Bioinformatik

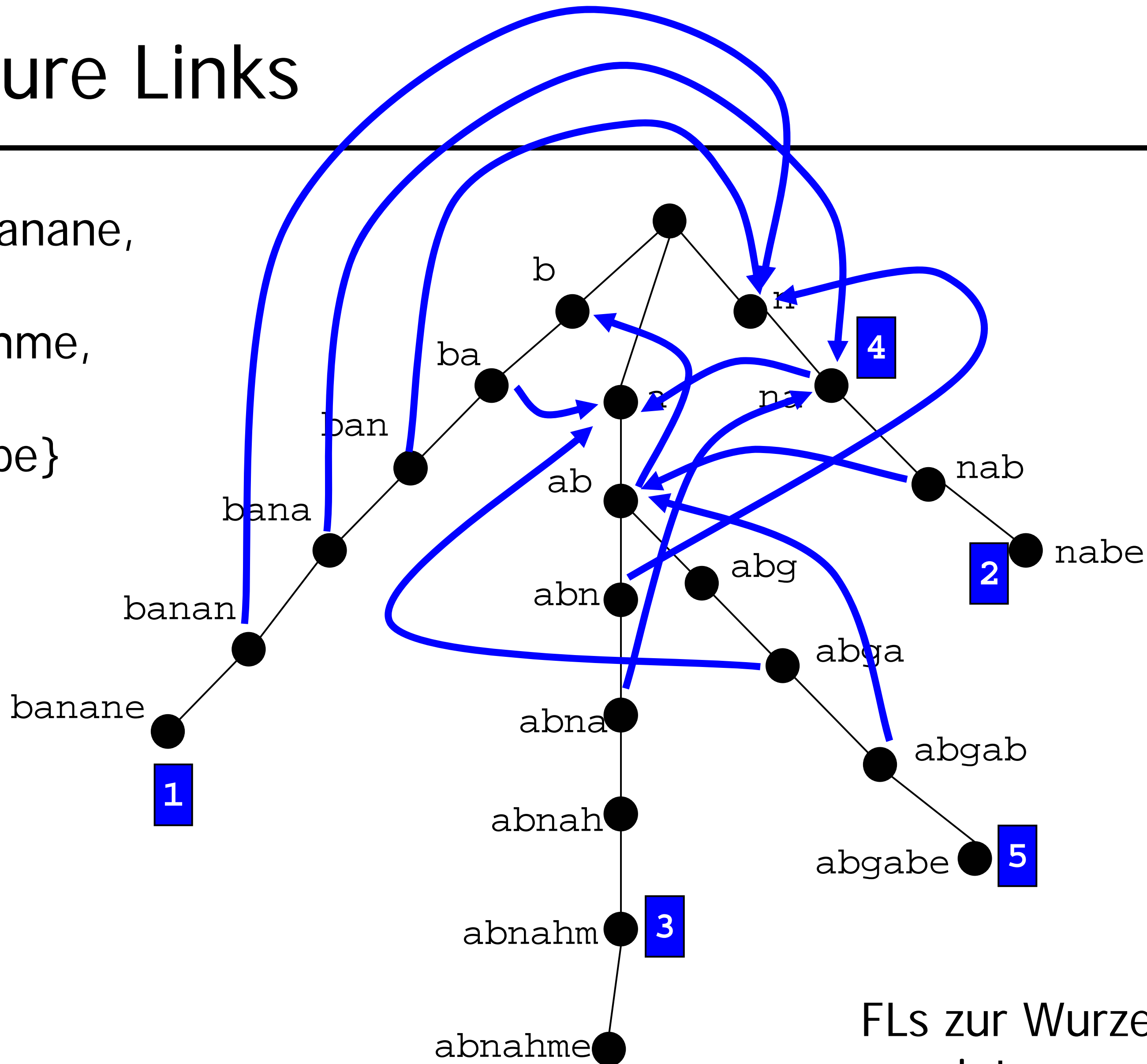
Suffixbäume

Ulf Leser

Wissensmanagement in der Bioinformatik

Failure Links

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



FLs zur Wurzel nicht gezeigt

Konstruktion der Failure Links

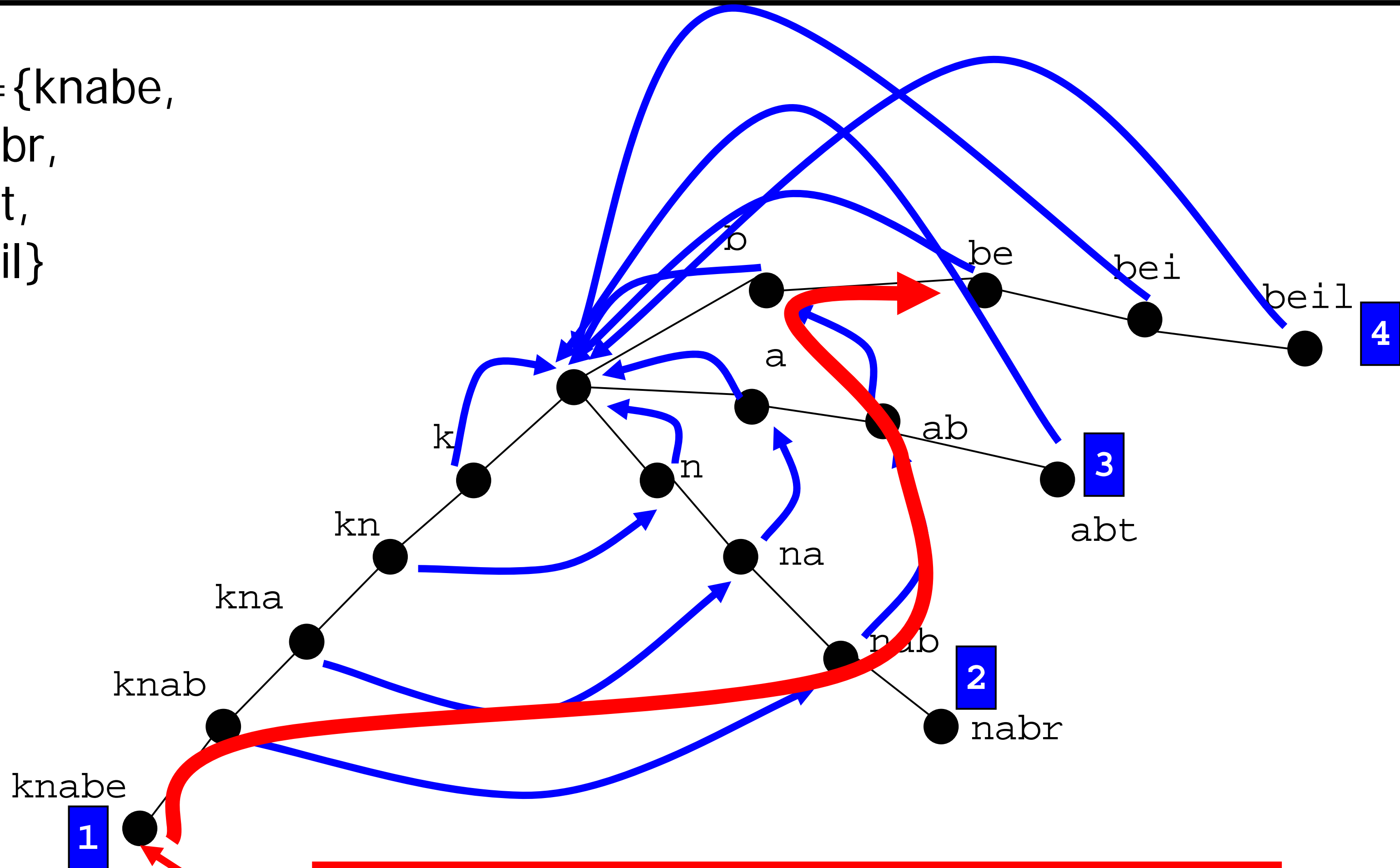
- Wir bauen erst (in linearer Zeit) den Keyword-Tree
- Dann alle Failure Links in O(n)
  - Beachte: Failure Links zeigen immer zu echten Suffixen
  - D.h., für alle k gilt: depth(k) > depth(fl(k))
  - Wir konstruieren alle Failure Links per Breitensuche
- Man kann zeigen
  - Nicht mehr als $|P_i|$ Sprünge über Failure Links notwendig für alle Knoten von P_i zusammen
  - Also: Konstruktion ist (mindestens) O(n)

Algorithmusidee 2

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

Beispiel

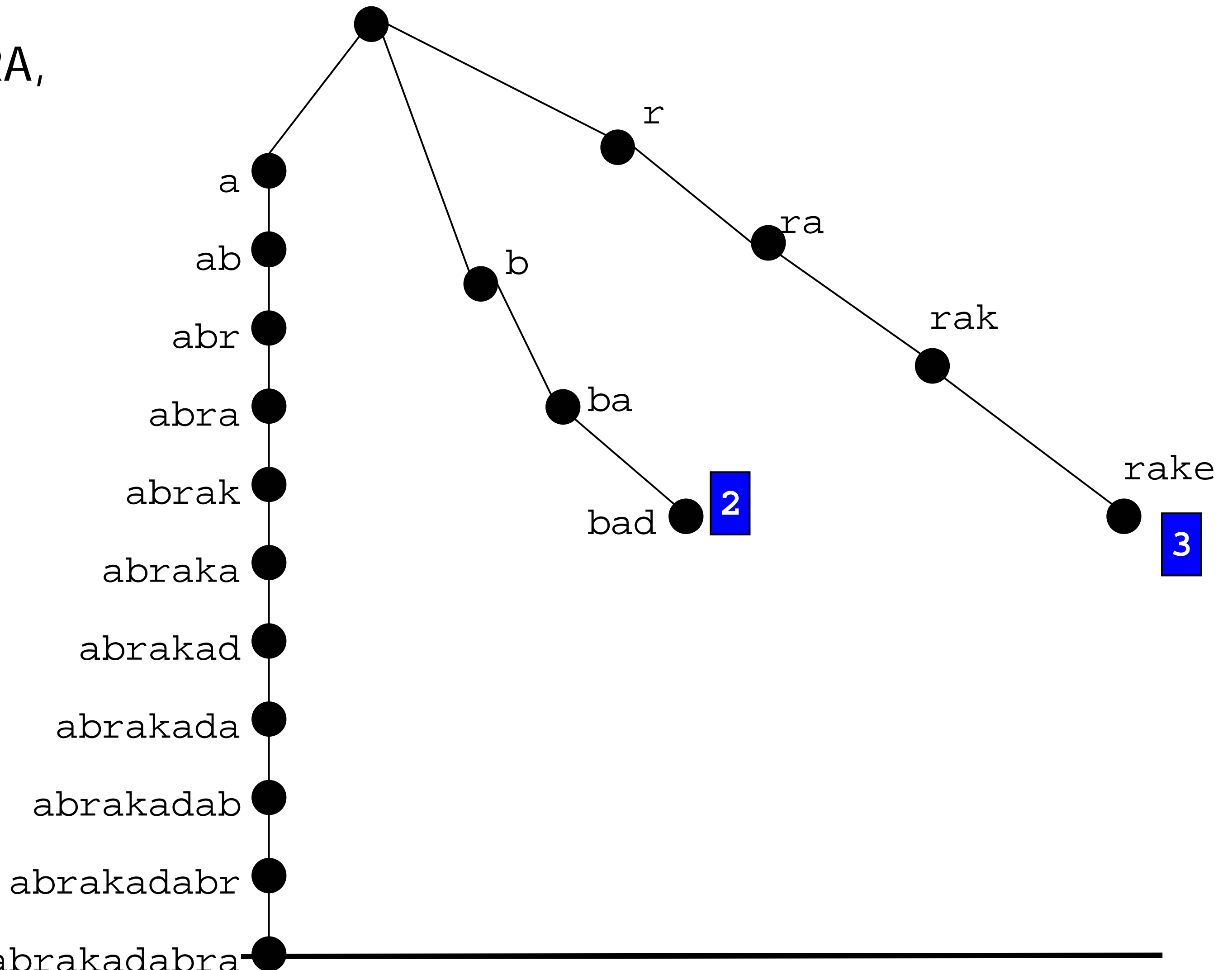
P={knabe,
nabr,
abt,
beil}



Failure Link für diesen Knoten suchen; alle FL für Knoten k mit depth(k)<depth(„knabe“) sind bekannt

Intuition – Ist das wirklich linear?

- P={
ABRAKADABRA,
BAD,
RAKE}



Intuition – 2

- P={ ABRAKADABRA, BAD, RAKE}

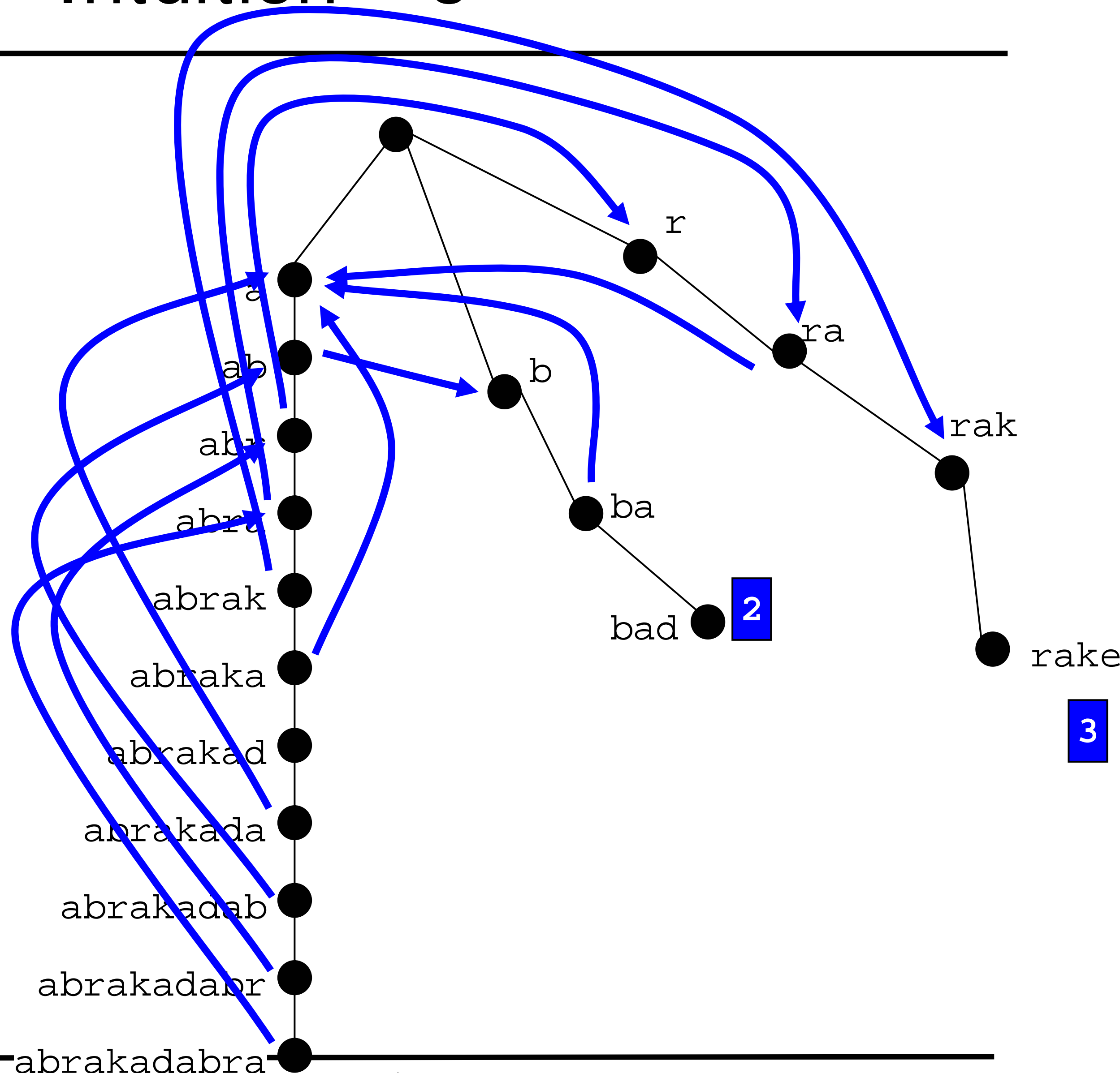
a
ab
abr
abra
abrak
abraka
abrakad
abrakada
abrakadab
abrakadabr
abrakadabra
1
b
ba
bad
2
r
ra
rak
rake
3

Intuition – 3

- Präfix|Suflänge|Max Suflänge

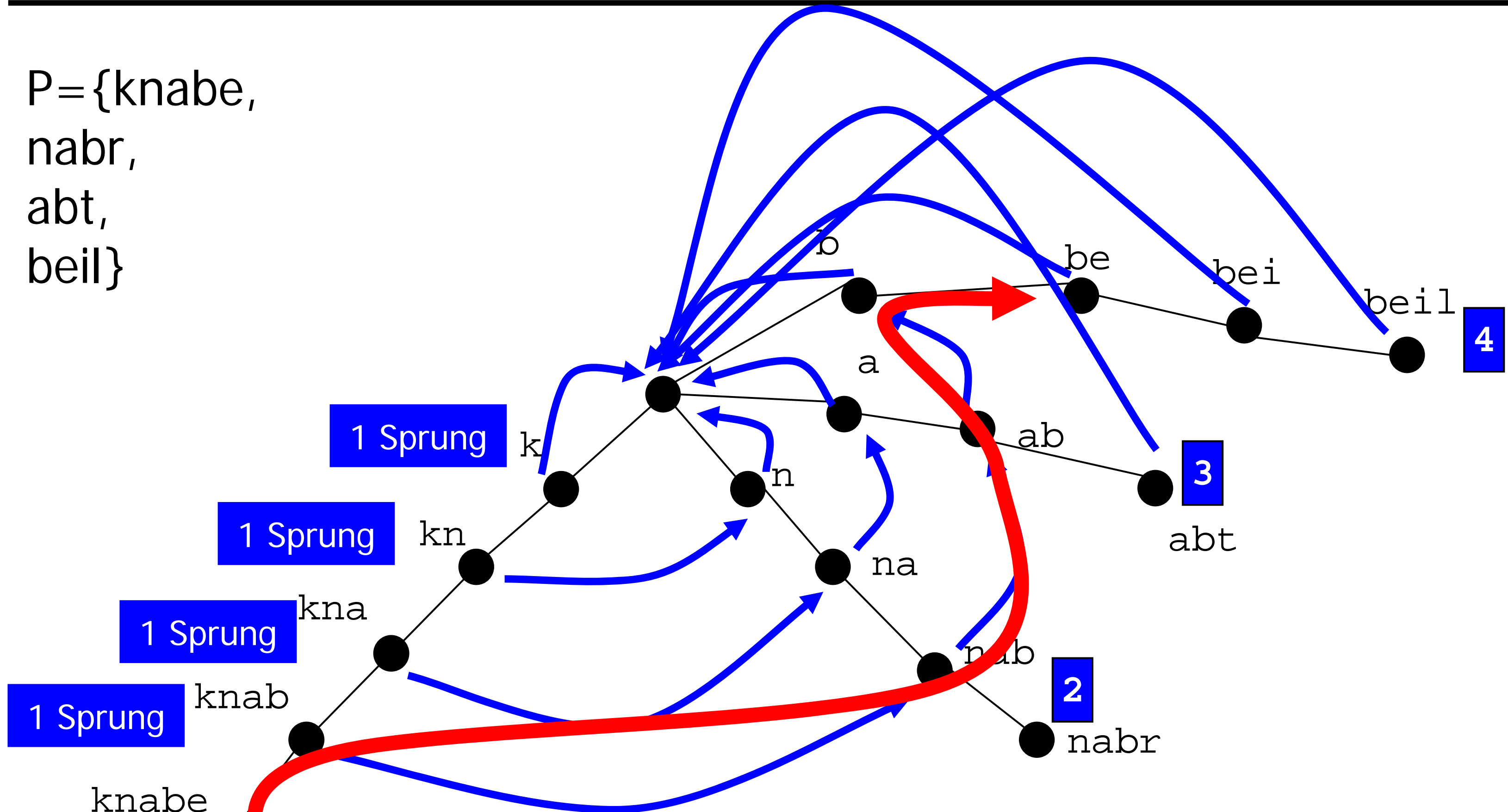
Präfix	Suflänge	Max Suflänge
A	0	1
AB	1	2
ABR	1	2
ABRA	2	3
ABRAK	3	4
ABRAKA	1	2
ABRAKAD	0	1
ABRAKADA	1	2
ABRAKADAB	2	3
…AKADABR	3	4
…AKADABRA	4	5

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



Mehrere Sprünge

P={knabe,
nabr,
abt,
beil}



3 Sprünge

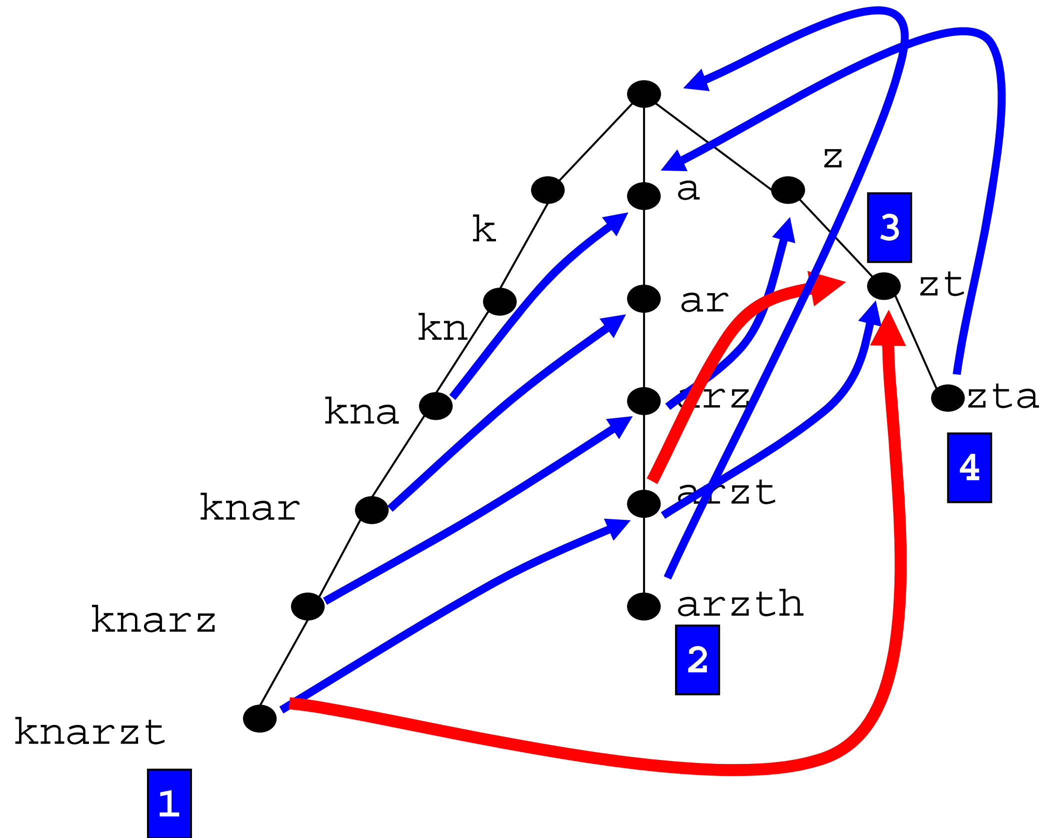
- Maximale Suffixe können in Zukunft nur noch n-3 lang sein
- Für einen FL kann man deshalb auch in Zukunft nur noch max n-3 Sprünge machen
- Gesamtzahl O(n) wird eingehalten

Spezialfall

- Problem: Pattern, die andere Pattern enthalten
- 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}



Suchphase mit Output Links

- Man muss bei jedem Knoten k, den man abläuft, nachsehen, ob es einen Output Link gibt
- Wenn ja, beschreite einen Nebenweg
  - Sei v=out(k)
  - Gib mark(v) aus
    - Der Zielknoten muss markiert sein
  - Wenn vorhanden, folge 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

```
P={AB**DA*A}                P1=AB, l1=1
   12345678                 P2=DA, l2=5
T={TABTABDADAZA}            P3=A,  l3=8
   123456789012

C=[000000000000]

             P1 an j=2,  z=2    => 010000000000
             P1 an j=5,  z=5    => 010010000000
             P2 an j=7,  z=3    => 011010000000
             P2 an j=9,  z=5    => 011020000000
             P3 an j=2,  z=-5   => 011020000000
             P3 an j=5,  z=-2   => 011020000000
             P3 an j=8,  z=1    => 111020000000
             P3 an j=10, z=3    => 112020000000
             P3 an j=12, z=5    => 112030000000
                                            ^ Treffer
```

Nachspiel

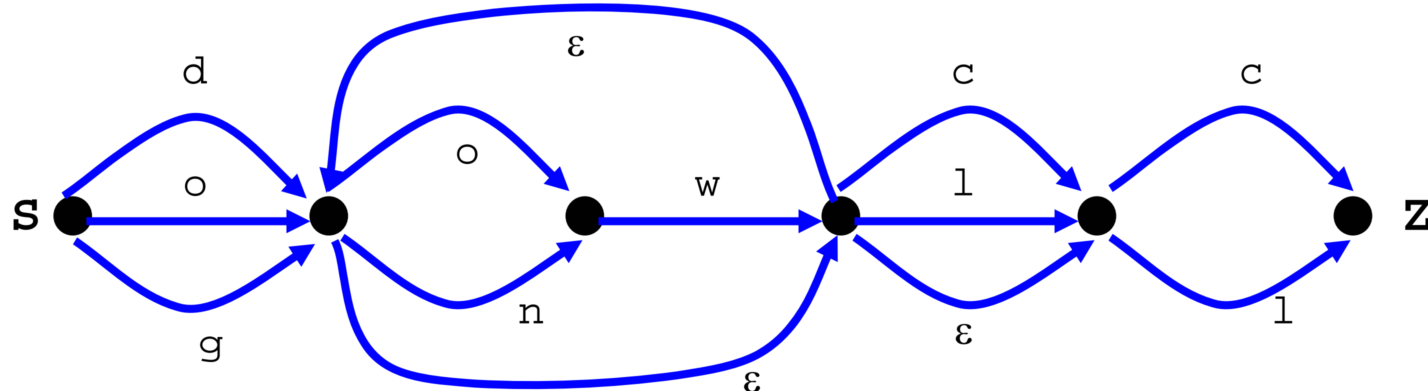
- Wir können in linearer Zeit
  - Alle Vorkommen eines Pattern P in einem Template T finden
  - Alle Vorkommen einer Menge von Pattern in einem Template T finden
  - Alle Vorkommen eines Pattern P mit Wildcard in einem Template T finden
  - Alle Vorkommen eines Pattern P mit maximal k Mismatches in T 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

Suche mit regulären Ausdrücken

- Reguläre Ausdrücke
  - „ε" Leeres Zeichen
  - „|" „Oder"
  - „()" Gruppierung
  - „*" Kleensche Hülle
  - (Es fehlen „+", „.", „[]", Zählen, ...)
  - Rekursive Definition sollte bekannt sein
- PROSITE: Datenbank für Motiven
  - Proteindomänen – Funktionale Einheiten in Proteinsequenzen
  - Beschrieben durch reguläre Ausdrücke
  - Aber andere Operatoren als die oben genannten

Problem

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



Definitionen

- Definition
  - *Sei R ein regulärer Ausdruck. Dann ist G(R) der dazugehörige endliche Automat*
  - *Ein Substring T' von T matched mit R, wenn T' durch einen Pfad in G(R) von S nach Z ausgesprochen wird*

- Komplexität der Konstruktion von G(R) ist linear
- Die Menge aller Strings, die von G(R) akzeptiert werden, bilden die Sprache zu R
- Gesucht: Algorithmus, um alle T' in T zu finden, die mit R matchen

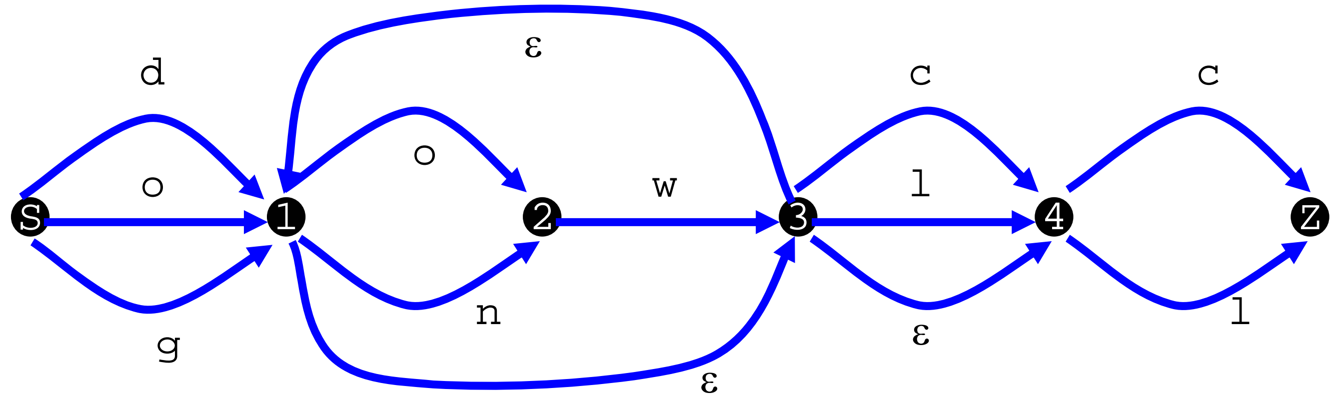
Algorithmusidee

- Wir beginnen mit dem etwas leichteren Problem
  - Matched ein Präfix von T mit R?
- Idee
  - Betrachte G(R) und T von links nach rechts
  - Aufzählen aller Pfade in G(R) mit steigender Länge, die mit T matchen
  - Wird Terminal Z gefunden, haben wir einen Match
- Erweiterung zu beliebigen Substrings von T
  - Betrachte Pattern R'=„Σ*R"
  - Präfix Σ* von R' „frisst" beliebige Präfixe von T

Pfadaufzählung

- Induktion über Pfadlänge i
  - Anfang: Sei N(0) die Menge aller Knoten, die von S per ε-Kante erreicht werden können. Außerdem ist $S \in N(0)$
  - Schritt:
    - Sei N(i-1) bekannt
    - N(i) ist die Menge aller Knoten, die von einem Knoten aus N(i-1) erreicht werden durch:
      - erst genau eine Kante mit Label T(i)
      - dann keine, eine, oder mehrere ε-Kanten
- Enthält N(i) das Terminalsymbol Z, endet im Template T an Position i ein Auftreten von R

Beispiel



- Pattern: (d|o|g)((n|o)w)*(c|l|ε)(c|l)
- Suche: ol
  - N(0) = {S}
  - N(1) = {1,3,4}
  - N(2) = {4,Z} Success
- Suche: dnwnwowc
  - N(0) = {S} dnwnwowc
  - N(1) = {1,3,4} dnwnwowc
  - N(2) = {2} dnwnwowc
  - N(3) = {3,4,1} dnwnwowc
  - N(4) = {2} dnwnwowc
  - N(5) = {3,4,1} dnwnwowc
  - N(6) = {2} dnwnwowc
  - N(7) = {3,4,1} dnwnwowc
  - N(8) = {4,Z} Success

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(R) ist
    - N ist eine Menge, keine Multimenge!
    - Wie wollen nur wissen, welche Zustände wir erreichen können – wie, ist egal
    - Zyklen können abgebrochen werden
  - Also ist dieser Schritt O(e)
- Wir berechnen m Mengen N(1) … N(m)
- Also O(m*e)
- Beobachtung
  - Ein regulärer Ausdruck mit n=|R| Symbolen hat höchstens n ε-Kanten
  - Sonst kann er minimiert werden
- Zusammen: O(m*n)

Inhalt dieser Vorlesung

- Suffixbäume
- Verwendung von Suffixbäumen
- Naive Konstruktion

Wo sind wir überhaupt?

- 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
- 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 – approximative Suche – später
- Für exakte Suche sind Suffixbäume die schnellste Datenstruktur
  - Aber: Speicherplatz, Sekundärspeicherverhalten
  - Alternative: Suffix-Arrays
- Außerdem: Viele weitere Anwendungen von Suffixbäumen
  - Vorstufe des approximativen Suchens: Suche nach „Seeds“
    - 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 $|S|=m$

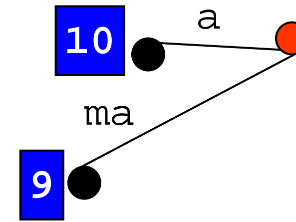
Suffixbäume

- Definition: *Der Suffixbaum T für S ist ein Baum mit*
  - *T hat eine Wurzel und 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 $(k_1,k_2,\ldots,k_n)$ 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]*
- Intuition: Kompakte Repräsentation aller Suffixe von S in einem Baum
- Beachte: Bei Suffixbäumen stehen Substrings an Kanten, bei Keyword Trees einzelne Zeichen

Beispiel 1

1234567890

- S= BANANARAMA

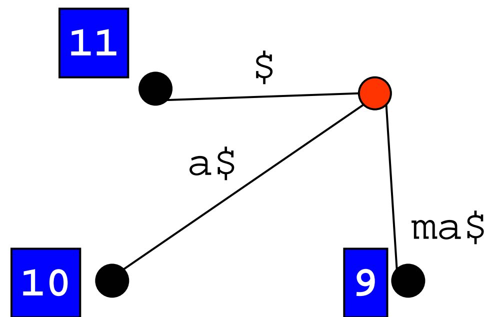


- Problem: Wohin kommt „AMA“?
  - Verlängerung von „a“ verboten – 10 sonst kein Blatt
  - Neue Kante „ama“ verboten – zwei Pfade aus der Wurzel würden sonst mit gleichem Zeichen beginnen
  - ➢ Es gibt keinen Suffixbaum für BANANARAMA
  - ➢ Problem tritt auf, sobald ein Suffix Präfix eines anderen Suffix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

Beispiel 2

```
        12345678901
```

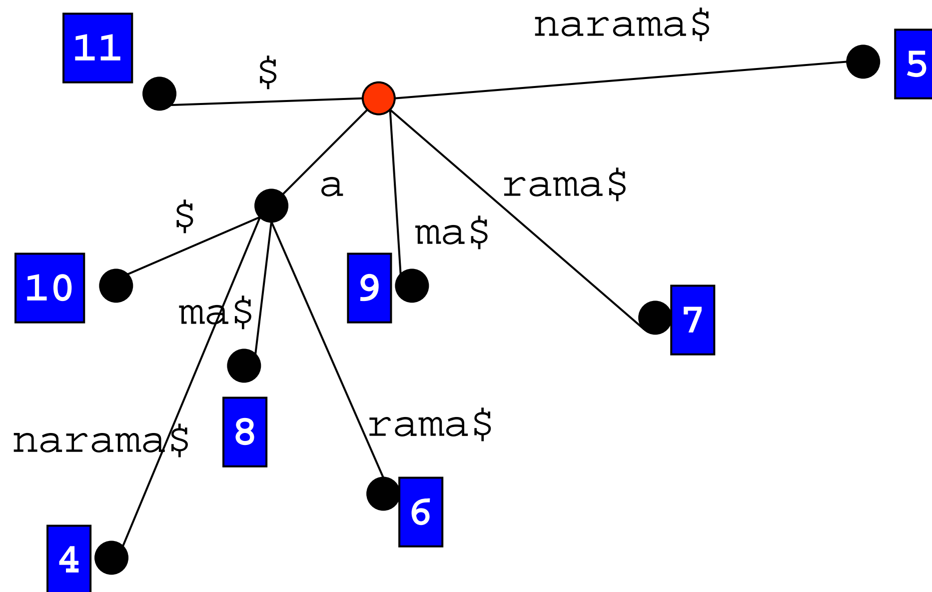
- S= BANANARAMA$



Beispiel 2

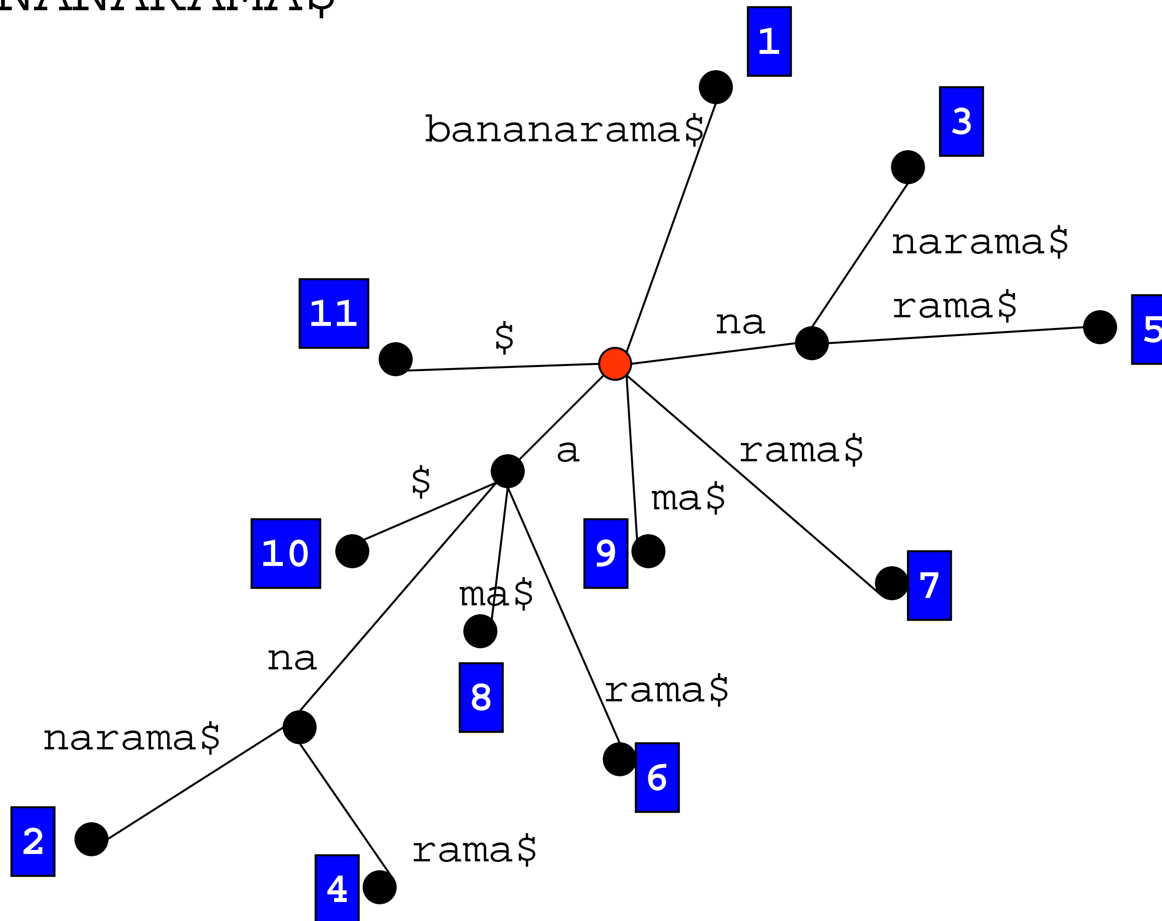
```
      12345678901
```

- S= BANANARAMA$



Beispiel 3

      12345678901
- S= BANANARAMA$

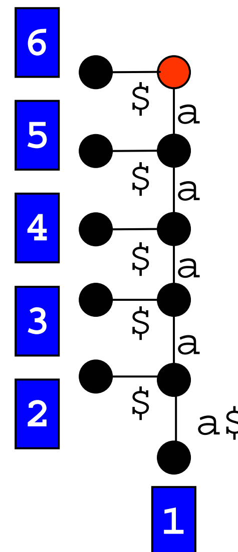


Eigenschaften von Suffixbäumen

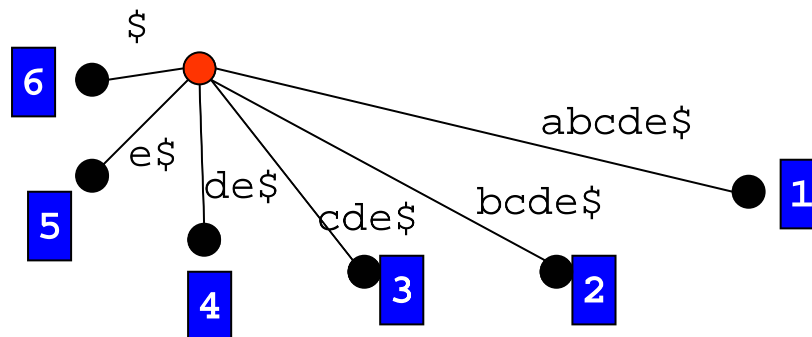
- Zu jedem String (plus $) gibt es genau einen Suffixbaum
- Jeder Pfad von der Wurzel zu einem Blatt ist unterschiedlich
  - Nämlich auf alle Fälle unterschiedlich lang
- Jede Verzweigung an einem inneren Knoten ist eindeutig bzgl. des nächsten Zeichens auf dem Pfad
- Gleiche Substrings können an mehreren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



- 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) 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

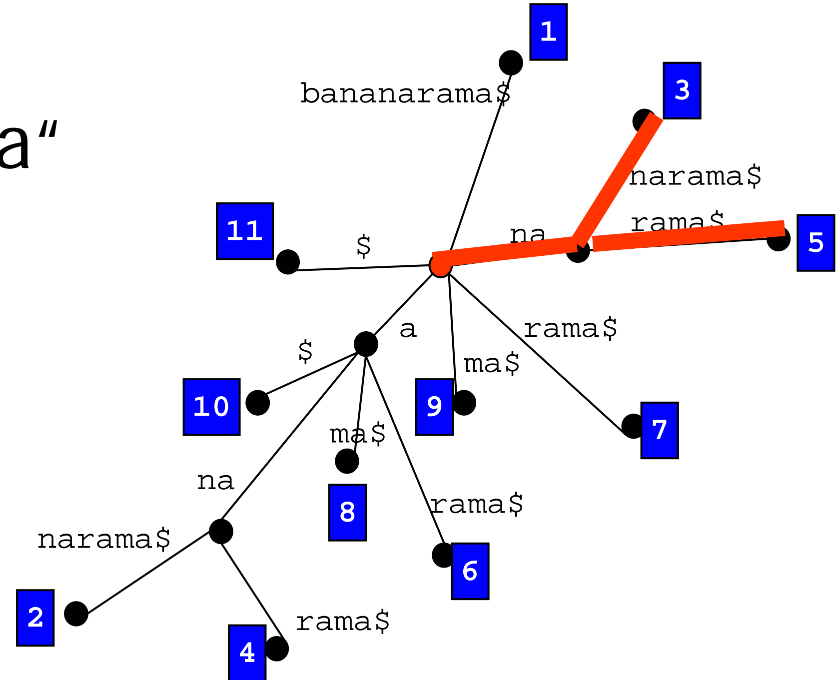
- Suche eines Pattern P
- Längster gemeinsamer Substring zweier Strings
- Längstes Palindrom

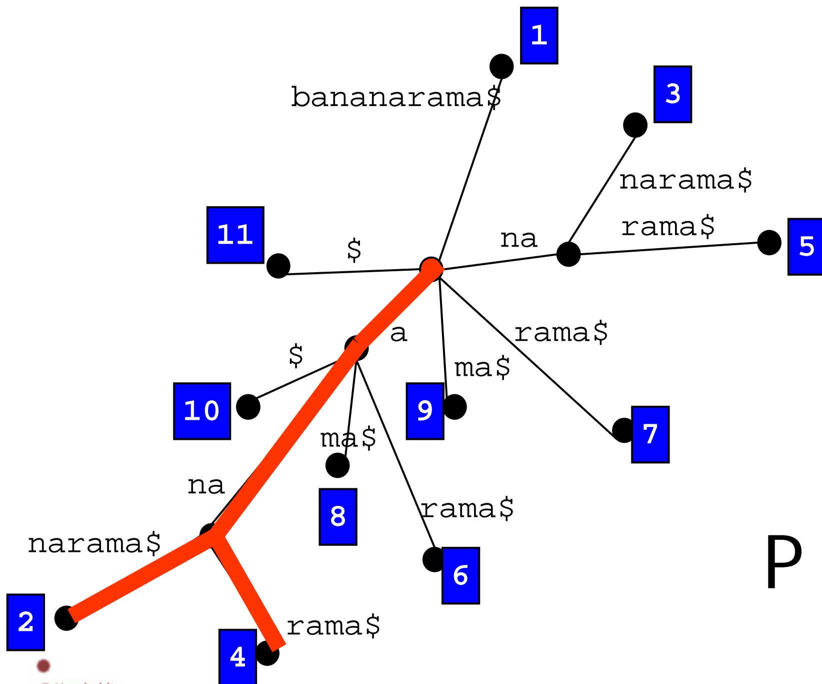
Suche mit Suffixbäumen

- Intuition
  - Jedes Vorkommen 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 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

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%
- ...

Anwendungen

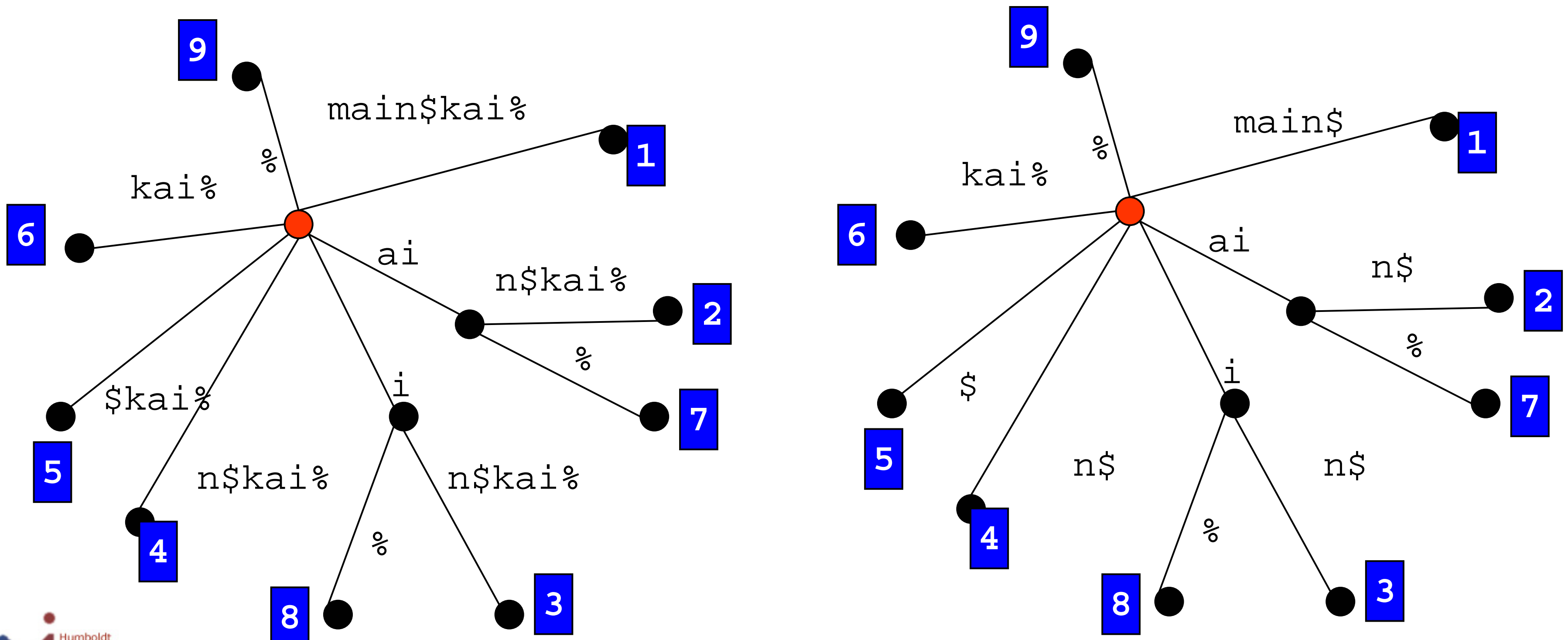
- Suche eines Pattern P
- Längster gemeinsamer Substring zweier Strings
- Längstes Palindrom

Längster gemeinsamer Substring

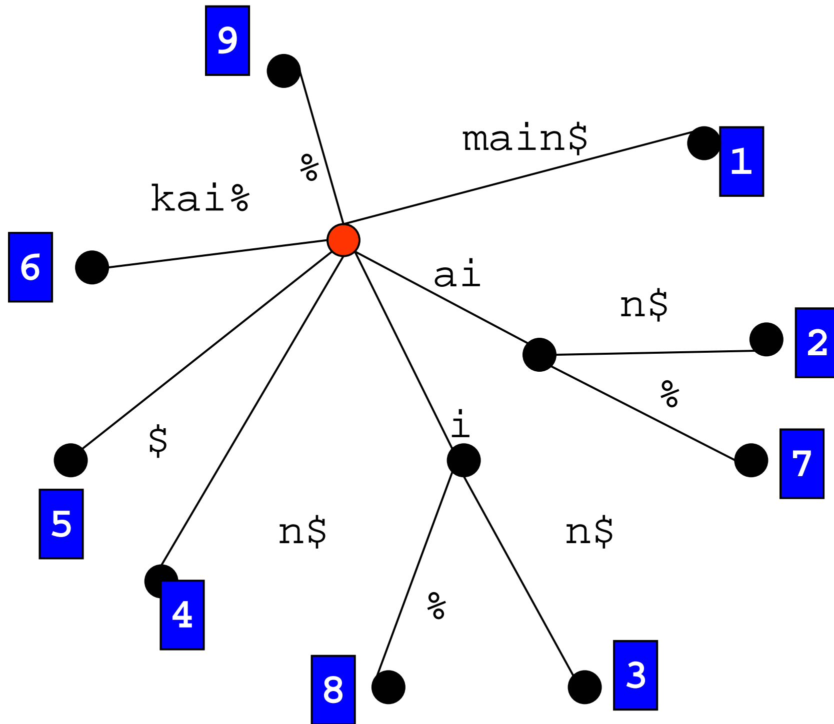
- Gegeben zwei Strings S_1 und S_2
- Gesucht: Längster gemeinsamer Substring s
- Vorschläge ?
- Lösung
  - Konstruiere Suffixbaum T für $S_1\$S_2\%$
  - Streiche aus diesem Baum alle Pfade unterhalb eines „$“
  - Durchlaufe den Baum
    - markiere alle internen Knoten mit 1, wenn im Baum darunter ein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

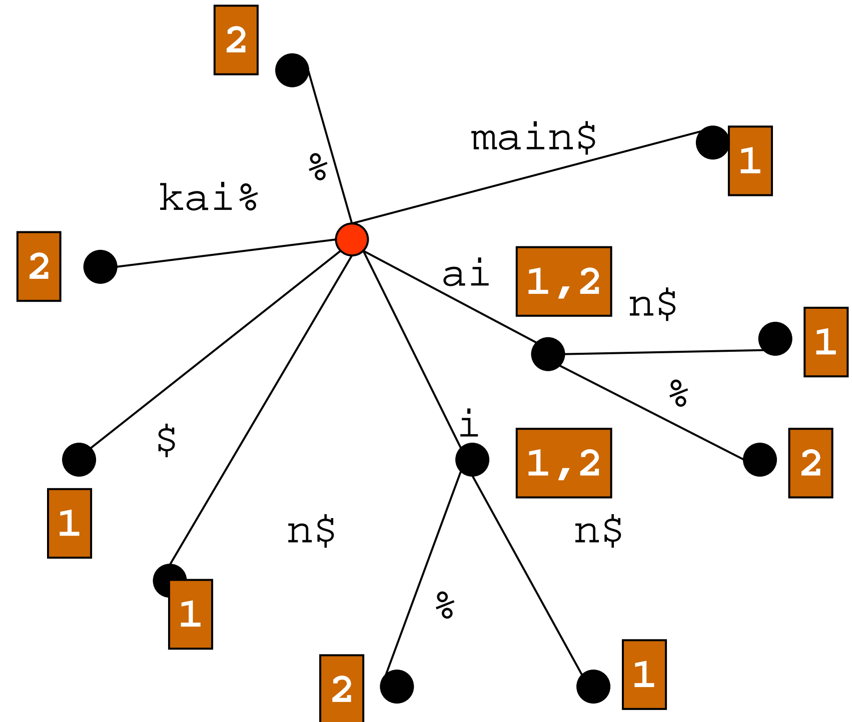
Beispiel

- 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 1,2
    - 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'..]$"
   - 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