Bioinformatik

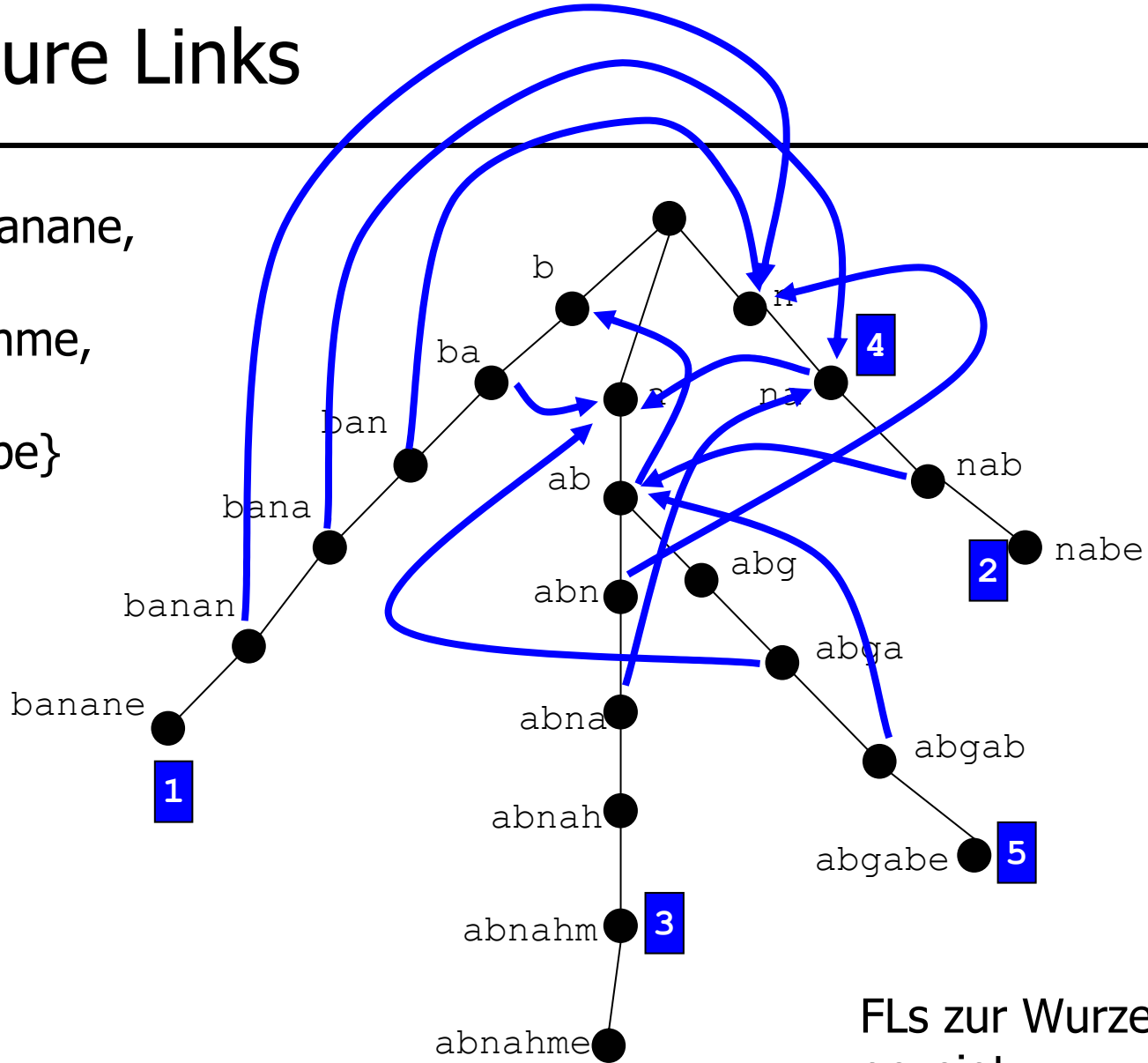
Suffixbäume

Ulf Leser

Wissensmanagement in der Bioinformatik

Failure Links

P={banane,
nabe,
abnahme,
na,
abgabe}



FLs zur Wurzel nicht gezeigt

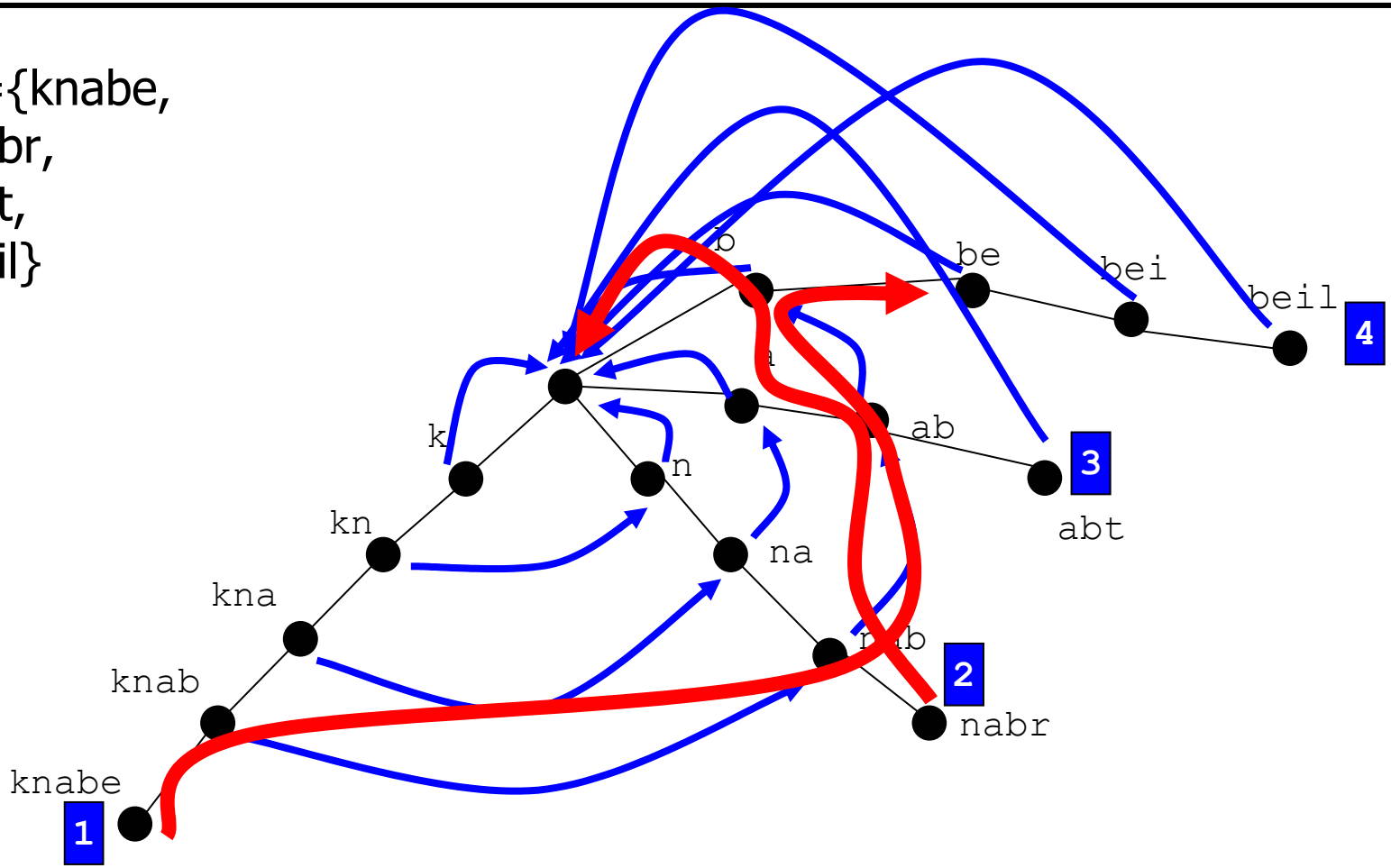
Konstruktion der Failure Links

- Bisher
  - Mit Failure Links ist die Suchphase O(m)
  - Konstruktion des Keyword Trees ist O(n)
  - Wie lange braucht man, um Failure Links zu berechnen?
  - ... und dann war da noch das spezielle Problem ...
- Definition
  - *Sei depth(k) die Tiefe des Knoten k (Abstand zu root(K))*

- Wir bauen erst (in linearer Zeit) den Keyword-Tree
- Dann alle Failure Links in O(n)
  - Beachte: Failure Links zeigen immer zu echten Suffixen
  - D.h., wenn depth(k)=X und depth(fl(k))=Y, dann muss gelten: Y<X
  - Wir konstruieren alle Failure Links per Breitensuche

Beispiel

P={knabe,
nabr,
abt,
beil}

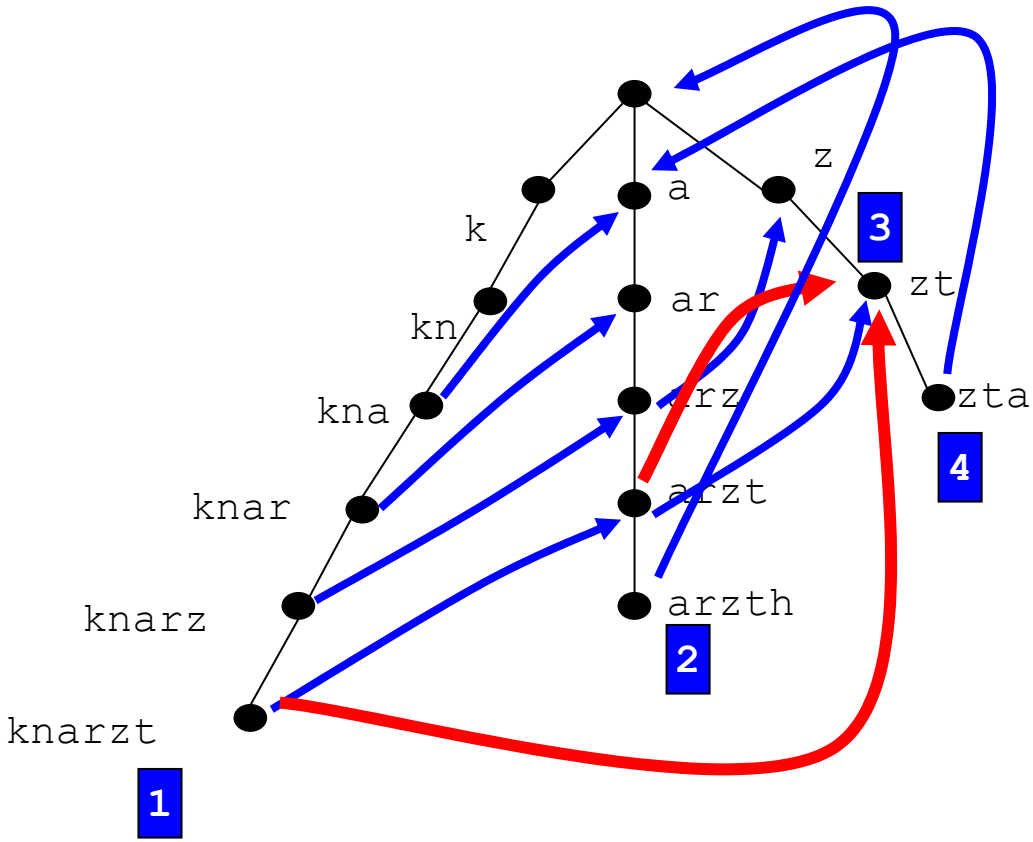


Spezialfall

- Problem: Pattern, die andere Pattern enthalten
- Lösung: Output Links
  - Wir konstruieren noch einen Pointer für (einige) Knoten in k
- Erst eine Beobachtung über die Problemfälle
  - Sei P_1 in P_2 enthalten (also unser Problemfall)
  - Dann muss P_1 Suffix von $P_2[1..i]$ für irgendein $i \geq |P_1|$ sein
  - Wenn P_1 das längste echte Suffix von $P_2[1..i]$ ist, dann gilt
    - $fl(P_2(i))=P_1$
  - Sonst gibt es ein P_3 mit
    - P_3 ist längstes Suffix von $P_2[1..i]$
    - Also gilt $fl(P_2(i)) = P_3$
    - Wiederum gilt: P_1 ist Suffix von P_3 – ist es auch das längste?
      - Suche rekursiv über Failure Links
      - Schließlich muss man bei P_1 ankommen

Breadth-First Konstruktion von Output Links

P={knarzt,
arzth,
zt,
zta}



Gesamtkomplexität Aho-Corasick

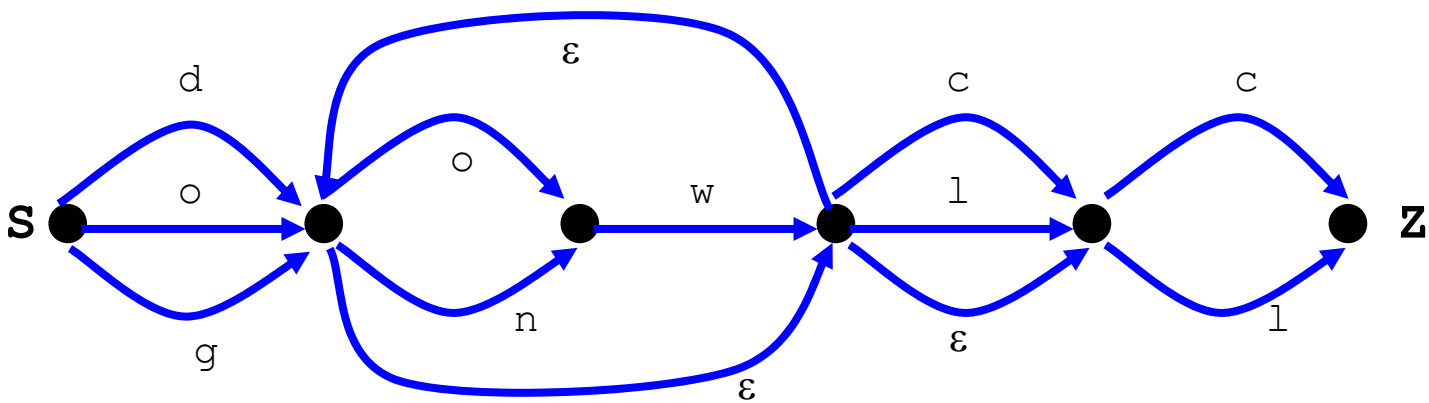
- Gegeben P={...} mit Gesamtlänge n, T mit |T|=m
- Elemente des Algorithmus
  - Berechnung Keyword Tree für P O(n) (trivial)
  - Berechnung Failure Links O(n) (BF)
    - Dabei auch Berechnung der Output Links
  - Suche mit Failure/Output Links O(m+k)
  - Zusammen O(n+m+k)
- Ziel erreicht ...

Algorithmus

- Clevere Verwendung von Aho-Corasick
  - Gegeben: Pattern P, Template T
  - Initialisiere Integerarray C=[0,0,0,...,0], mit |C|= |T|
  - P‘={P_1,...P_s} sei die Multimenge aller maximalen Substrings in P ohne Wildcards, l_1,...l_s ihre Startpositionen
  - Bilde den Keyword Tree für P‘ mit FL/OL und suche mit AC
    - Wenn ein P_i an Position j in T gefunden wird, dann
      - z=j-l_i+1 ist der (potentielle) Startpunkt von P in T
      - Wenn z>0, dann setze C[z] = C[z]+1
  - Schließlich: Jede Position x mit C[x]=k repräsentiert ein Vorkommen von P in T an Position x
    - Denn dort wurden alle k Subpattern P_i gefunden
- Komplexität O((n-s)+m+k)
  - Array wird in konstanter Zeit während der AC Suche aktualisiert
  - Immer wenn dabei C[z]=k, gib z aus.

Problem

- Gegeben regulärer Ausdruck P, Template T
- Gesucht: Alle Vorkommen von P in T
- Regulärer Ausdruck ist äquivalent zu einem nichtdeterministischen regulären Automaten
  - Konstruktion des Automaten ist straight-forward
  - Beispiel: (d|o|g)((n|o)w)*(c|l|ε)(c|l)
  - Matcht z.B. dnwnwowc, ol, gowll, ...



Komplexität

- Kritisch ist nur der Schritt N(i-1) $\rightarrow$ N(i)
  - Matchen von T(i) in konstanter Zeit
  - Danach können höchstens e ε-Kanten folgen, wenn e die Anzahl von ε-Kanten in G(p) ist
    - N ist eine Menge, keine Multimenge!
  - Also ist dieser Schritt O(e)
- Wir berechnen m Mengen N(1) ... N(m)
- Also O(m*e)
- Aber
  - Ein regulärer Ausdruck mit n Symbolen hat höchstens n ε-Kanten
  - Sonst kann er minimiert werden
- Zusammen: O(m*n)

Inhalt dieser Vorlesung

- Suffixbäume
- Verwendung von Suffixbäumen
- Naive Konstruktion

Problemstellung

- Bisherige Algorithmen
  - Gegeben ein Template T (m) und ein oder mehrere 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 eine lange Zeichenkette T
    - Z.B. Komplettes Genom des Menschen
  - Benutzer weltweit schicken kontinuierlich sich ändernde Sequenzstücke (P)
- Also: T (und nicht P) vorverarbeiten
- Lösung: Suffixbäume

Unterschiede zu anderen Algorithmen

- Gegeben: Festes T, k verschiedene P mit jeweils Länge n
  - Konstante Länge ist Vereinfachung zur Verdeutlichung
- Boyer-Moore, KMP
  - Jedes P vorverarbeiten in O(k*n)
  - Jeweils in T suchen O(k*m)
  - Gesamt: O(k*m+k*n)
  - Suffixbäume: O(k*n [+ m])
- Aho-Corasick
  - Vorverarbeitung der Pattern: O(k*n)
  - Suche nach allen Pattern parallel: O(m)
  - Gesamt: O(k*n + m)
  - Aber: Pattern sind hier nicht vorab bekannt, T ist fest

Übersicht – Suche nach k Pattern

	Vorverarbeitung	Suche
Boyer-Moore	O(k*n)	O(k*m)
Aho-Corasick	O(k*n)	O(m)
Suffixbäume	O(m)	O(k*n+m)

- Keyword-Trees und Suffixbäume sind tw. komplementär
  - Bessere Datenstruktur abhängig von |P| > |T| oder |P| < |T|
  - Suffixbäume haben viele, viele Erweiterungen (siehe Gusfield)
- Suffixbäume sind auch für einfache Suchen (ein P in T) theoretisch nicht schlechter als KMP etc.
  - Aber in der Praxis schon – längere Laufzeiten, höherer Speicherverbrauch

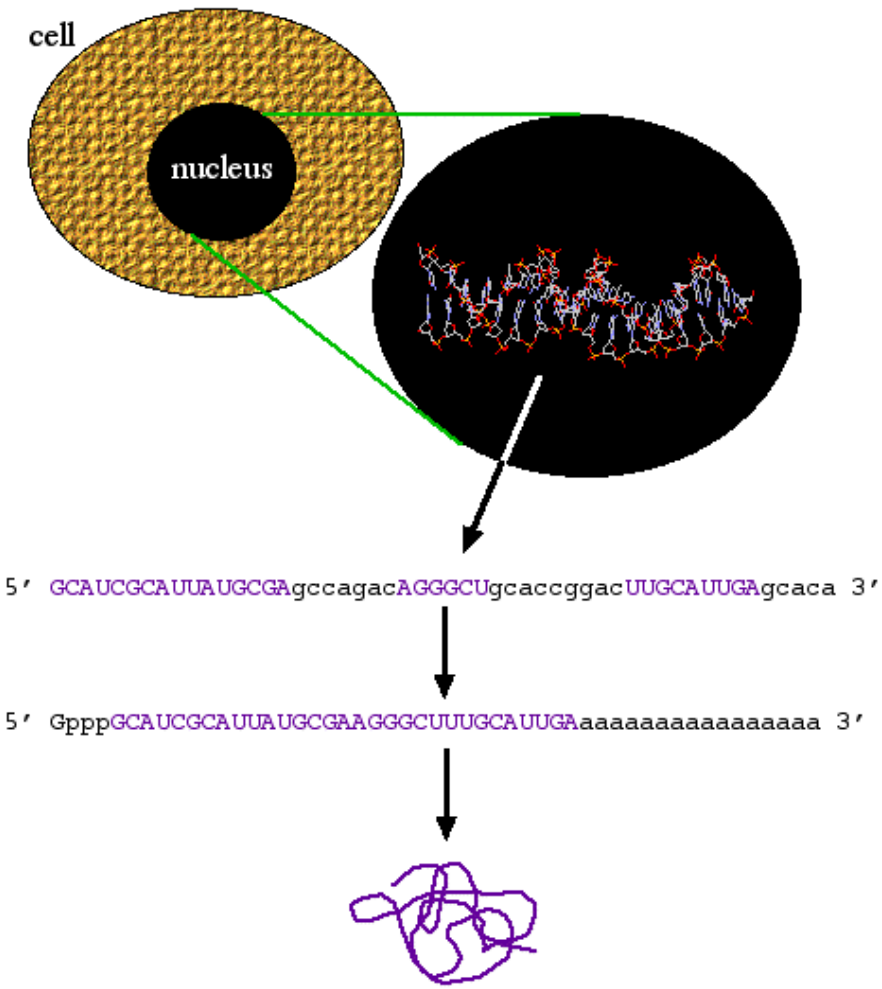
Ganz schlimmer Worst case; eigentlich eher O(k*n)

Biologische Motivation

- Beispiel: Gene Finding
- Gegeben ein Genom ...
- Wie findet man die Gene in einer Sequenz?
  - Der Sequenz kann man es nicht ansehen...
  - Vergleich mit ähnlichen Genen anderer Spezies
  - Umweg über ESTs

DNA -> Protein

- Central Dogma
  - DNA
  - RNA
  - Protein
- RNA editing
  - 5‘ CAP
  - 3‘ PolyA Tail
  - Splicing
- ➢ messenger RNA (mRNA)



mRNA - cDNA

- Trick: Rück-Übersetzung von mRNA in cDNA
  - Reverse Transcriptase (RT)
  - Primer z.B. am Poly-A Tale (oder zufällig)
- Clonierung der cDNA in Bibliotheken

cDNA Libraries

- cDNAs: Gene / kodierende Regionen
  - Viel interessanter als genomische DNA

- Beachte
  - Differential Splicing – different cDNAs
  - Inhalt einer cDNA Library hochgradig abhängig von
    - Gewebe
    - Entwicklungsstadium (Embryo - Erwachsen)
    - Organismusstatus (Krank – Gesund)
  - Aber: Jede cDNA entspricht einem Gen, und das muss irgendwo im Genom liegen

cDNA -> EST

- EST: Expressed Sequence Tags
- Single Read Sequenzierung der cDNA
  - 3‘ Enden oder
  - 5‘ Enden
- Sehr populär (Stand 4/2003)
  - 8 von 30GB in Genbank
  - 16 von 23 Millionen Submissions in Genbank
  - Ca. 4.500.000 humane ESTs
- Idee
  - Ansequenzierung die cDNA ergibt EST
  - Suchen der EST Sequenz im Genom
  - Damit kennt man einen Teil eines Gens
  - Vervollständigen durch weitere ESTs (Clustering)

Motivation II: Datenbanksuche

- Gilt nicht nur für ESTs
  - Suche in bekannten Sequenzdatenbanken nach neuen Sequenzen ist eines der Hauptthemen der Bioinformatik
  - I.d.R. sucht man nicht nach exakten Vorkommen – approximative Suche – später
- Weitere Anwendungen von Suffixbäumen
  - Suche nach exakten „Seeds" zur approximativen Suche
    - Später mehr
  - Suche nach längsten identischen Subsequenzen
    - Vergleich zweier Genome
  - Suche nach längsten Repeats
    - Finden von typischen, sich im Genom wiederholenden Sequenzen
  - ...

Weiteres Vorgehen

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

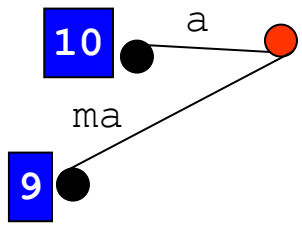
- Ab jetzt: Wir bauen einen Suffixbaum T für String S mit |S|=m

Suffixbäume

- Intuition: Kompakte Repräsentation aller Suffixe von S in einem Baum
- Definition: *Suffixbaum T für S ist ein Baum mit*
  - *T hat m Blätter, markiert mit 1,...,m*
  - *Jede Kante E ist mit einem Substring label(E)≠∅ von S beschriftet*
  - *Jeder innere Knoten k hat mindestens 2 Kinder*
  - *Alle Label der Kanten von einem Knoten k aus beginnen mit unterschiedlichen Zeichen*
  - *Sei $\{k_1,k_2,...,k_n\}$ ein Pfad von der Wurzel zu einem Blatt mit Markierung i. Dann ist die Konkatenation der Label der Kanten auf dem Pfad gleich S[i..m]*
    - *Also das Suffix von S, das an Position i startet*
- Beachte
  - Bei Suffixbäumen stehen Substrings an Kanten, bei Keyword Trees einzelne Zeichen

Beispiel 1

```
      1234567890
• S=  BANANARAMA
```

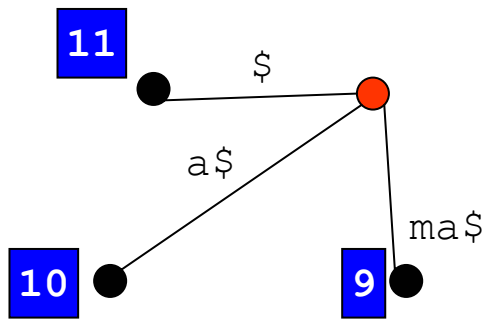


- Problem: Wohin kommt „AMA“?
  - Verlängerung von „a“ verboten – 10 sonst kein Blatt
  - Neue Kante „ama“ verboten – zwei Pfade aus der Wurzel würden sonst mit gleichem Zeichen beginnen
  - ➢ Es gibt keinen Suffixbaum für BANANARAMA
  - ➢ Problem tritt auf, sobald ein Suffix Präfix eines anderen Suffix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

Beispiel 2

```
    12345678901
```

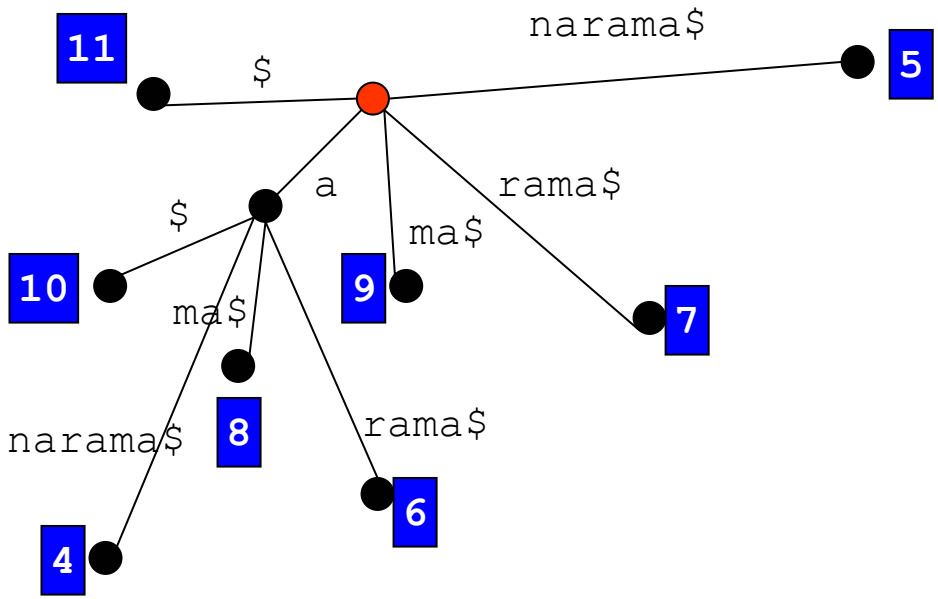
- S= BANANARAMA$



Beispiel 2

```
    12345678901
```

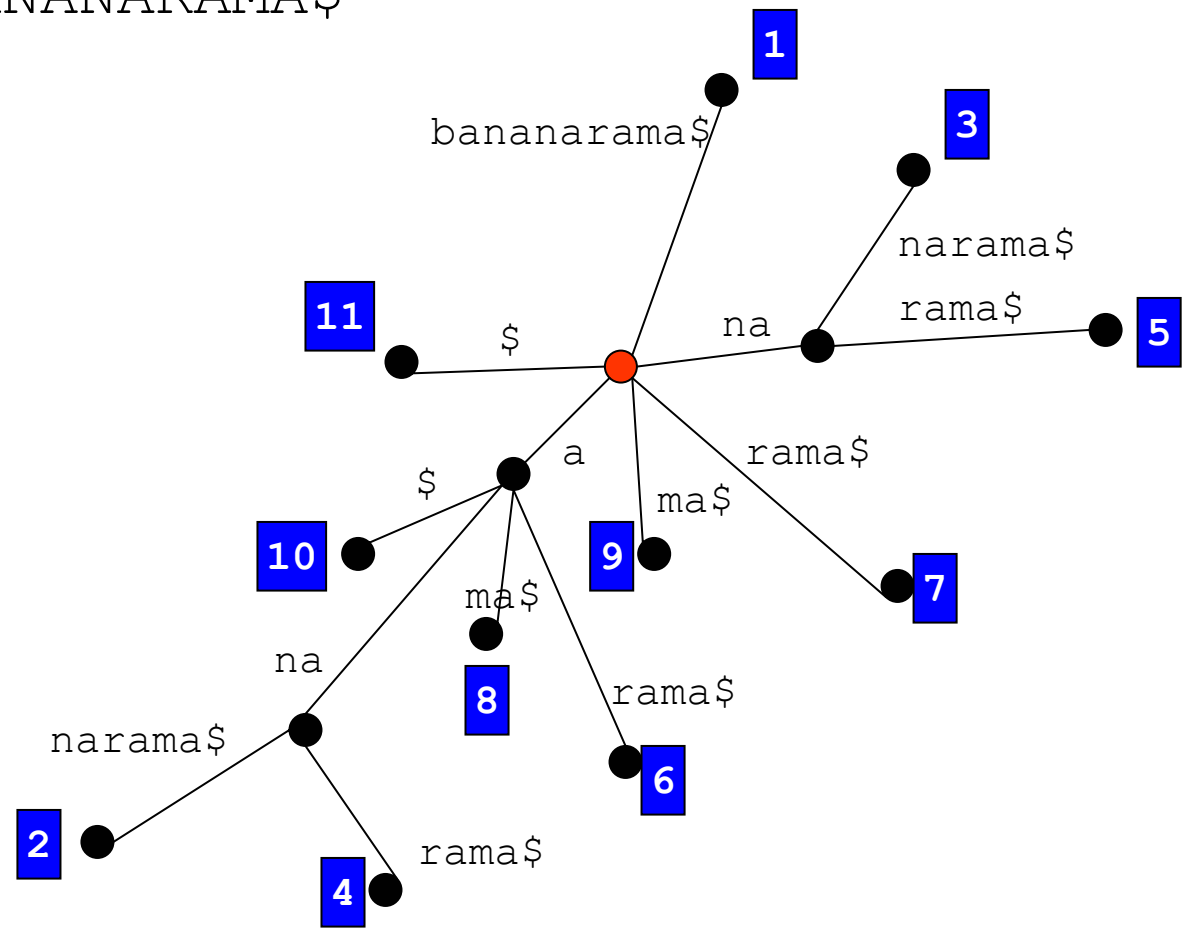
- S= BANANARAMA$



Beispiel 3

12345678901

- S= BANANARAMA$

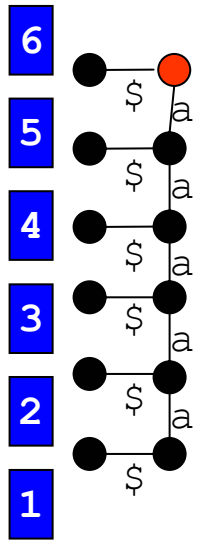


Eigenschaften von Suffixbäumen

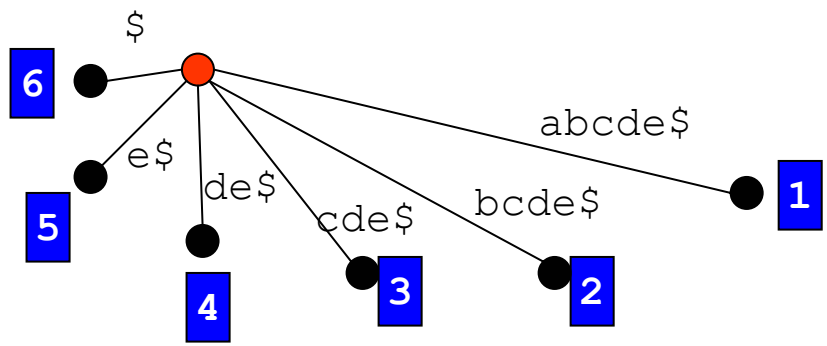
- Zu jedem String gibt es genau einen Suffixbaum
- Jeder Pfad von der Wurzel zu einem Blatt ist unterschiedlich
- Jede Verzweigung an einem inneren Knoten ist eindeutig bzgl. des nächsten Zeichens auf dem Pfad
- Gleiche Substrings können an mehreren Kanten stehen
- Suffixbäume und Keyword-Trees
  - Betrachte alle Suffixe von S als Pattern
  - Konstruiere den Keyword-Tree
  - Verschmelze alle Knoten auf einem Pfad ohne Abzweigungen zu einer Kante
  - Dann haben wir einen Suffixbaum für S
  - Komplexität?

Weitere Beispiele

- S= aaaaaa$



- S= abcde$



Definitionen

- Sei T der Suffixbaum für S
  - *Sei p ein Pfad in T von root(T) zu einem Knoten k. Dann ist label(p) (path label) die Konkatenation der Label der Kanten auf dem Pfad p*
  - *Sei k ein Knoten von T und p der Pfad zu k. Dann ist label(k) (node label) = label(p)*
  - *Sei k ein Knoten von T. Dann ist depth(k)=|label(k)|*

Anwendungen

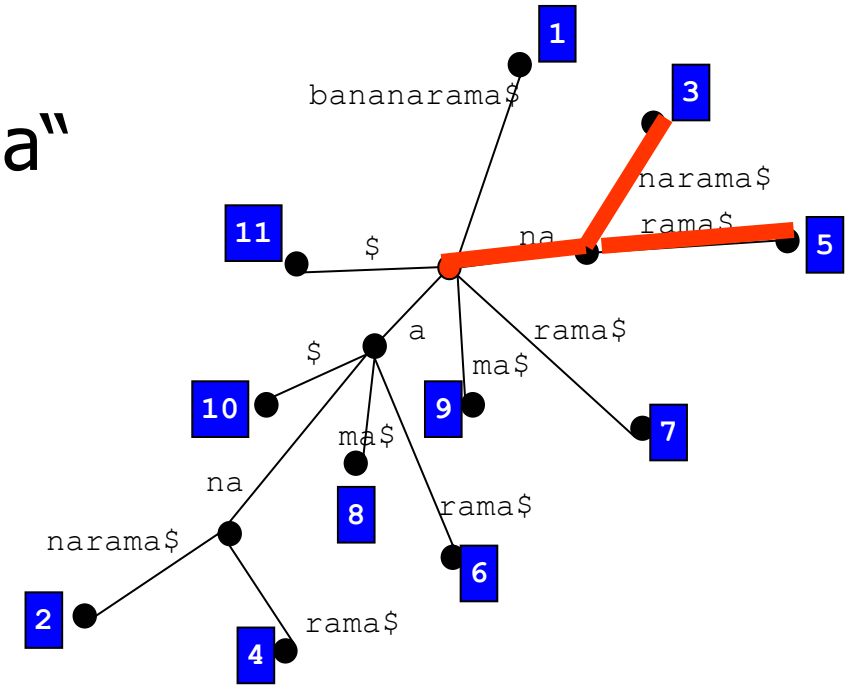
- Suche eines Pattern P
- Längster gemeinsamer Substring zweier Strings
- Längstes Palindrom

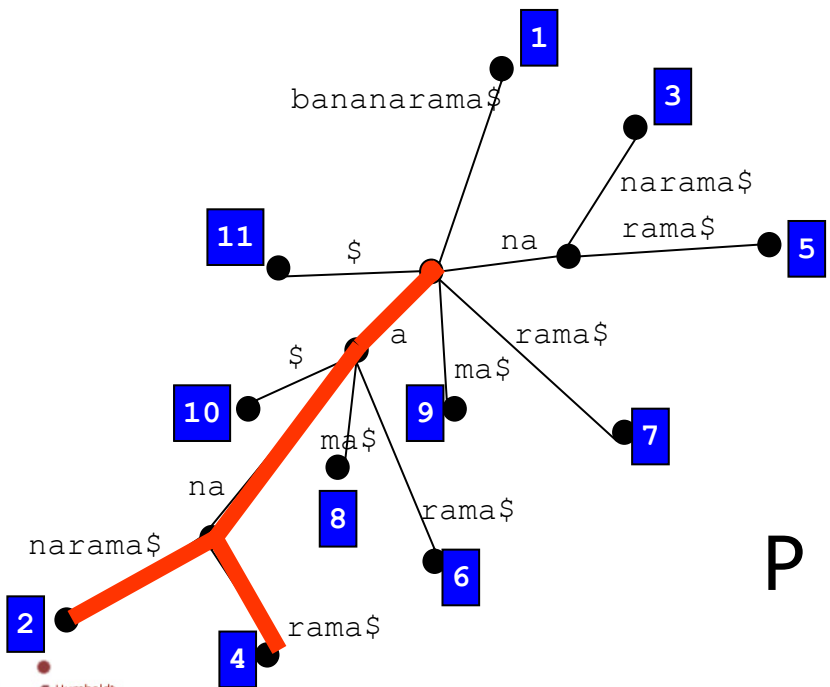
Suche mit Suffixbäumen

- Intuition
  - Jedes Vorkommen von P muss Präfix eines Suffix sein
  - Und die haben wir alle auf Pfaden von der Wurzel aus
- Gegeben S und 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 T ab der Wurzel
    - Wenn das nicht geht, kommt P in S nicht vor
    - P kann in einem Knoten k enden; merke k
    - Oder P endet in einem Kantenlabel; sei k der Endknoten dieser Kante
  - Die Markierungen aller unterhalb von k gelegenen Blätter sind Startpunkte von Vorkommen von P in S

Beispiel: bananarama$

P = „na“





P = „an“

Komplexität

- Theorem
  Sei T ein Suffixbaum für S+"$". Die Suche nach allen Vorkommen eines Pattern P, |P|=n, in S ist O(n+j), wenn j die Anzahl Vorkommen von P in S ist.

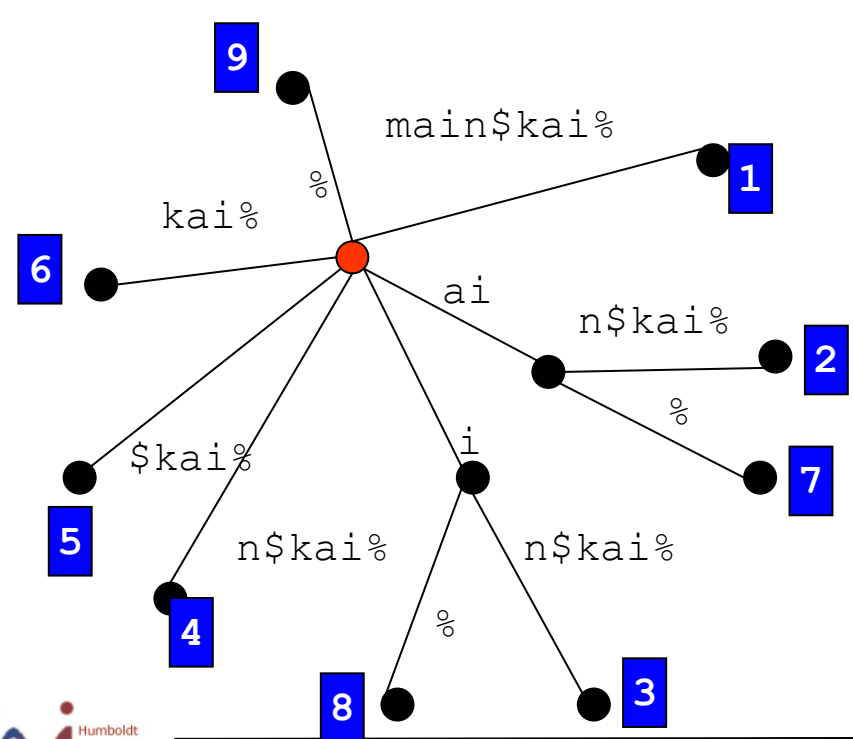
- Beweisidee
  - P in T matchen kostet nur O(n)
    - Pfade sind eindeutig – Entscheidung an jedem Knoten ist klar
    - Damit maximal O(n) Zeichenvergleiche
  - Blätter aufsammeln ist O(j)
    - Baum unterhalb Knoten K hat j Blätter
    - Die finden wir per Depth-First Suche
- Suche ist damit schlimmstenfalls O(n+m)
  - Aber das ist ein echter Worst case

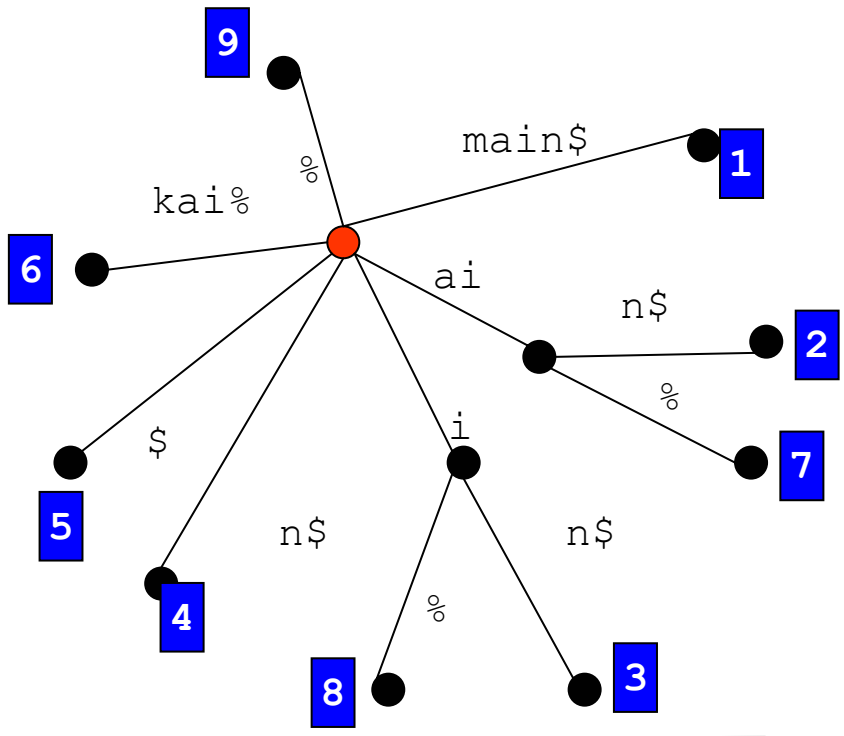
Längster gemeinsamer Substring

- Gegeben zwei Strings S_1 und S_2
- Gesucht: Längster gemeinsamer Substring s
- Vorschläge ?
- Lösung
  - Konstruiere Suffixbaum T für $S_1\$S_2\%$
  - Streiche aus diesem Baum alle Pfade unterhalb eines „$"
  - Durchlaufe den Baum
    - markiere alle internen Knoten mit 1, wenn im Baum darunter ein Blatt aus S_1 kommt
    - markiere Knoten mit 2, wenn ... Blatt aus S_2 vorkommt
  - Suche den tiefsten Knoten mit Beschriftung 1 und 2

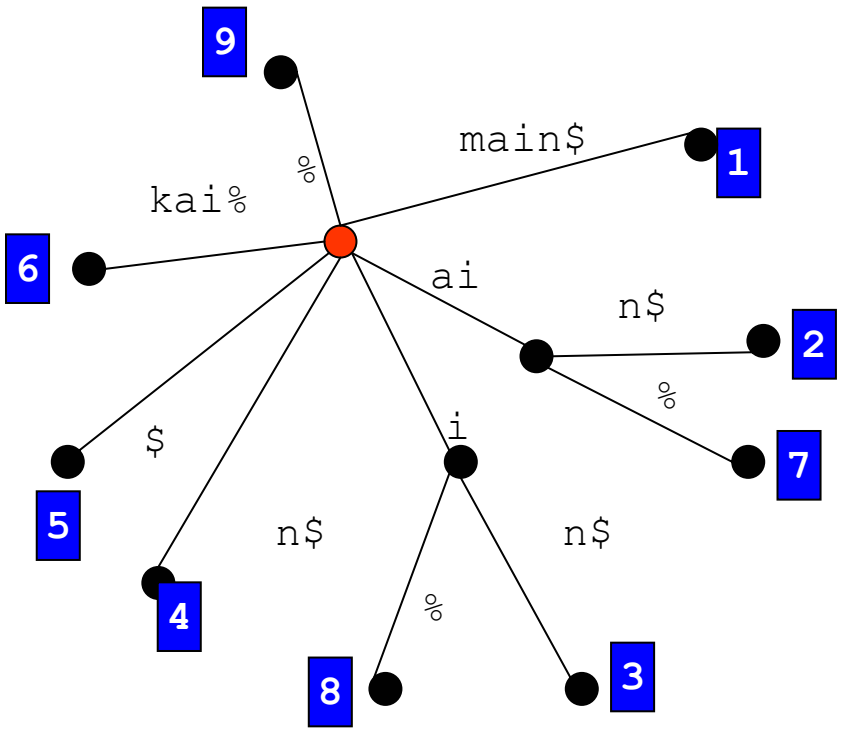
Beispiel

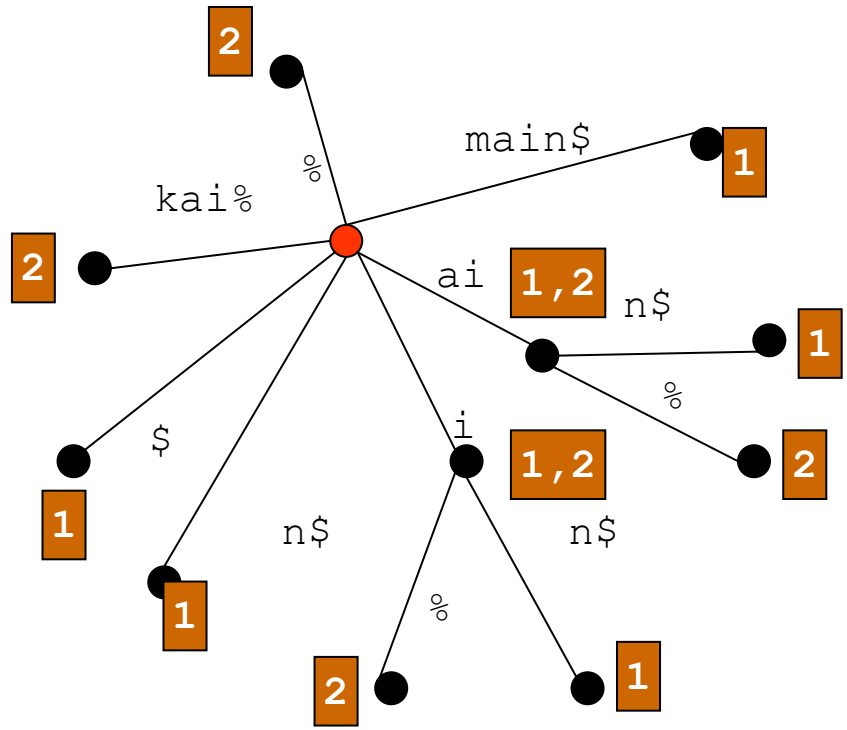
- S_1=main$, S_2=kai%
- ...





Beispiel





- Verallgemeinerbar zu n Strings $S_1,\ldots,S_n$

Komplexität

- Annahme: Wir können T für S in O(|S|) berechnen
- Die Schritte
  - Sei m=$|S_1|+|S_2|$
  - Konstruiere Suffixbaum T für $S_1\$S_2\%$
    - Ist O(m) nach Annahme
  - Streiche aus diesem Baum alle Pfade unterhalb eines „$“
    - Depth-First Traversal – O(m)
  - Durchlaufe den Baum und markiere innere Knoten mit 1,n
    - Depth-First Traversal – O(m)
  - Suche den tiefsten Knoten mit Beschriftung 1 und 2
    - Breadth-First Traversal – O(m)
- Zusammen: O(m)

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$
  - Füge Suffixe von String reverse(S)% ein
  - Suche längsten gemeinsamen Substring in T

Naive Konstruktion von Suffixbäumen

- Gegeben: String S. Gesucht: Suffixbaum T für S
- Start
  - Bilde Baum T_0 mit Wurzelknoten und einer Kante mit Label „S$" zu einem Blatt mit Markierung 1
- Konstruiere T_{i+1} aus T_i wie folgt
  - Betrachte das Suffix S_{i+1} = S[i+1..]
  - Matche S_{i+1} in T_i so weit wie möglich
  1. Wenn S_{i+1} auf einer Kante n an Position j des Labels aufgebraucht ist
     - Füge in n an Position j einen neuen Knoten k ein
     - Erzeuge eine Kante von k zu einem neuen Blatt k'; beschrifte die Kante mit „$"
     - Markiere k' mit i+1

Naive Konstruktion von Suffixbäumen 2

- Konstruiere T_{i+1} aus T_i wie folgt ...
  2. Wenn S_{i+1} genau am Ende einer Kante n aufgebraucht ist
     - Sei k der Zielknoten von n
     - Erzeuge eine Kante von k zu einem neuen Blatt k‘; beschrifte die Kante mit „$“
     - Markiere k‘ mit i+1
  3. Wenn S_{i+1} auf einer Kante n an Position j des Labels nicht mehr matched (der Mismatch in S_{i+1} sei an Position j‘),
     - Füge in n an Position j einen neuen Knoten k ein
     - Erzeuge eine Kante von k zu einem neuen Blatt k‘; beschrifte die Kante mit „S[j‘..]$“
     - Markiere k‘ mit i+1

Beispiel

- „barbapapa“
- ...

Komplexität

- Komplexität
  - Jeder Schritt von T_i zu T_{i+1} ist O(m)
  - Es gibt m-1 solche Schritte
  - Zusammen: $O(m^2)$
- Nächstes Thema
  - O(m) Algorithmus von Ukkonen

Zusammenfassung

- Bisherige Algorithmen sind exzellent für veränderliche Texte (Textverarbeitung etc.)
- Für Datenbanksuchen (Feste T, dauernd andere P) sind Suffixbäume besser
- Naive Konstruktion ist zu teuer
- Was fehlt?
  - Besserer Konstruktionsalgorithmus
  - Strukturen mit weniger Speicherverbrauch
  - Suffixbäume auf Sekundärspeichern

Schöne Wörter

- Rokokomode
- Rokokokokolores
- Rokokokokosnuss
- Panamakanal