b in T_{i+1} (im Entstehen)
 - 3 mögliche Situationen können entstehen
- Beispiel
 - $S = „acagcg“$
 - Wir haben T_5 und bauen T_6
 - Also: Alle Suffixe von „acagc“ um $S[6] = „g“$ erweitern



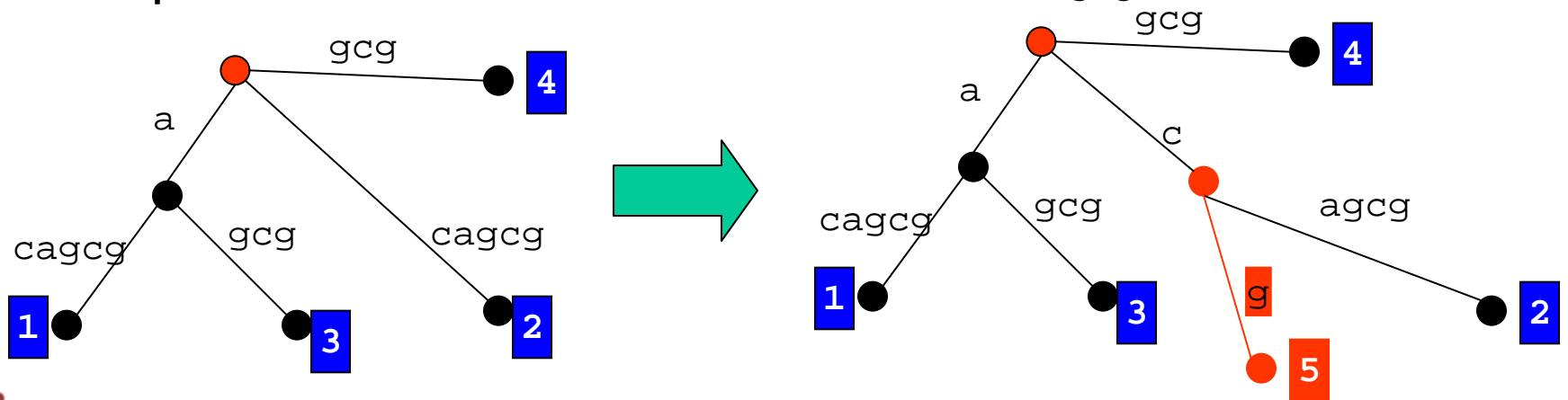
Extensionsregel 1

- 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“]



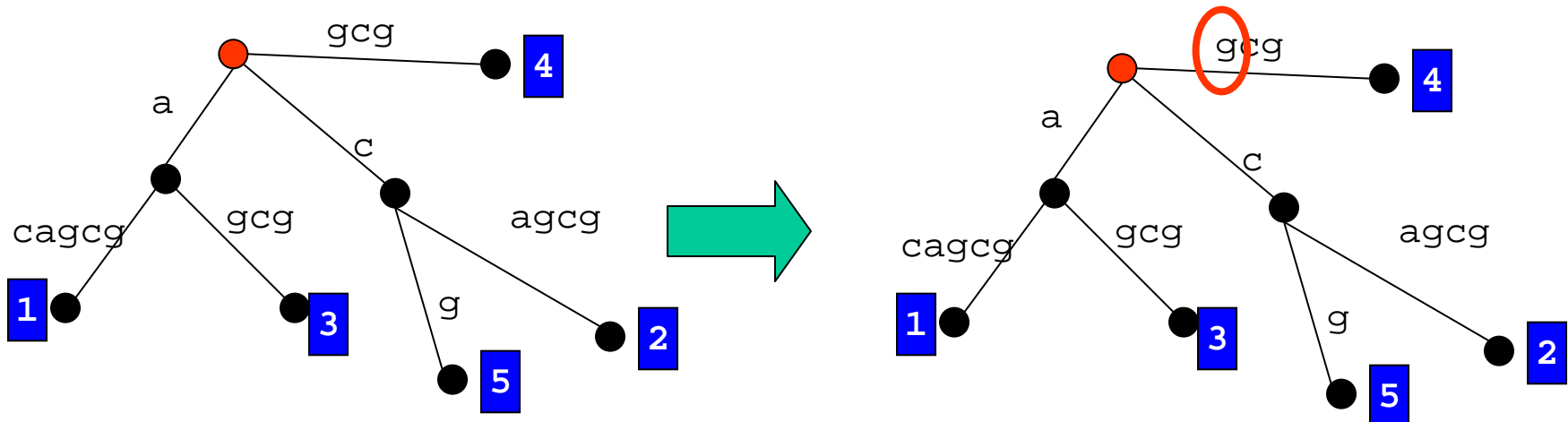
Extensionsregel 2

- 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, altes Suffix „c“ ($S = „acagcg“$)



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, altes Suffix „“ ($S = \text{„acagcg“}$)



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;
```

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

Suffix-Links

- Wir wenden uns dem „innersten“ Problem zu – immer wieder das **Ende von Suffixen** zu finden
 - Wir reduzieren die Zeit für alle „match $S[j..i]$...“ Operationen in einer Phase auf zusammen $O(m)$
- Intuition
 - Für jedes gefundene $S[j..i]$ gibt es per Konstruktion irgendwo schon $S[j+1..i]$
 - Wo? Wir wollen nicht suchen, sondern **Suffix-Links** merken und direkt anspringen
 - Außerdem: Da $S[j..i]$ und $S[j+1..i]$ bis auf $S[j]$ identisch sind, werden wir von Knoten zu Knoten springen – **Skip/Count Trick** – statt Zeichen einzeln im Baum zu matchen
 - Zusammen: Komplexität in einer Phase abhängig von **Anzahl der Knoten**, nicht der Zeichen

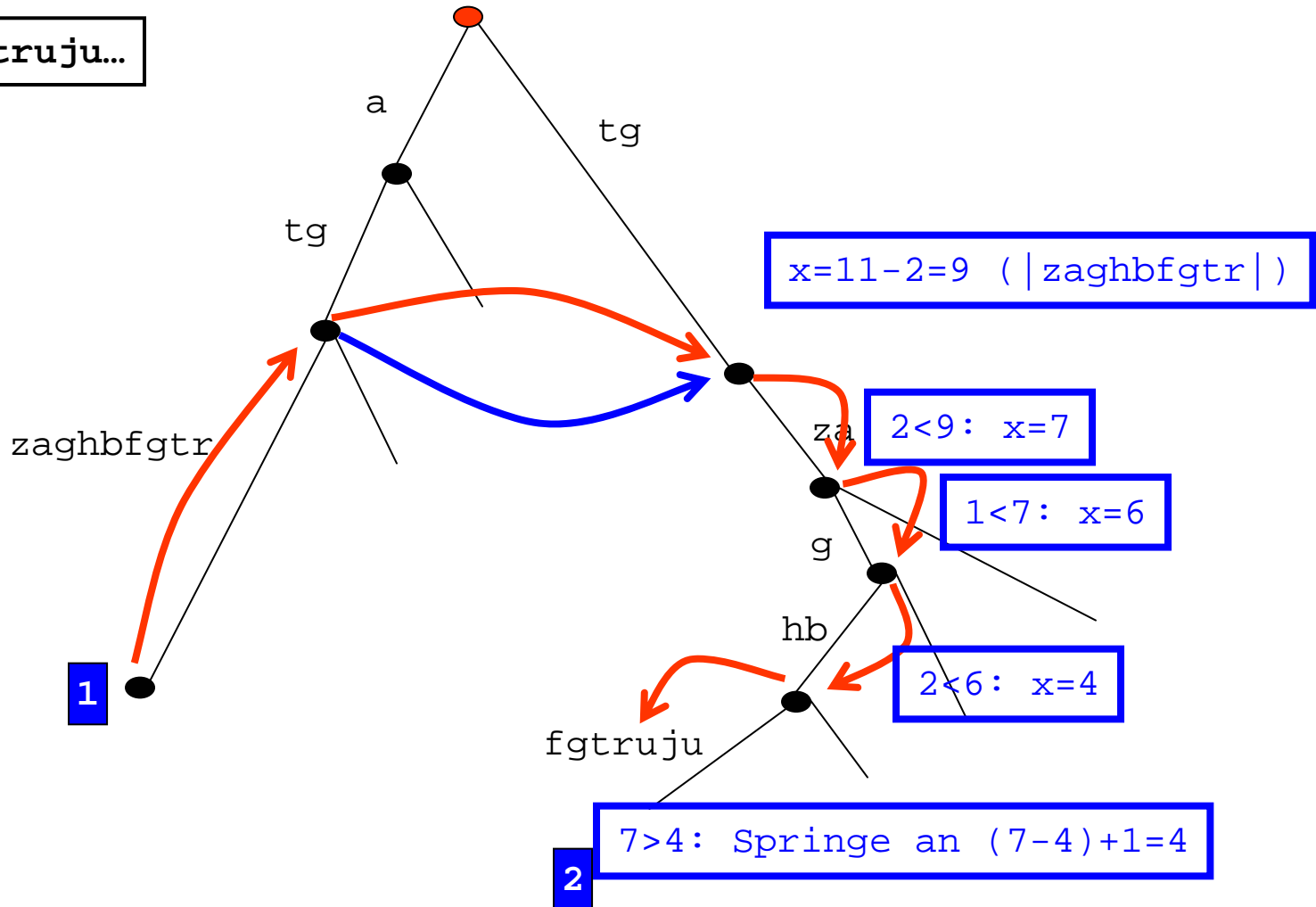
Verwendung der Suffix-Links

- 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 einen Ast herunter (siehe später: Skip/Count)
 - 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



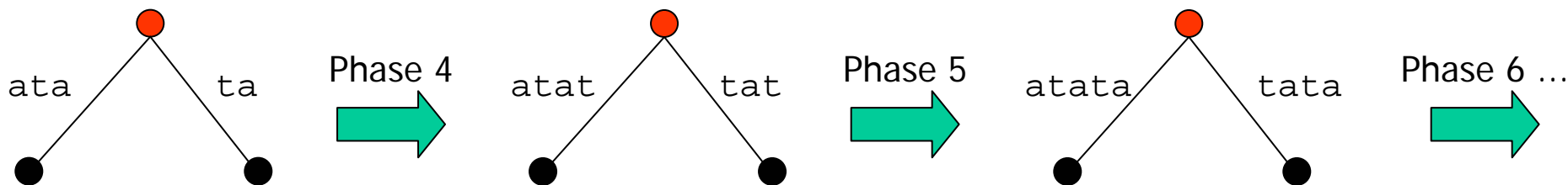
Skip/Count Beispiel

atgzaghbfgtruju...



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: Tue nichts, denn es gibt das Suffix $S[j..i+1]$ schon im Baum (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 Extension 3

Stop nach Extension 3

Extensionsregel, Abkürzung 2

- Wir unterscheiden
 - **Explizite Extension**: Schritt mit Anwendung einer Regel
 - **Implizite Extension**: Schritt nur „gedacht“
- Alle Schritte nach einer erste Anwendung von Regel 3 sind implizit
- Gesucht
 - Welche Schritte können wir in einer Phase noch sparen?
- Wichtiger und einfacher Trick für das folgende
 - An eine Kante k zu einem Blatt schreiben wir nicht das Label $\text{label}(k)$, sondern Indizes p, q mit $\text{label}(k) = S[p..q]$

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 $1,2,2,2,1,2,1,3,stop$
 - Erst einige Anwendungen von 1 oder 2, dann einmal 3
 - Bei jeder Anwendung von 1 oder 2 wird das Label einer Blattkante verlängert oder Blatt+Kante geschaffen
 - Sei j' die **letzte Extension** mit Regel 1 oder 2
 - 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

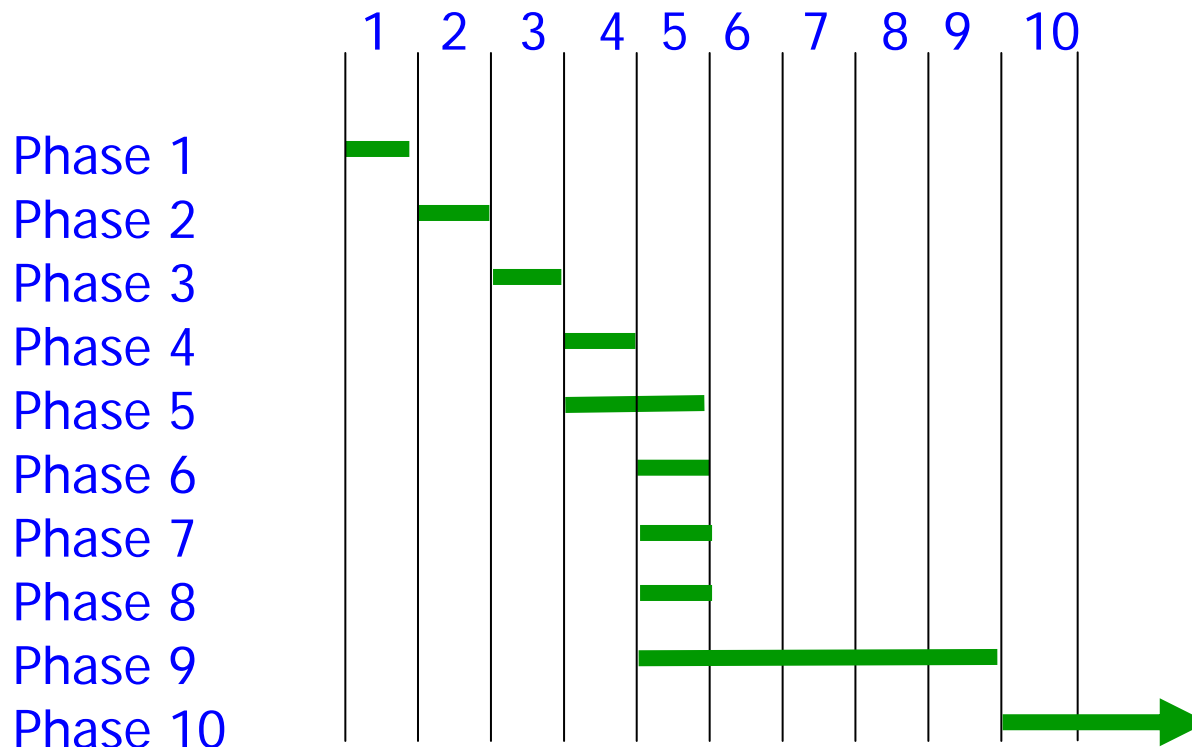
- Diese Schritte können wir uns schenken
 - Extensionen bis j' werden ab Phase $i+1$ implizit erledigt
 - Wir müssen lediglich Blattkanten anpassen
 - Dazu erhalten Blattkanten statt (p,q) die Beschriftung (p,E)
 - E steht für „Bis zum Ende“
 - Am Ende wird jede Blattkante bis zum Ende von S gehen (denn Blättern repräsentieren schließlich Suffixe von S)
- Zusammen: Eine Phase im einzelnen
 - Überspringe alle Schritte bis j' der letzten Phase
 - Führe 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

- Theorem
 - *Ukkonen's Algorithmus benötigt $O(m)$ zur Konstruktion von T_m*
- Beweisidee
 - Jede Phase führt **explizit höchstens einen Extensionsschritt** aus, der schon in einer früheren Phase ausgeführt wurde
 - Das war der erste Schritt mit Regel 3 aus der letzten Phase
 - Danach kommt in der Phase entweder
 - ein weiterer expliziter Schritt
 - Der ist keine Wiederholung und wird auch nicht wieder wiederholt
 - oder es greift Extensionsregel 3 und die Phase ist beendet
 - Es gibt nur m Phasen
 - Alle Phasen zusammen führen also **höchstens $2m$ explizite** Extensionsschritte aus
 - Außerdem werden insgesamt höchstens m Knotensprünge ausgeführt

Komplexität

- Welche Schritte haben wir ausgeführt?



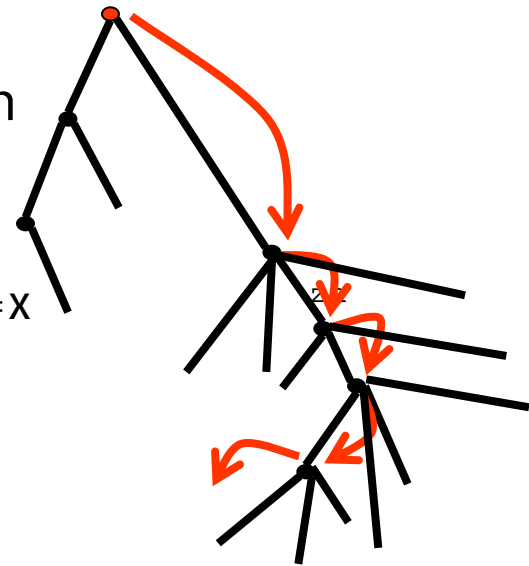
- Schritte markieren einen Pfad von links oben nach rechts unten
- Das können zusammen **höchstens 2^*m Schritte** sein

Inhalt dieser Vorlesung

- Implementierung von Suffixbäumen
 - Problem: Größe
 - Schlechtes Laufzeitverhalten auf Sekundärspeichern
 - Lösung: Spezielle Konstruktionsalgorithmen
- MUMmer
 - Whole Genome Alignment
 - Gemeinsame Substrings zweier sehr grosser Strings

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: **$\log(|\Sigma|)$**



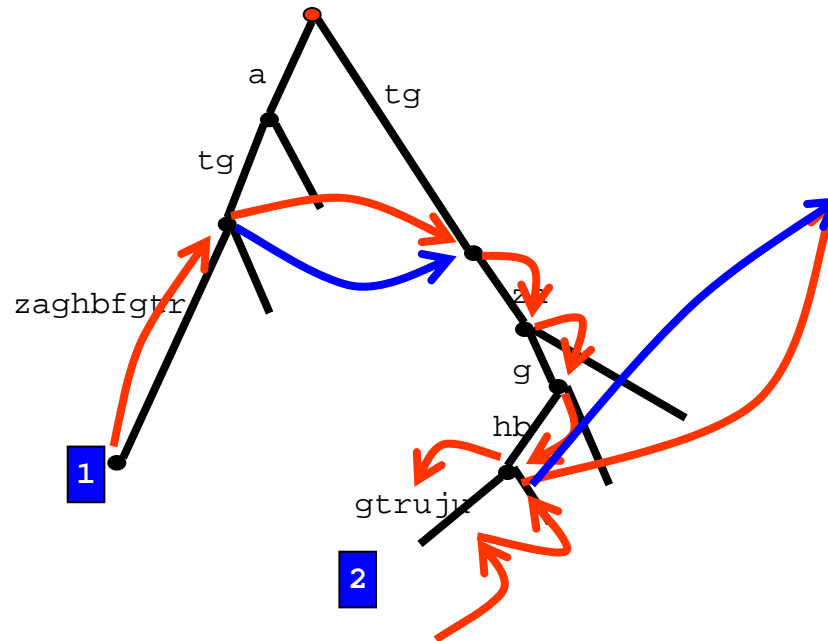
Problem 1: Trade-Off Zeit / Platzbedarf

- 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
 - Zusätzlicher Platzverbrauch
- Konstruktion wird **im Hauptspeicher** nicht gelingen
 - Oder jetzt doch? 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 galten lange als unbenutzbar für große Strings

Scalability

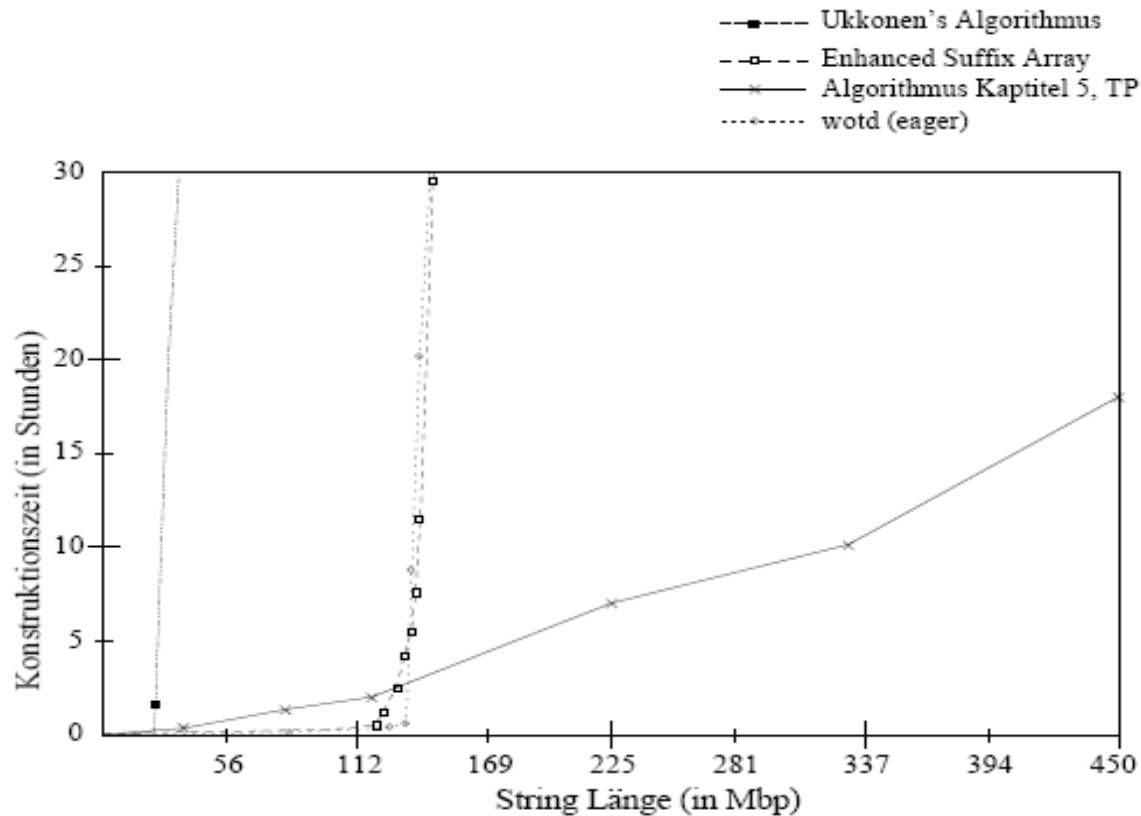


Abbildung 6.3: Entwicklung der Laufzeiten im Vergleich zu anderen Algorithmen

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

Bis dieses Jahr bestes Verfahren

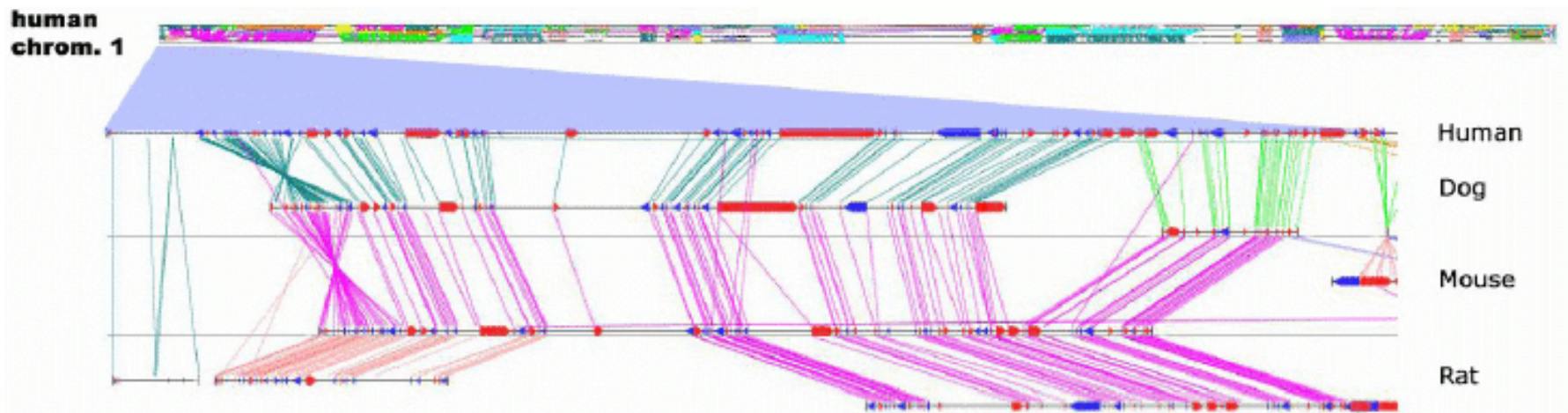
- TDD – Top-Down, Disc-Based
 - Tata, Hankins, Patel: „Practical Suffix Tree Construction“, VLDB 2004
 - Partitionierung ähnlich Hunt
 - Sequenz wird gecached – Vermeidung der k Scans
 - In einer Partition
 - Alle Suffixe bestimmen
 - **Sortieren** (man braucht nur Pointer in die Originalsequenz)
 -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

MUMmer

- Eine Anwendung für 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 konservierter Bereiche

- 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 ($>500\text{MB}$)
 - 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: $O(c^2 * 4^m)$
 - Achtung: Missbrauch der O-Notation

Möglichkeit 2

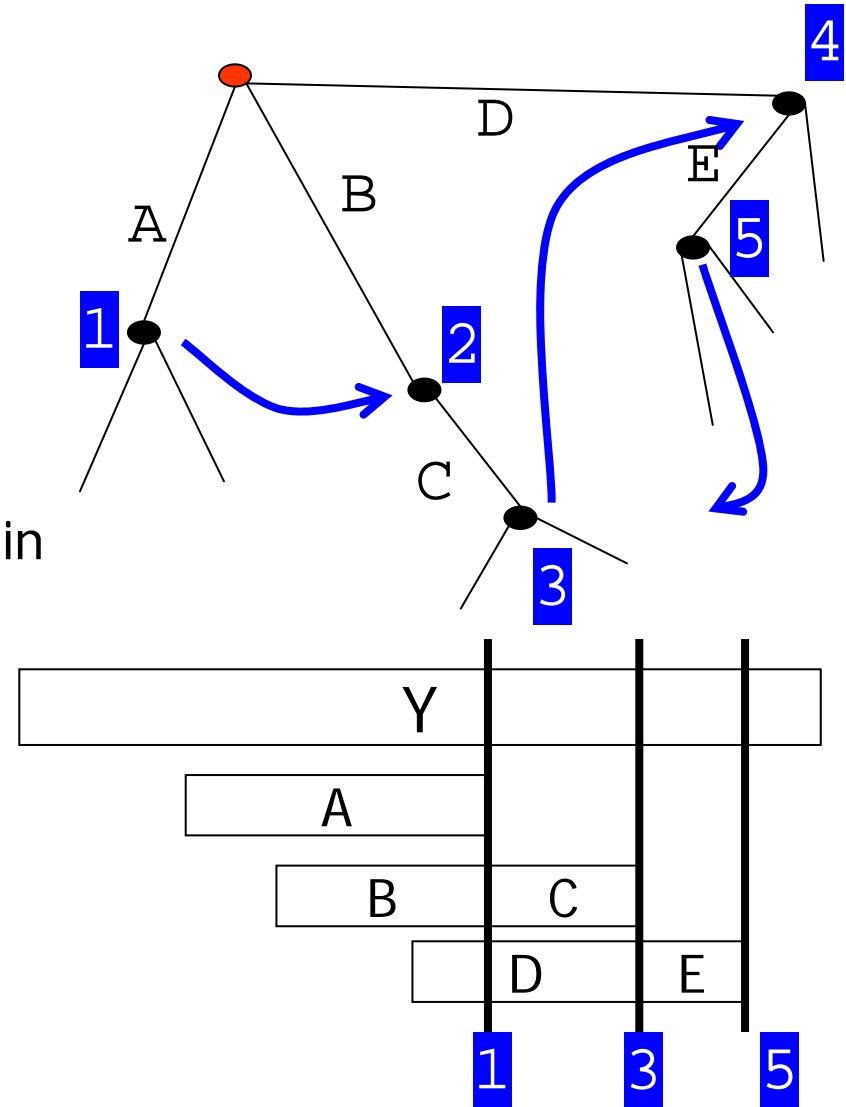
- Idee nach **Chang and Lawler**
 - (verwendet in MUMmer 2)
 - Wir bauen Suffixtrees für ein Chromosom X
 - **Suffix-Links** werden behalten
 - Alle Chromosomen der anderen Spezies „streamen“ wir gegen X
- Komplexität
 - Man baut $2c$ „kleinere“ Suffixbäume in jeweils $O(m)$
 - Gegen jeden streamen wir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

„Streamen“ gegen einen Suffixbaum

- Sei T der Suffixbaum für Chromosom X , Y ein Vergleichschromosom
- Wir matchen Y mit T
 - Wir schieben einen **Pointer** durch die Sequenz Y und einen durch T
 - Bei Matches verschieben wir beide Pointer
 - Wenn Mismatch an Position i in Y und nach Knoten k in T
 - Überprüfe Tiefe (Minimaltiefe erreicht?)
 - Prüfe Unterbaum: wenn mehr als 1 Blatt, ist der Match nicht eindeutig
 - Ggf: Gib potentiellen MUM aus (kann noch mal in Y kommen)
 - 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 Y schon gesehen
 - Skip/Count-Trick bis Position i ; dann weitermatchen

Illustration

- Vereinfachte Annahme: Letzte Matches immer in Knoten
- Mismatch in Knoten 1
 - Mit Label A
- SL zu Knoten 2
 - Mit Label B
 - B ist längstes echtes 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 sequentiell vergleichen
- Damit matched man **Zeichen in Y mehrmals**
- 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 als MUM übernehmen, wenn wieder Unique
- Um Eindeutigkeit im Genom zu garantieren, muss man am Ende noch alle MUMs aus allen Vergleichen abgleichen
- 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](#)