

Algorithmische Bioinformatik

Suffixbäume

Ulf Leser

Wissensmanagement in der
Bioinformatik



Konstruktion der Failure Links

- Bisher
 - Mit Failure Links ist die Suchphase $O(m)$
 - Wir ignorieren erstmal das Übersehens-Problem
 - Konstruktion des Keyword Trees ist $O(n)$
 - Wie lange braucht man, um Failure Links zu berechnen?
- Definition
 - Sei *depth(k)* die Tiefe des Knoten k (Abstand zu root)
- Vorgehen
 - 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., für alle k gilt: $\text{depth}(k) > \text{depth}(\text{fl}(k))$
 - Wir konstruieren Failure Links per Breitensuche

Algorithmusidee 2

- Induktionsschritt von $i-1$ zu i
 - Seien alle Failure Links von Knoten l mit $\text{depth}(l) < i$, bekannt
 - $\forall k \in K$ mit $\text{depth}(k) = i$
 - Sei k' der Vater von k und x stehe auf der Kante (k', k)
 - Folge dem Failure Link von k' aus zu $\text{fl}(k') = v$
 - Wenn es eine Kante (v, v') mit Label x gibt: $\text{fl}(k) = v'$
 - Wenn nicht
 - Wenn $v = \text{root}(K)$, dann $\text{fl}(k) = \text{root}$
 - Sonst: folge Kante $\text{fl}(v) = v''$, suche eine ausgehende Kante mit x ...
 - Wiederhole rekursiv

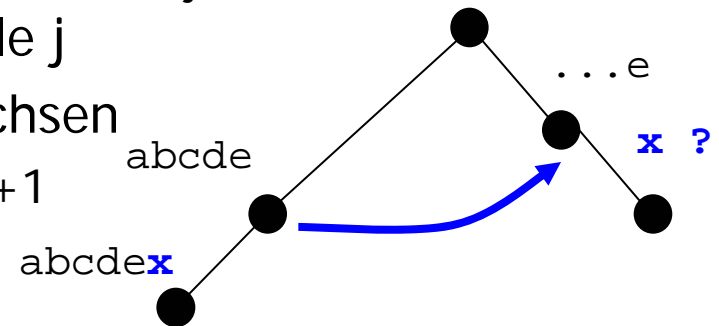
Algorithmus

```
// We search failure link for k, depth(v)>1
// Let k' be the father of k, label(k',k)=x
v := fl(k');
while (v≠root(K)) and (not exists edge (v,v') with label(v,v')=x)
    v = fl(v);           // Follow failure link
end while;
if (v=root(K)) then
    if (exists edge (v,v') with label(v,v')=x)
        fl(k) = v';
    else
        fl(k) = root(K);
else
    fl(k) = v';           // Continuation of prefix with x
```

- Komplexität?

Beweis

- Betrachten wir ein P_i , $t = |P_i|$, mit Knoten v_1, \dots, v_t
- Betrachten wir $\text{length}(v_j)$ eines Knoten v_j
 - $\text{length}(v_1) = 0$, und $\text{length}(v_j) \geq 0$ für alle j
 - $\text{length}(v_j)$ kann mit steigendem j wachsen
 - Es gilt immer: $\text{length}(v_j) \leq \text{length}(v_{j-1}) + 1$

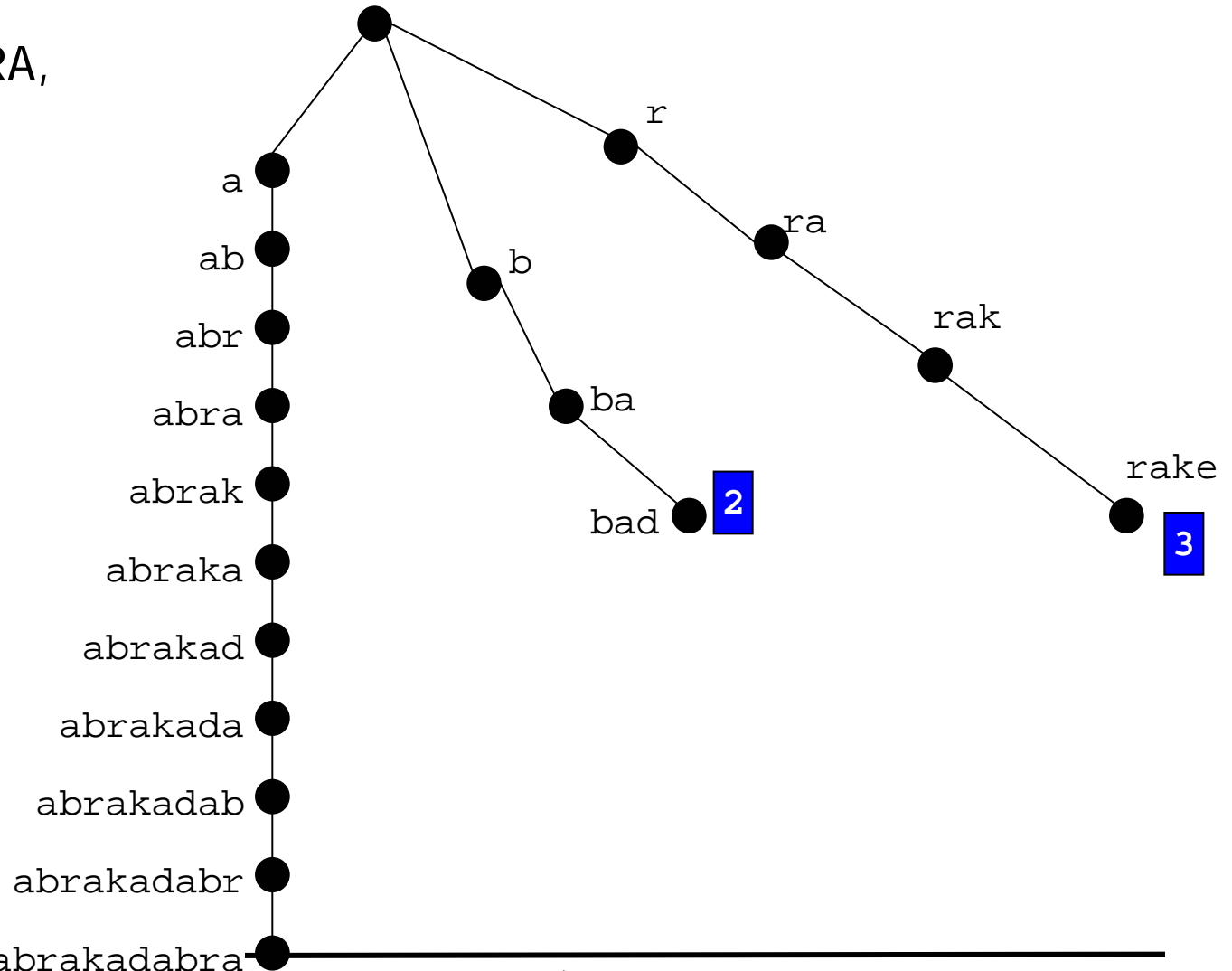


- $\text{length}(v_j)$ wird also höchstens t mal um 1 größer über alle Knoten v_j
- $\text{length}(v_j)$ schrumpft auch mit steigendem j
 - Bei jedem Sprung in der WHILE Schleife um mindestens 1
- Zusammen
 - Mit jedem Sprung 1 weniger, nie kleiner 0, insgesamt nur t mal 1 mehr
 - Also kann es maximal t Sprünge geben

• qed.

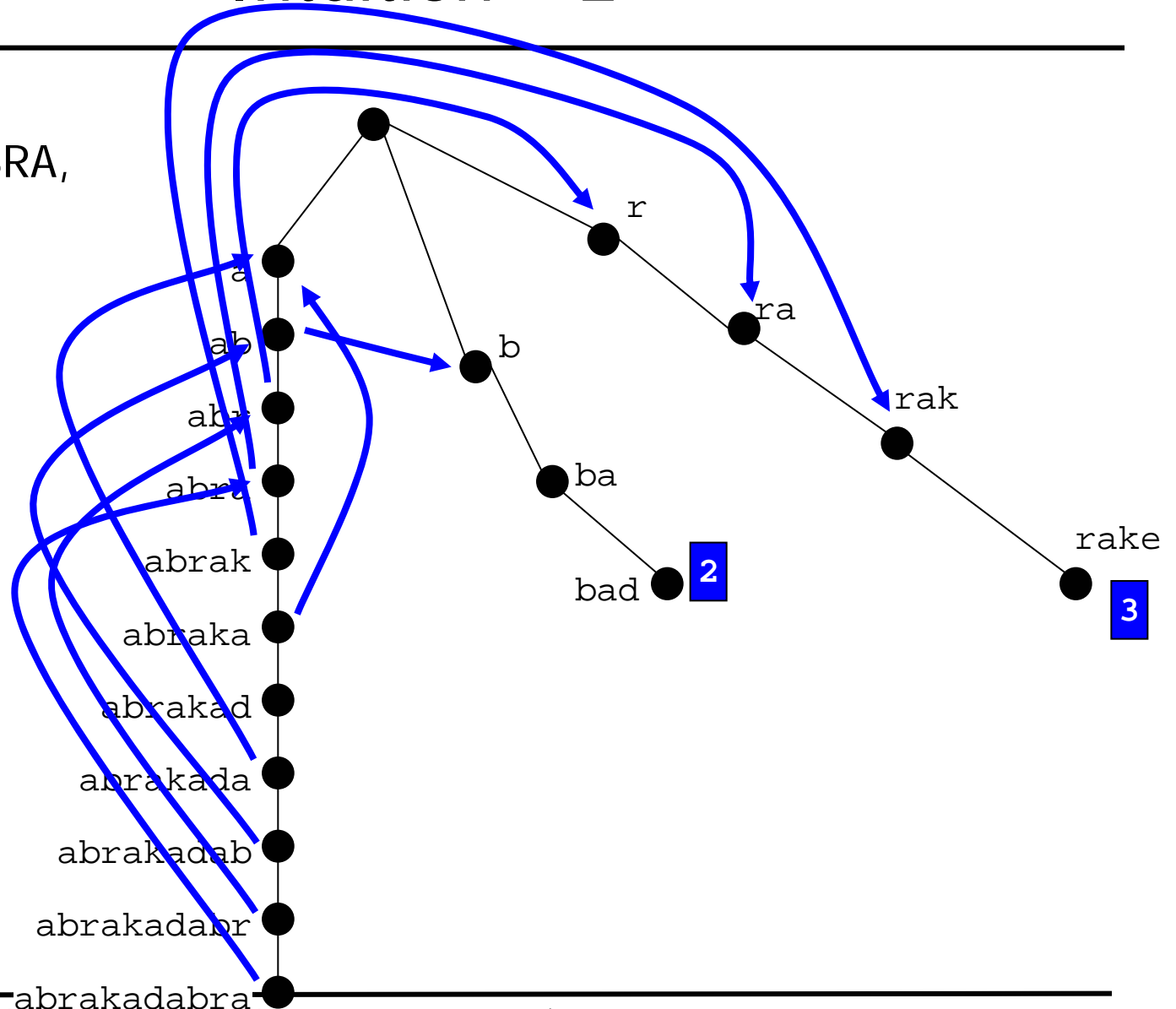
Intuition – Ist das wirklich linear?

- $P = \{ \text{ABRAKADABRA, BAD, RAKE} \}$



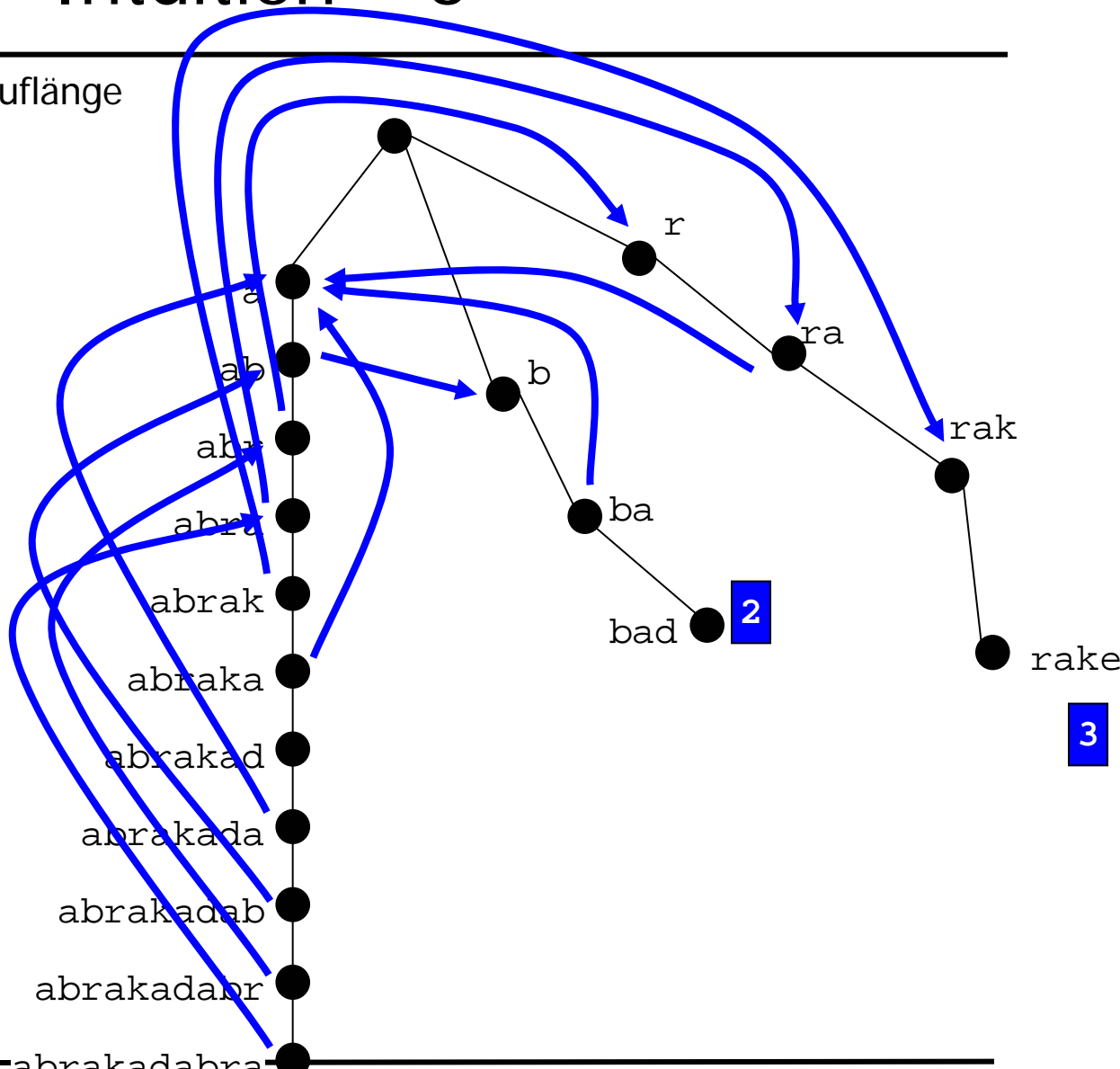
Intuition – 2

- $P = \{ \text{ABRAKADABRA, BAD, RAKE} \}$



Intuition – 3

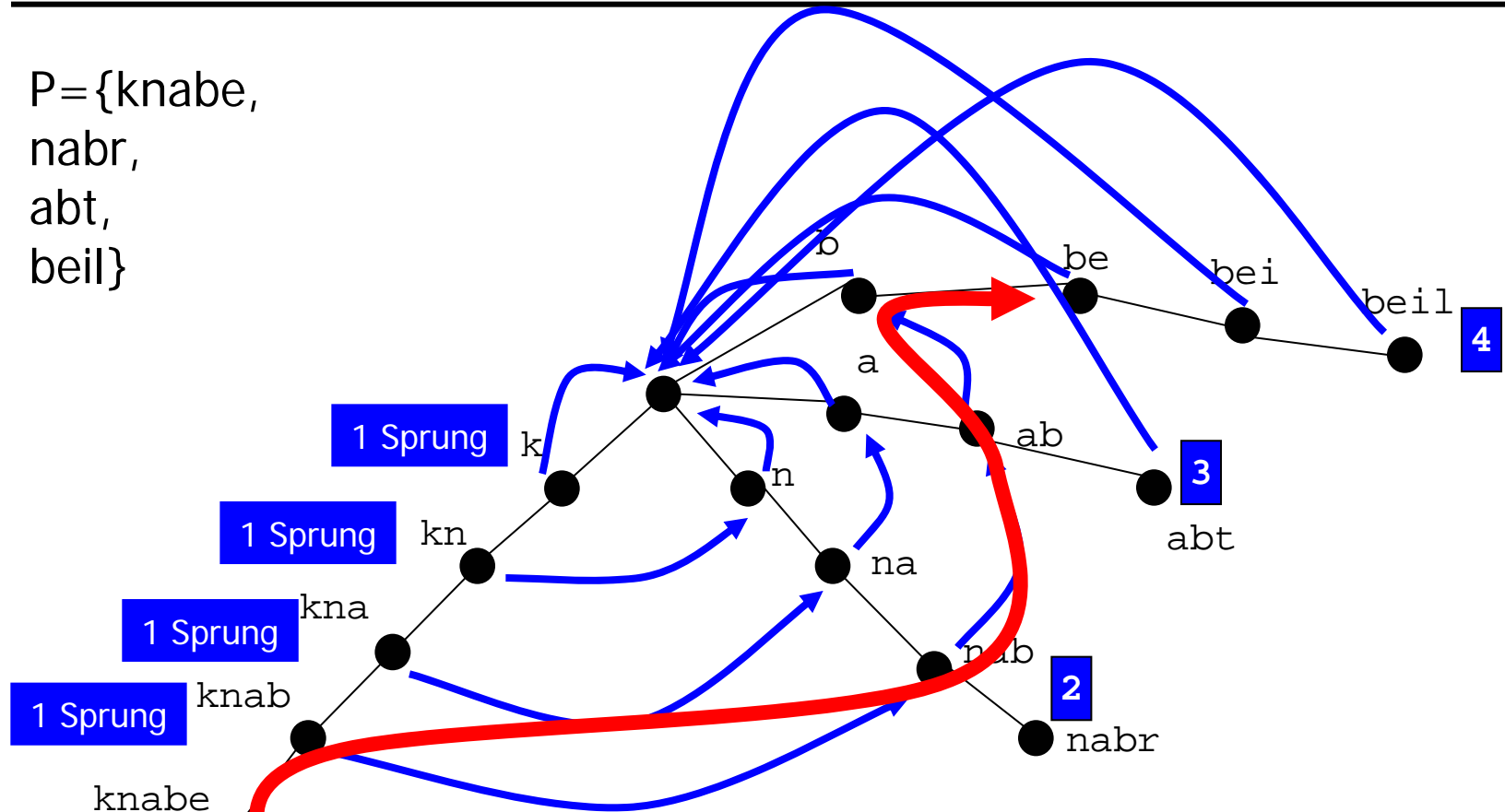
Präfix	length(k)	Max Suffixlänge
- A	0	1
- AB1	2	2
- ABR	1	2
- ABRA	2	3
- ABRAK	3	4
- ABRAKA	1	2
- ABRAKAD	0	1
- ABRAKADA	1	2
- ABRAKADAB	2	3
- ...AKADABR	3	4
- ...AKADABRA	4	5



- Also: Maximale Suffixe der Zukunft werden kürzer, wenn man vorher zu kurzen Suffixen springt (evt. über mehrere Sprünge)

Mehrere Sprünge

$P = \{ \text{knabe, nabr, abt, beil} \}$



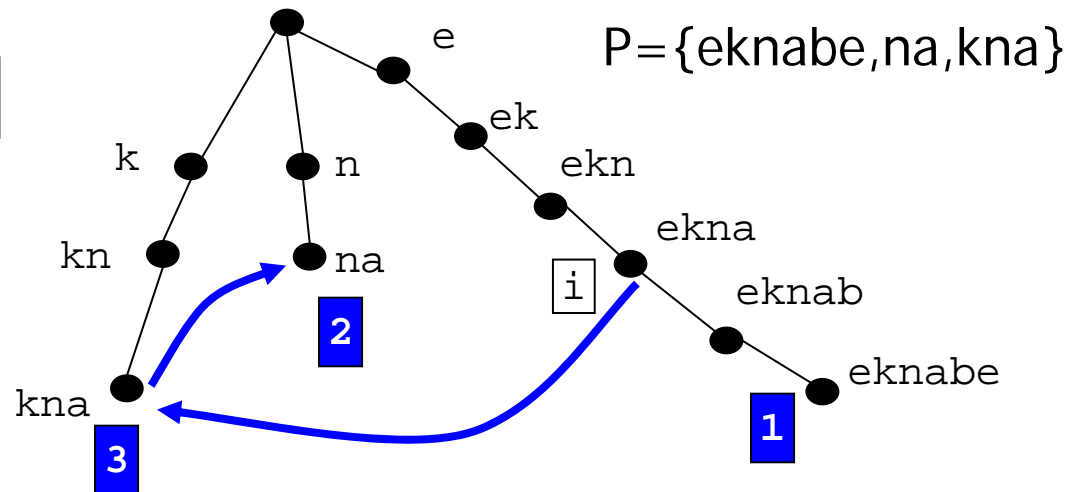
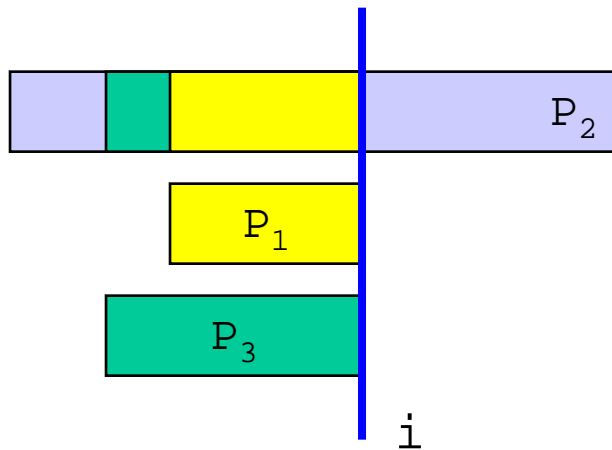
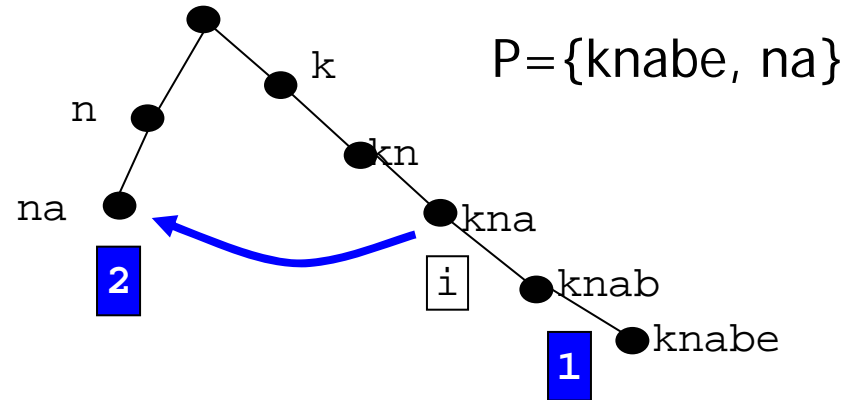
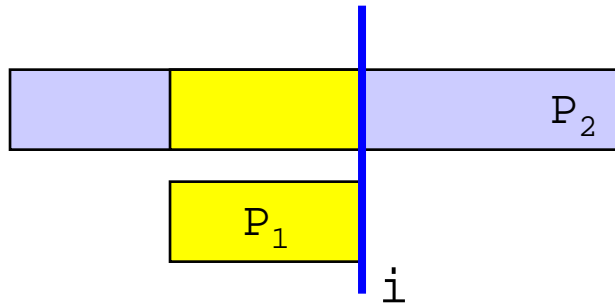
3 Sprünge

- Maximale Suffixe können in Zukunft nur noch $n-3$ lang sein
- Für einen FL kann man deshalb **auch in Zukunft nur noch max $n-3$ Sprünge** machen
- Gesamtzahl $O(n)$ wird eingehalten

Spezialfall

- Problem: Pattern, die andere Pattern enthalten
 - Muss kein Präfix sein
- Lösung: **Output Links**
 - Wir konstruieren noch einen Pointer für (einige) Knoten in K
- Beobachtung über die Problemfälle
 - Sei P_1 in P_2 enthalten (also unser Problemfall)
 - Dann muss P_1 **Suffix vom Präfix $P_2[1..i]$** für irgendein $i \geq |P_1|$ sein
 - Wenn P_1 das längste echte Suffix von $P_2[1..i]$ ist, dann gilt $fl(P_2[i]) = P_1$
 - Was nichts heisst; bei der Suche müssen wir diesem fl nicht folgen
 - Wenn das nicht gilt, gibt es ein P' mit
 - P' ist längstes Suffix von $P_2[1..i]$
 - Also gilt $fl(P_2[i]) = P'$
 - Wiederum gilt: P_1 ist Suffix von P' – ist es auch das längste?
 - **Suche rekursiv über Failure Links**
 - **Schließlich muss man bei P_1 ankommen**

Beispiel



Suchphase mit Output Links

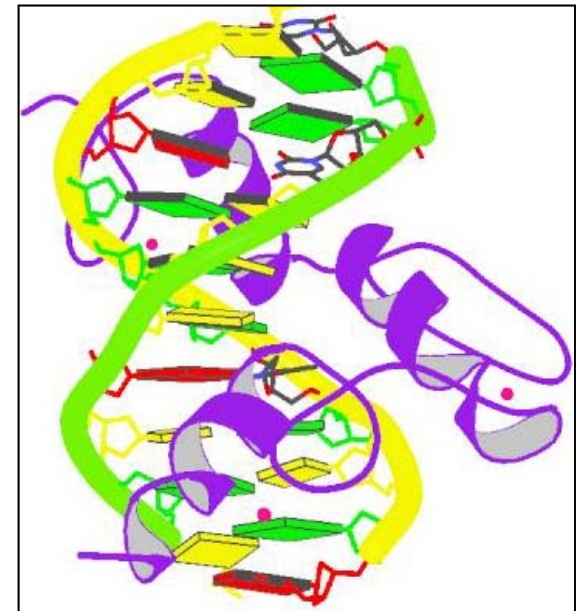
- Man muss bei jedem Knoten k , den man ablauft, nachsehen, ob es einen Output Link gibt
- Wenn ja, beschreite einen Nebenweg
 - Sei $v = \text{out}(k)$
 - Gib $\text{mark}(v)$ aus
 - Der Zielknoten muss markiert sein
 - Wenn vorhanden, folge $\text{out}(v)$ rekursiv
- Danach bei k weitermachen

Komplexität

- Komplexität der Suchphase
 - Sei k die **Gesamtzahl an Matches von Pattern aus P in T**
 - Die innere WHILE Schleife wird maximal k -mal passiert
 - **Also: $O(m+k)$**
- Gesamtkomplexität
 - 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 $O(n+m+k)$**

Suche mit Wildcards

- Nur ein Pattern P, aber P darf **Wildcards** enthalten
 - „*“ steht für exakt ein beliebiges Zeichen
 - Wir begrenzen die Anzahl von „*“ auf s
- Beispiel
 - Zinc Finger Domain
 - C**C*****H**H
 - Typisches Motiv für DNA/RNA bindende Proteine
 - Interpro IPR007087, PDB 1A1F



Algorithmus

- Clevere Verwendung von Aho-Corasick
 - Gegeben: Pattern P , Template T
 - Initialisiere Integerarray $C = [0, 0, 0, \dots, 0]$, mit $|C| = |T|$
 - $P' = \{P_1, \dots, P_s\}$ sei die **Multimenge** aller maximalen Substrings in P ohne Wildcards und l_1, \dots, l_s seien ihre Startpositionen
 - Berechne Keyword Tree für P' 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 $C[x] = s$** repräsentiert ein Vorkommen von P in T an Position x
 - Alle s Subpattern P_i wurden an den richtigen Stellen gefunden

Die Grenzen des linearen Stringmatchen

- Wir können in linearer Zeit
 - Alle Vorkommen eines Pattern P in einem Template T finden
 - Alle Vorkommen einer Menge von Pattern in einem Template T finden
 - Alle Vorkommen eines Pattern P mit Wildcard in einem Template T finden
 - Alle Vorkommen eines Pattern P mit maximal k Mismatches in T finden
 - Zeigen wir nicht
- Was können wir nicht mehr in linearer Zeit?
 - Alle Vorkommen eines **regulären Ausdrucks R** in T finden
 - Alle **approximativen Vorkommen** eines Pattern P in T finden

Inhalt dieser Vorlesung

- Suffixbäume
- Verwendung von Suffixbäumen
- Naive Konstruktion

Wo sind wir?

- Alle exakten Vorkommen von P in T
- Alle exakten Vorkommen einer Menge von P in T
- **Datenbankformulierung: Alle exakten Vorkommen von P in T, aber man darf T prä-prozessieren**
- Approximatives Stringmatching
- Datenbankformulierung: Finden ähnlichster Sequenzen
- Heuristiken für approximatives Stringmatching
- Multiple Sequence Alignment
- Phylogenetische Algorithmen

Problemstellung

- Bisherige Algorithmen
 - Ein Template T (m) und ein oder mehrere Pattern P (n)
 - Prinzip: 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 darf vorverarbeitet werden
 - Kosten amortisieren sich über viele Suchen
 - Zählen nicht für die Suche eines Pattern
- Lösung: Suffixbäume

Motivation: Datenbanksuchen

- Suche in bekannten Sequenzdatenbanken nach neuen Sequenzen ist eines der Hauptthemen der Bioinformatik
 - I.d.R. sucht man nicht nach exakten Vorkommen, sondern approximativ (später)
 - Aber: Schnelle Heuristiken für approximatives Suchen benutzen (fast) immer ein exaktes Suchverfahren **zum Finden aussichtsreicher Regionen**
- Für exakte Suche sind **Suffixbäume die schnellste Datenstruktur**
 - Aber: Speicherplatz, Sekundärspeicherverhalten
 - Alternative: Suffix-Arrays, Enhanced Suffix-Arrays
- Es gibt viele weitere Anwendungen von Suffixbäumen
 - Vorstufe des approximativen Suchens: Suche nach „Seeds“
 - Suche nach längsten identischen Subsequenzen
 - Vergleich zweier Genome
 - Suche nach längsten Repeats
 - Finden von 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

- Idee: Kompakte Repräsentation **aller Suffixe** von S in einem Baum
- Definition

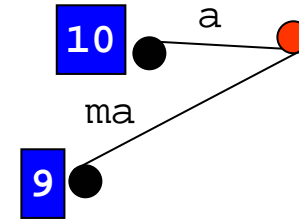
*Der **Suffixbaum** T für einen String S ist ein Baum mit*

 - *T hat eine Wurzel und m Blätter, markiert mit $1, \dots, m$*
 - *Jede Kante E ist mit einem **Substring** $\text{label}(E) \neq \emptyset$ 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, \dots, 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** $S[i..m]$*
- Suffixbäume versus Keyword-Trees
 - Substrings oder einzelne Zeichen als Kantenlabel
 - Indexierung mehrerer anhängiger Suffixe oder mehrerer unabhängiger Pattern

Beispiel 1

1 2 3 4 5 6 7 8 9 0

- S = BANANARAMA

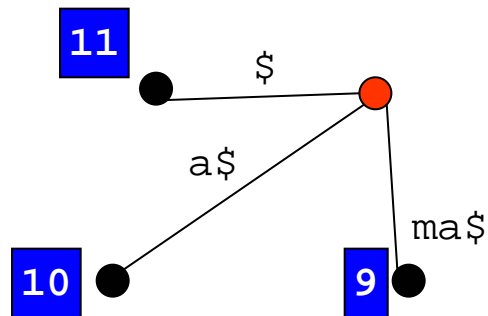


- Problem: Wohin kommt „AMA“?
 - Verlängerung von „a“ verboten – 10 sonst kein Blatt
 - Neue Kante „ama“ verboten – zwei Pfade aus der Wurzel würden sonst mit gleichem Zeichen beginnen
 - Es gibt **keinen Suffixbaum** für BANANARAMA
 - Problem tritt auf, sobald ein Suffix Präfix eines anderen Suffix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

Beispiel 2

12345678901

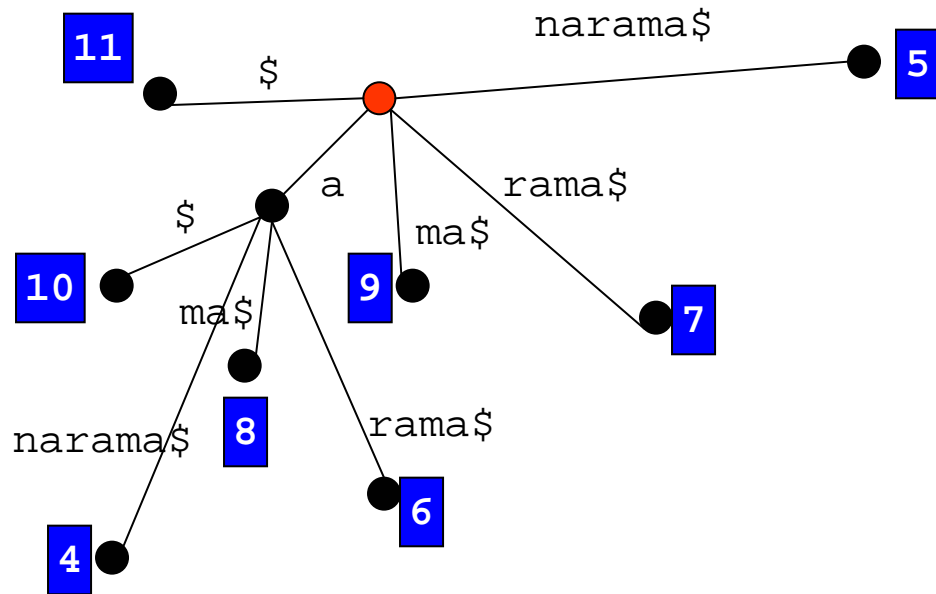
- $S = \text{BANANARAMA\$}$



Beispiel 2

12345678901

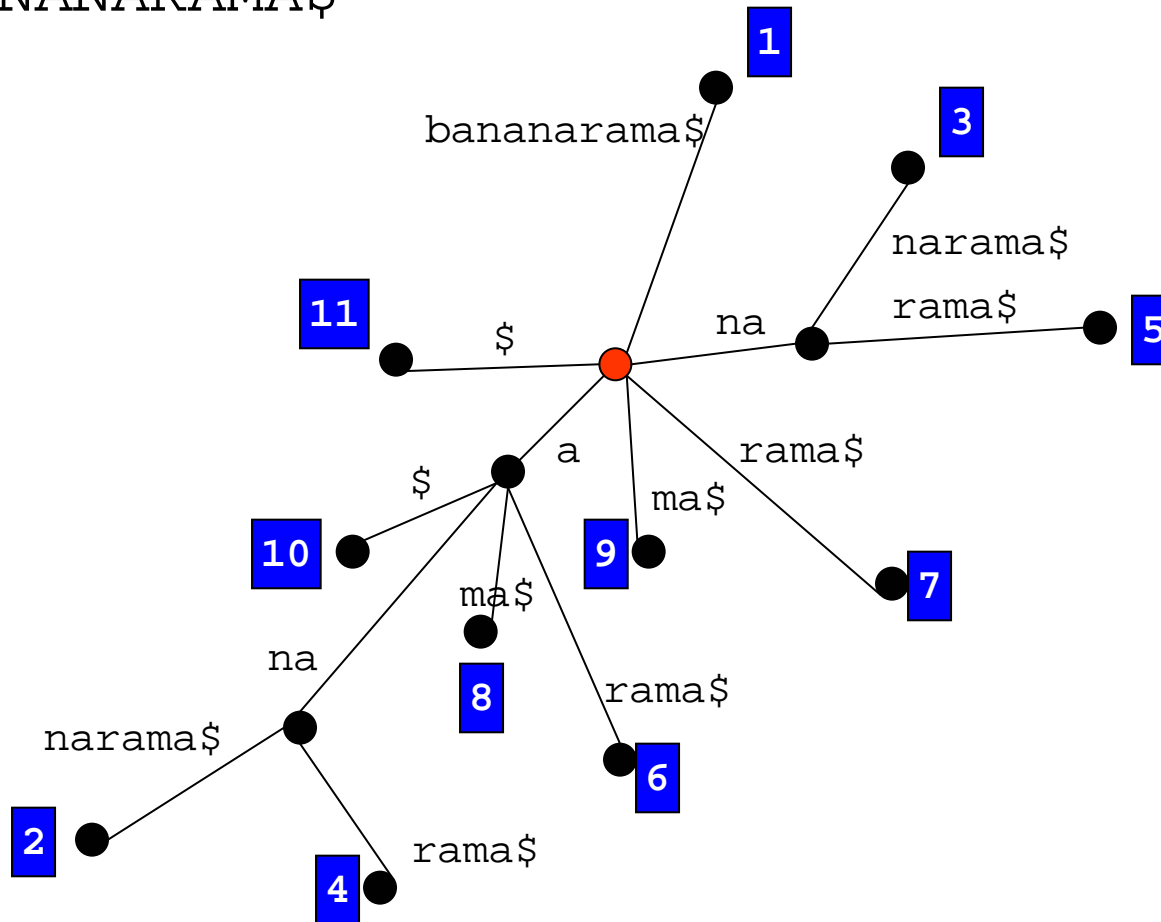
- S= BANANARAMA\$



Beispiel 3

12345678901

- S= BANANARAMA\$

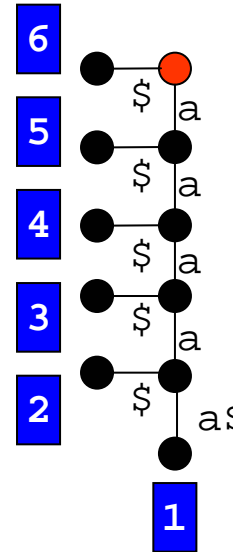


Eigenschaften von Suffixbäumen

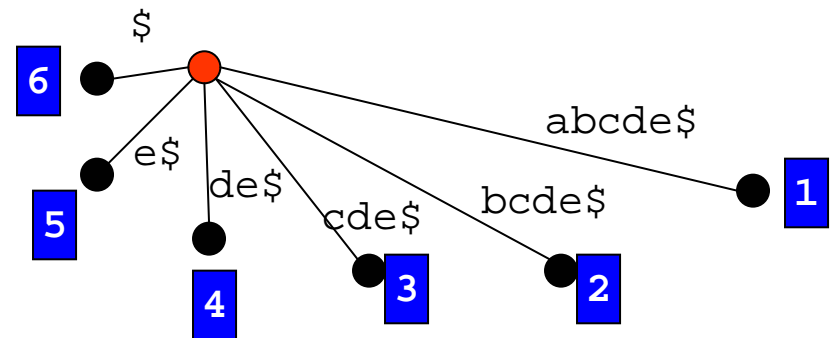
- Zu jedem String (plus \$) gibt es genau einen Suffixbaum
- Jeder Pfad von der Wurzel zu einem Blatt ist unterschiedlich
 - Nämlich auf alle Fälle unterschiedlich lang
- Jede Verzweigung an einem inneren Knoten ist eindeutig bzgl. des nächsten Zeichens auf dem Pfad
- Gleiche Substrings können an mehreren 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

- $S = \text{aaaaa}\$$



- $S = \text{abcde}\$$



Definitionen

- Definition

Sei T der Suffixbaum für String $S + "$"$

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

Anwendungen

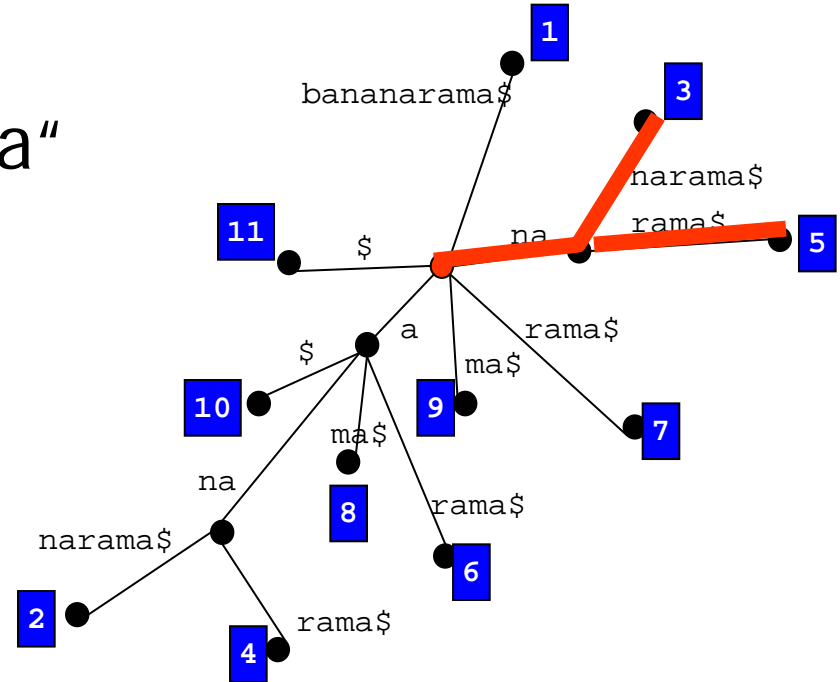
- Suche eines Pattern P
- Längster gemeinsamer Substring zweier Strings
- Längstes Palindrom

Suche mit Suffixbäumen

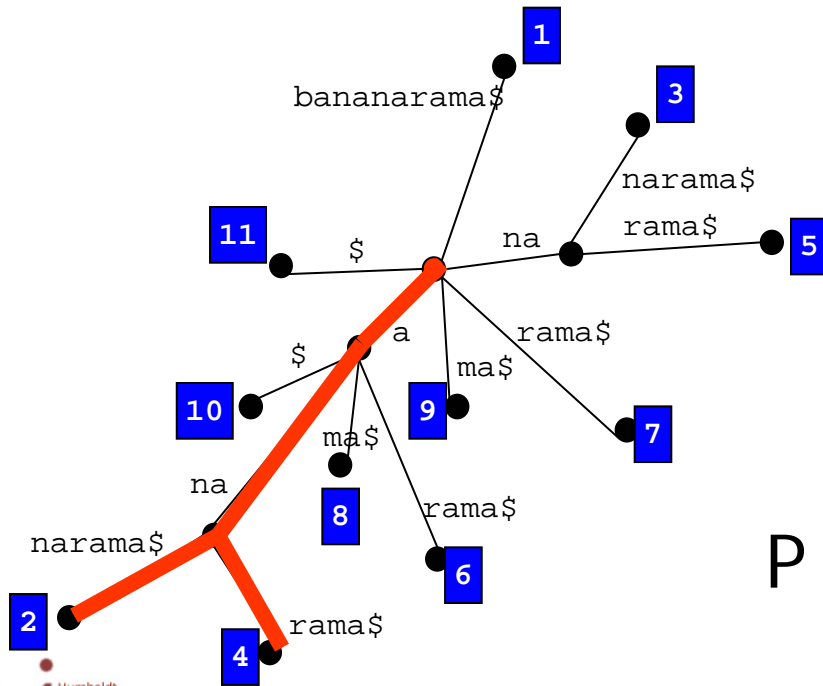
- Intuition
 - Jedes Vorkommen eines Suchpattern P muss **Präfix eines Suffix** sein
 - Und die haben wir alle auf Pfaden von der Wurzel aus
- Gegeben S und P. Finde alle Vorkommen von P in S
 - Konstruiere den Suffixbaum T zu S+ "\$"
 - Das geht in $O(|S|)$, wie wir sehen werden
 - Matche P auf einen 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

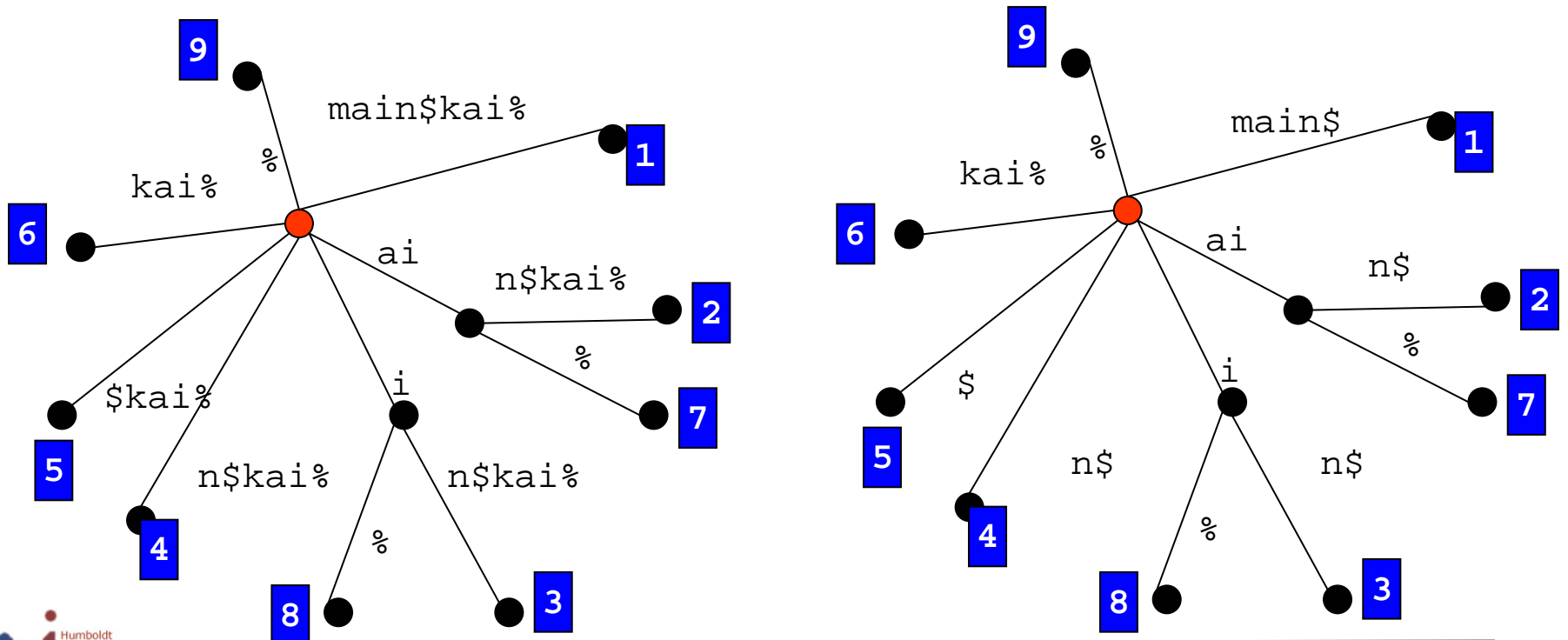
- Theorem
Sei T der Suffixbaum für $S + "$"$. Die Suche nach allen Vorkommen eines Pattern P , $|P|=n$, in S ist $O(n+k)$, wenn k die Anzahl Vorkommen von P in S ist.
- Beweisidee
 - P in T matchen kostet $O(n)$
 - Pfade sind eindeutig – Entscheidung an jedem Knoten ist klar
 - Damit maximal $O(n)$ Zeichenvergleiche
 - Blätter aufsammeln ist $O(k)$
 - Baum unterhalb Knoten K hat k Blätter
 - Die kann man in $O(k)$ finden
- Suche ist damit schlimmstenfalls $O(n+m)$
 - Aber das ist ein wirklich unwahrscheinlicher Worst case
- Frage: Was messen wir mit dem k wirklich? Wie erreicht man $O(k)$?

Längster gemeinsamer Substring

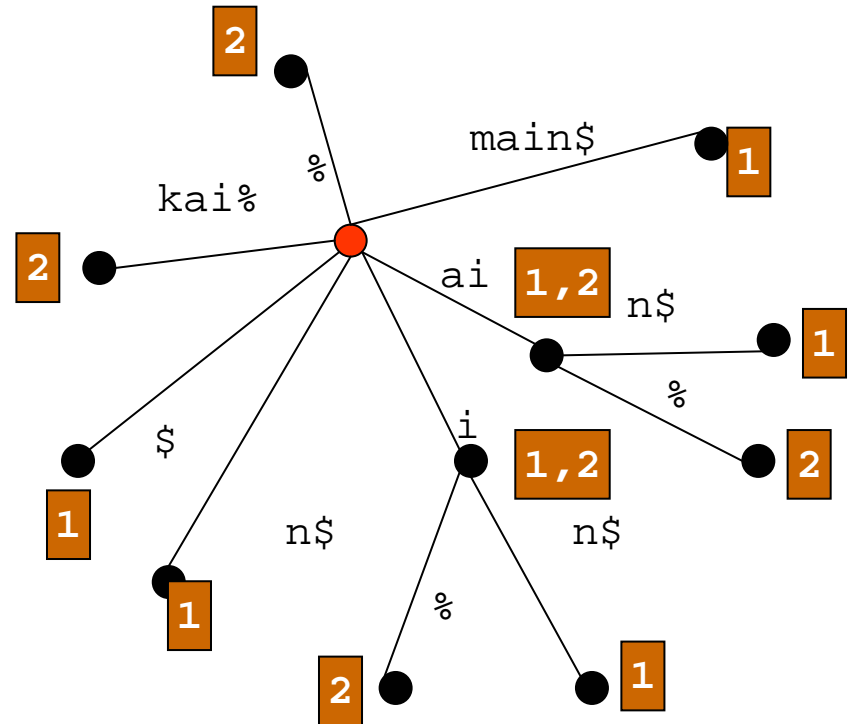
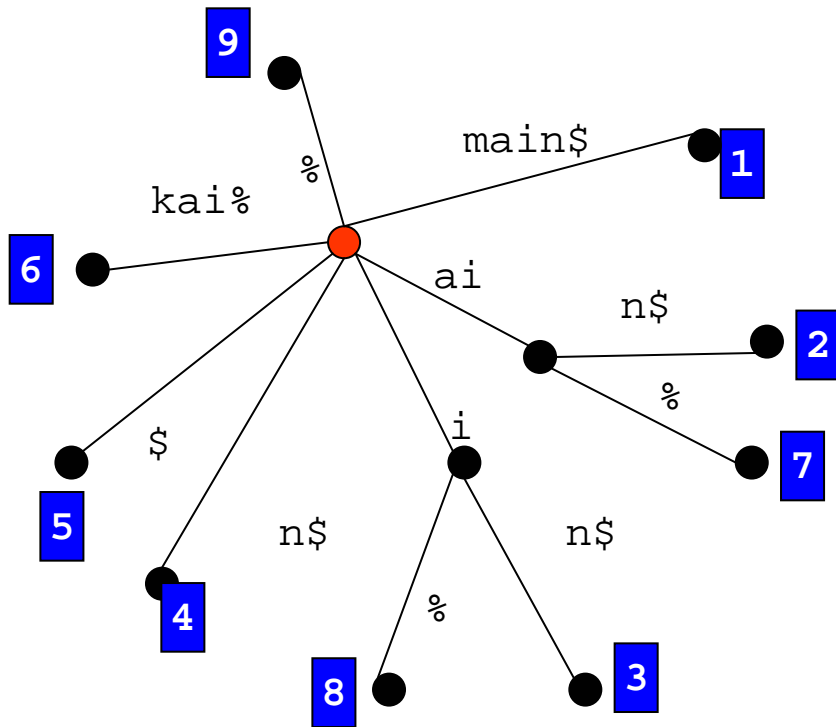
- Gegeben zwei Strings S_1 und S_2
- Gesucht: **Längster gemeinsamer Substring s**
- Vorschläge ?
- Lösung
 - Konstruiere Suffixbaum T für $S_1\$S_2\%$
 - Streiche aus diesem Baum alle Pfade unterhalb eines „\$“
 - Durchlaufe den Baum
 - markiere alle internen Knoten mit 1, wenn im Baum darunter ein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

- $S_1 = \text{main}\$, S_2 = \text{kai}\%$
- ...



Beispiel



- Verallgemeinerbar zu n Strings S_1, \dots, 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 1,2
 - 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
 - Suche längsten gemeinsamen Substring für S und reverse(S)

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
 - Schließlich muss es einen Mismatch geben
 - Alle bisher eingefügten Suffixe sind länger als S_{i+1} , also wird $\$$ nie mit $\$$ matchen
 - $\$$ kommt sonst nicht in S vor
 - Folgendes kann passieren ...

Naive Konstruktion von Suffixbäumen 2

- Konstruiere T_{i+1} aus T_i wie folgt ...
 1. S_{i+1} matched bis auf \$; **Mismatch auf einer Kante** n 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 „\$“
 - Markiere k' mit $i+1$
 2. S_{i+1} matched bis auf \$; **Mismatch am Ende einer Kante** n
 - 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. Mismatch **vor \$ auf einer Kante** n an Position j des Labels; der Mismatch in S_{i+1} sei an Position $j' < |S_{i+1}|$
 - 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 „ $S[j'..]S$ “
 - 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