

Bioinformatik

Suche nach mehreren Mustern – Teil II
Zwei Varianten der Stringsuche



Ulf Leser

Wissensmanagement in der
Bioinformatik



Inhalt dieser Vorlesung

- Wiederholung
 - Keyword-Trees
 - Failure Links
 - Suche mit Failure Links
- Output-Links
- Konstruktion in linearer Zeit

- Suche mit Wildcards

Suche nach mehreren Mustern

- Bisher: Suche eines Pattern P in einem String T
- Jetzt: Suche nach **mehreren Pattern** P_1, \dots, P_z in einem String T

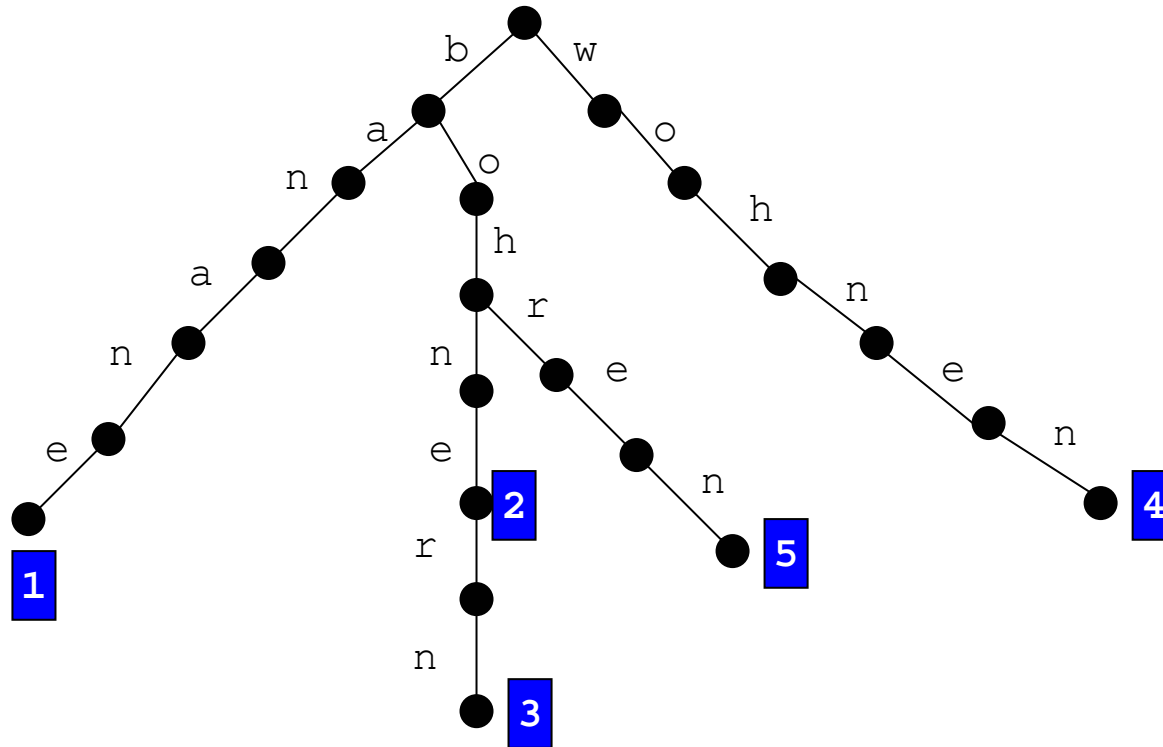
- Motivation
 - Suchmaschinen: „xyz AND abc“
 - Beispiel 1: Lokalisierung einer neuen Sequenz auf einer STS/EST Karte
 - Beispiel 2: Suche nach typischen Primern / Vektoren in einer DNA Sequenz (Maskierung)

Erster Versuch

- Sei $P = \{P_1, P_2, \dots, P_z\}$, $n = |P_1| + |P_2| + \dots + |P_z|$
- KMP braucht $O(m + |P_i|)$ für Suche mit einem Pattern P_i
 - $O(|P_i|)$ Preprocessing
 - $O(m)$ Suche durch T
- Naive Erweiterung auf z Pattern
 - Preprocessing für jedes Pattern
 - $O(|P_1| + |P_2| + \dots + |P_z|) = O(n)$
 - **Sequentielle Suche** aller Pattern in T: $O(z * m)$
 - Zusammen: $O(n + z * m)$
- Das geht schneller

Keyword Tree

- $P = \{\text{banane, bohne, bohner, wohnen, bohren}\}$



KMP für mehrere Pattern

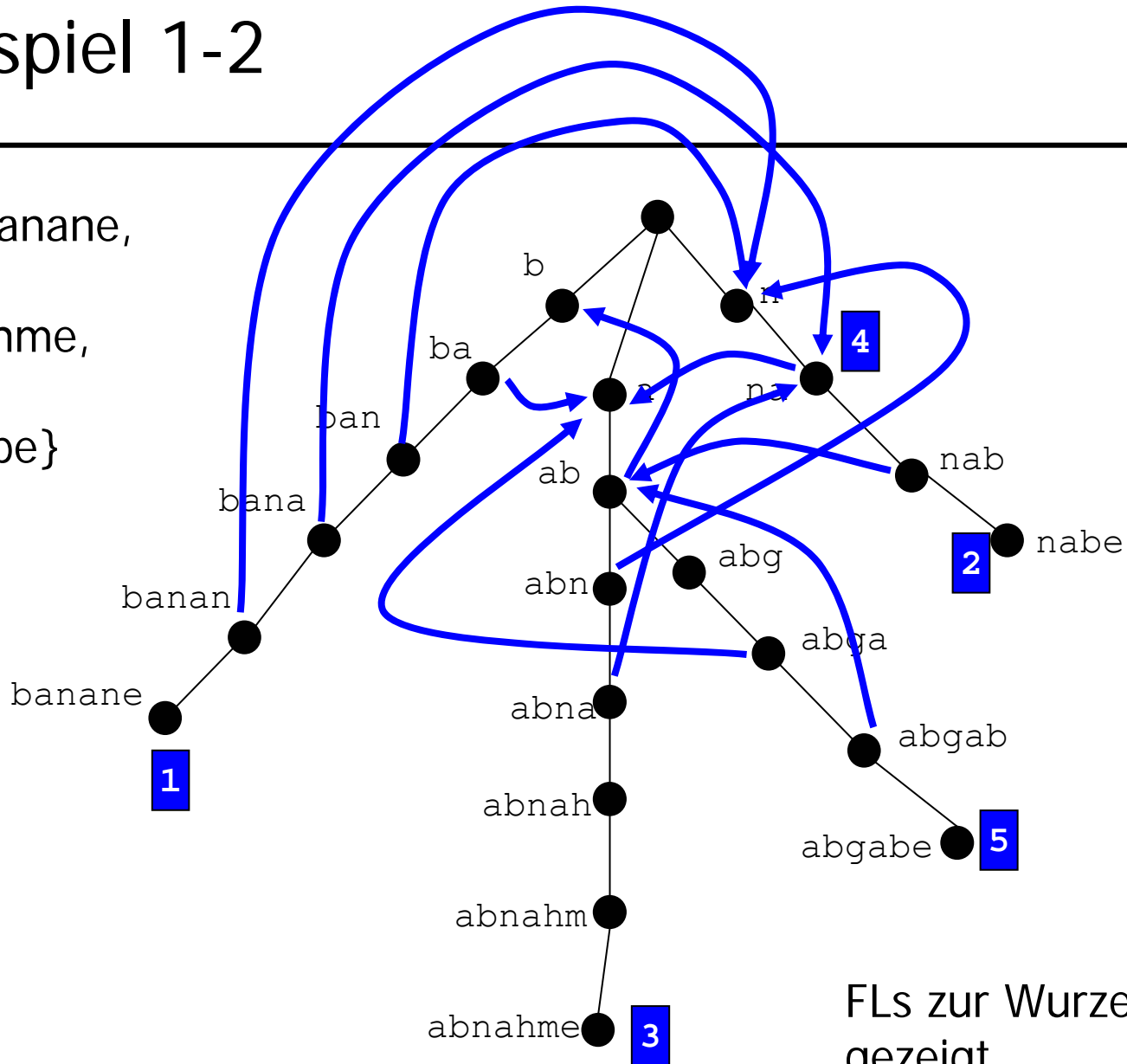
- Herzstück des KMP ist
 - „Sei sp_i die *Länge des längsten echten Suffix von $P[1..i]$, das mit einem Präfix von P matched*“
 - Verbesserung zu sp_i' betrachten wir hier nicht
 - Durch sp_i „merkt“ sich KMP die Zeichen von T vor dem Mismatch
 - sp_i erlaubt, kein Zeichen von T zweimal zu matchen
- **Aho-Corasick Algorithmus (AC)**
 - Übertragung des Tricks
 - Statt sp_i bauen wir **Failure Links** im Keyword Tree
 - Failure Links zeigen auf Knoten im Baum, deren Label echte Suffixe mindestens eines Pattern P_i sind
 - Verallgemeinerung der sp_i Werte
 - Wenn es einen Mismatch gibt, springen wir zu passenden Präfixen
 - Beachte: Per Konstruktion ist jedes Präfix von Pattern ist Label von genau einem Knoten

Failure Links

- Definition: Sei K ein Keyword Tree für Patternmenge P .
 $\forall k \in K$ ist
 - $length(k)$ die Länge des längsten echten Suffix von $label(k)$, das auch Präfix eines beliebigen Pattern aus P ist
 - Gibt es kein solches Suffix, dann $length(k)=0$
 - $fl(k)$ der Knoten für den gilt:
 $label(fl(k)) = label(k)[|label(k)|-length(k)+1 .. |label(k)|]$
 - Für Knoten k mit $length(k)=0$ gilt: $fl(k)=root$
- Beachte
 - Die Verbindung $(k, fl(k))$ heißt **Failure Link**
 - $label(fl(k))$ ist genau das „längste echte Suffix“ von $label(k)$, das wir suchen
 - $fl(k)$ ist eindeutig

Beispiel 1-2

$P = \{\text{banane, nabe, abnahme, na, abgabe}\}$



FLs zur Wurzel nicht gezeigt

Suchen mit Failure Links

- Gegeben: Keyword Tree K mit Failure Links, T
- Wir suchen ab Position j in T
- Matche Substring $T[j..]$ in K
 - Bei Match gehe weiter den Baum hinab
 - $j=j+1$
 - Wird ein markierter Knoten erreicht ist, gib Pattern aus
 - Wenn Blatt k erreicht ist an Position $j+x-1$
 - Jedes Blatt ist markiert; gib Pattern aus
 - Folge dem Failure Link aus k zu Knoten $fl(k)$
 - $lab(fl(k))$ haben wir gerade in T gesehen
 - Bzw. $fl(k)$ geht zur Wurzel
 - Matche weiter: in T ab $j=j+x$ und in K ab Kanten ausgehend von $fl(k)$

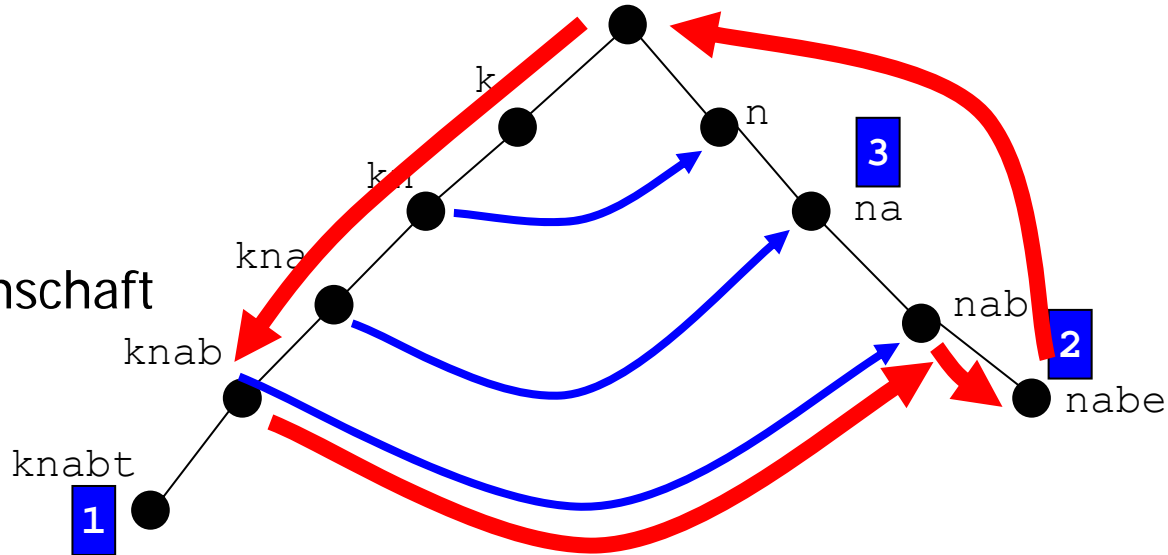
Suchen mit Failure Links

- Gegeben: Keyword Tree K mit Failure Links, T
- Wir suchen ab Position j in T
- Matche Substring $T[j..]$ in K
 - Bei Match ...
 - Wenn Blatt k erreicht ist ...
 - Bei einem Mismatch an Position x in T
 - Sei k der Knoten, dessen Label mit $T[j..x-1]$ matched
 - Alle Kinder von k sind also Mismatches für $T(j+x)$
 - Sonst können wir weiter im Baum gehen
 - Folge dem Failure Link aus k zu Knoten $fl(k)$
 - $lab(fl(k))$ haben wir gerade in T gesehen
 - Matche weiter: in T ab $j=j+x$ und in K ab Kanten ausgehend von $fl(k)$

Alles klar?

$P = \{\text{knabt}, \text{nabe}, \text{na}\}$

$T = \text{knabenschaft}$



- Algorithmus matcht KNAB in T
- B ist der letzte Match – Failure Link zu NAB
- Erweiterung zu NABE – Treffer P_2
- FL zu Root; Matchen geht weiter in T ist mit NSCHAFT
- P_3 (NA) wurde übersehen!
 - Grund: Pattern P_2 enthält P_3
 - Darüber wird man sprechen müssen



-
- Weiter geht's

Konstruktion der Failure Links

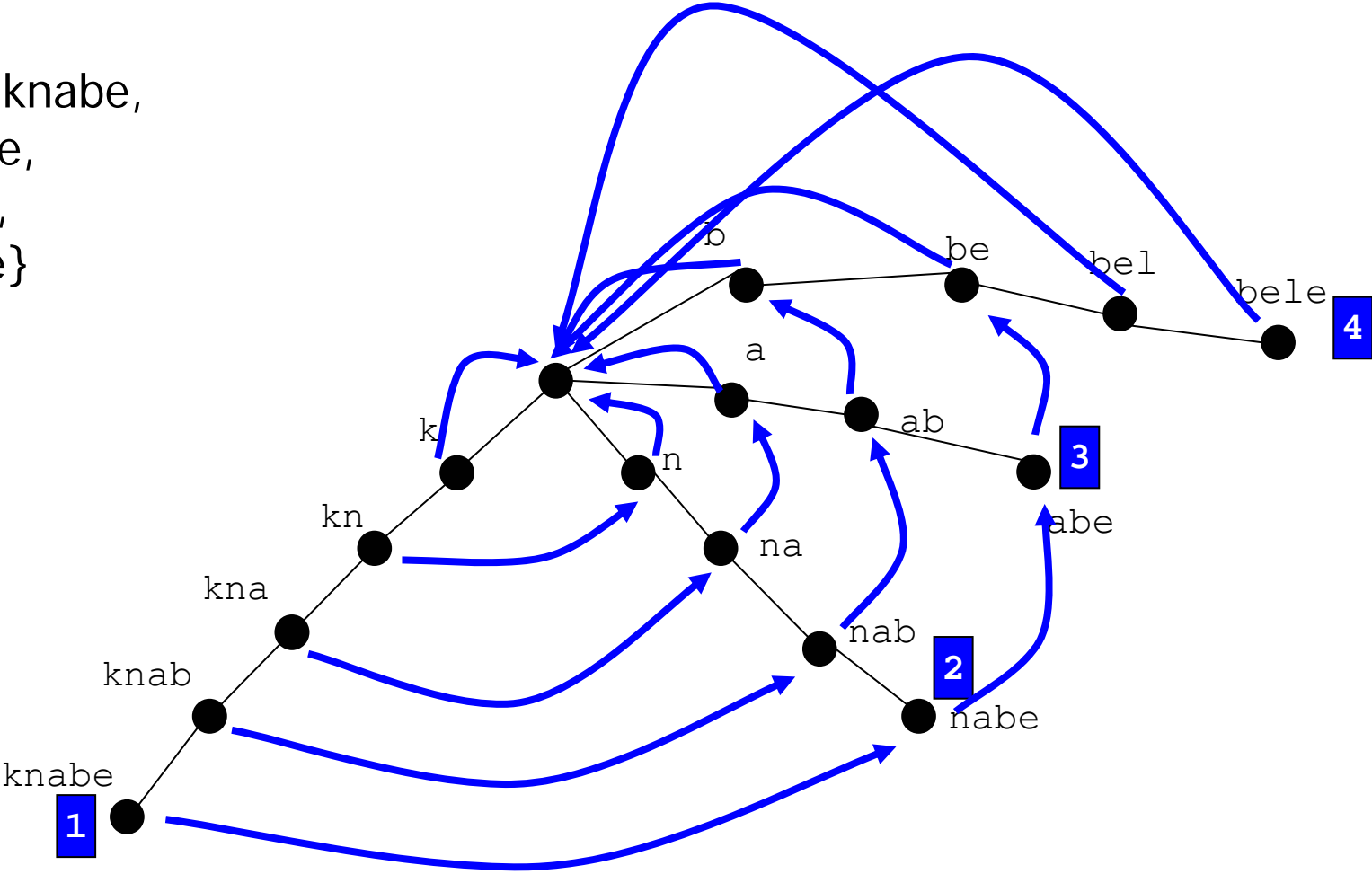
- Bisher
 - Mit Failure Links ist die Suchphase $O(m)$
 - Konstruktion des Keyword Trees ist $O(n)$
 - Wie lange braucht man, um Failure Links zu berechnen?
 - ... und dann war da noch das spezielle Problem ...
- Definition
 - Sei *depth(k)* die Tiefe des Knoten *k* (Abstand zu root)
- Wir bauen erst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

Beobachtung

- Betrachten wir einen Keyword Tree und einen Knoten k
- Alle Präfixe, die identisch zu einem Suffix von $lab(k)$ sind, erreichen wir durch Failure Links, und zwar in der Reihenfolge ihre Länge
 - Das längste Präfix findet man mit $fl(k)$
 - Wenn es weitere gibt, findet man die mit $fl(fl(k))$, $fl(fl(...))$

Beispiel

$P = \{\text{knabe, nabe, abe, bele}\}$



Algorithmusidee

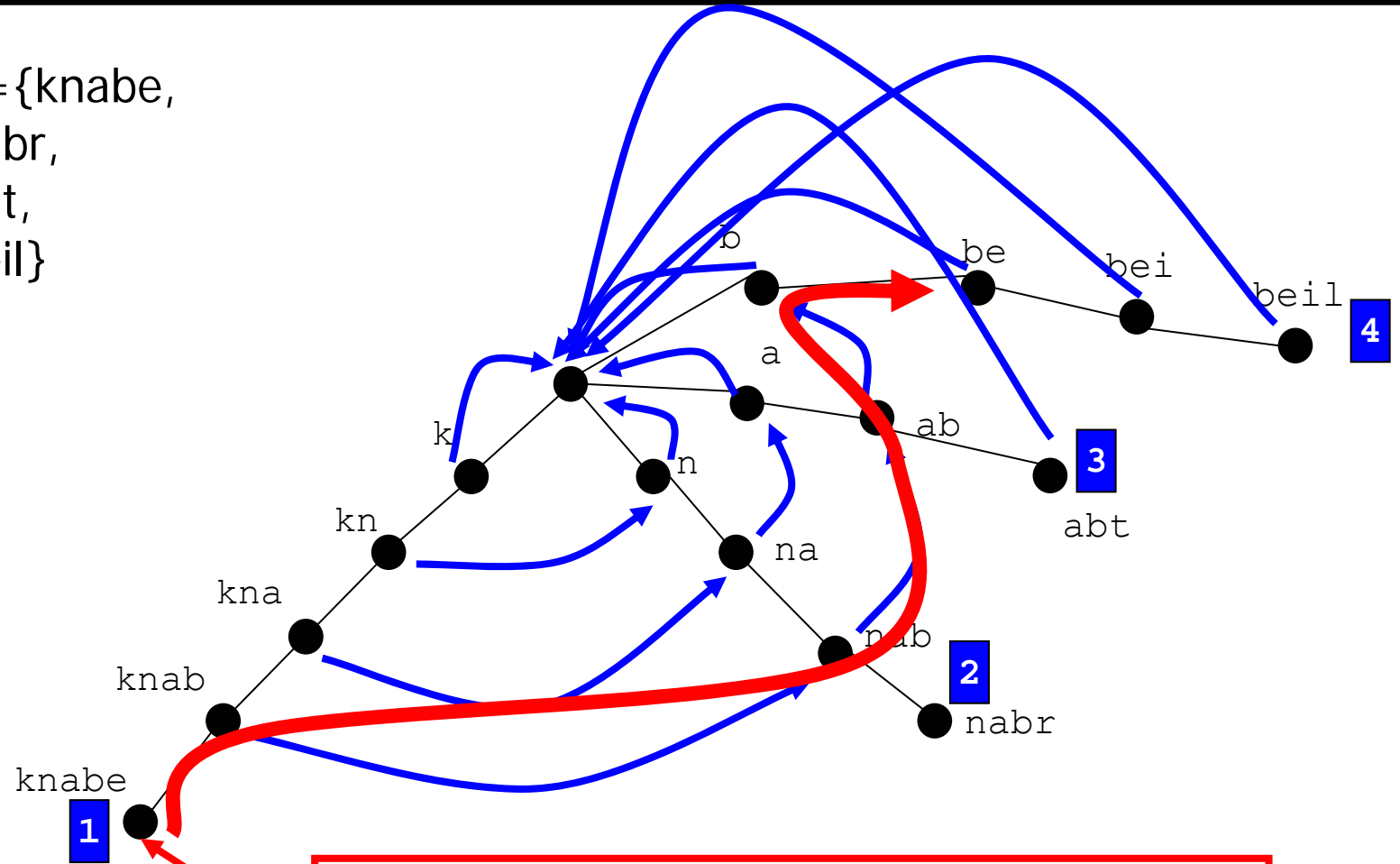
- Induktionsanfang
 - Für jeden Knoten mit $\text{depth}(k)=1$ sei $\text{fl}(k)=\text{root}(K)$
- Induktionsschritt von $i-1$ zu i
 - Seien alle Failure Links von Knoten l mit $\text{depth}(l) < i$, bekannt
 - $\forall k \in K$ mit $\text{depth}(k)=i$
 - Sei k' der Vater von k und x das Label der Kante (k',k)
 - Jedes Suffix von $\text{label}(k')$ wird durch x verlängert
 - Alle Präfixe, die identisch zu einem Suffix von $\text{label}(k')$ sind, erreichen wir durch Traversieren von Failure Links von k' aus
 - Und damit auch das längste
 - Uns interessieren die, die wir durch x verlängern können
 - Falls die Failure Links bei Root ankommen, müssen wir auch dort nach x suchen

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 der Kante zu $\text{fl}(k') = v$
 - Wenn es eine Kante (v, v') mit Label x gibt: $\text{fl}(k) = v'$
 - Wenn nicht
 - Wenn $v = \text{root}(K)$, $\text{fl}(k) = \text{root}$
 - Sonst folge Kante $\text{fl}(v) = v''$ und wiederhole rekursiv

Beispiel

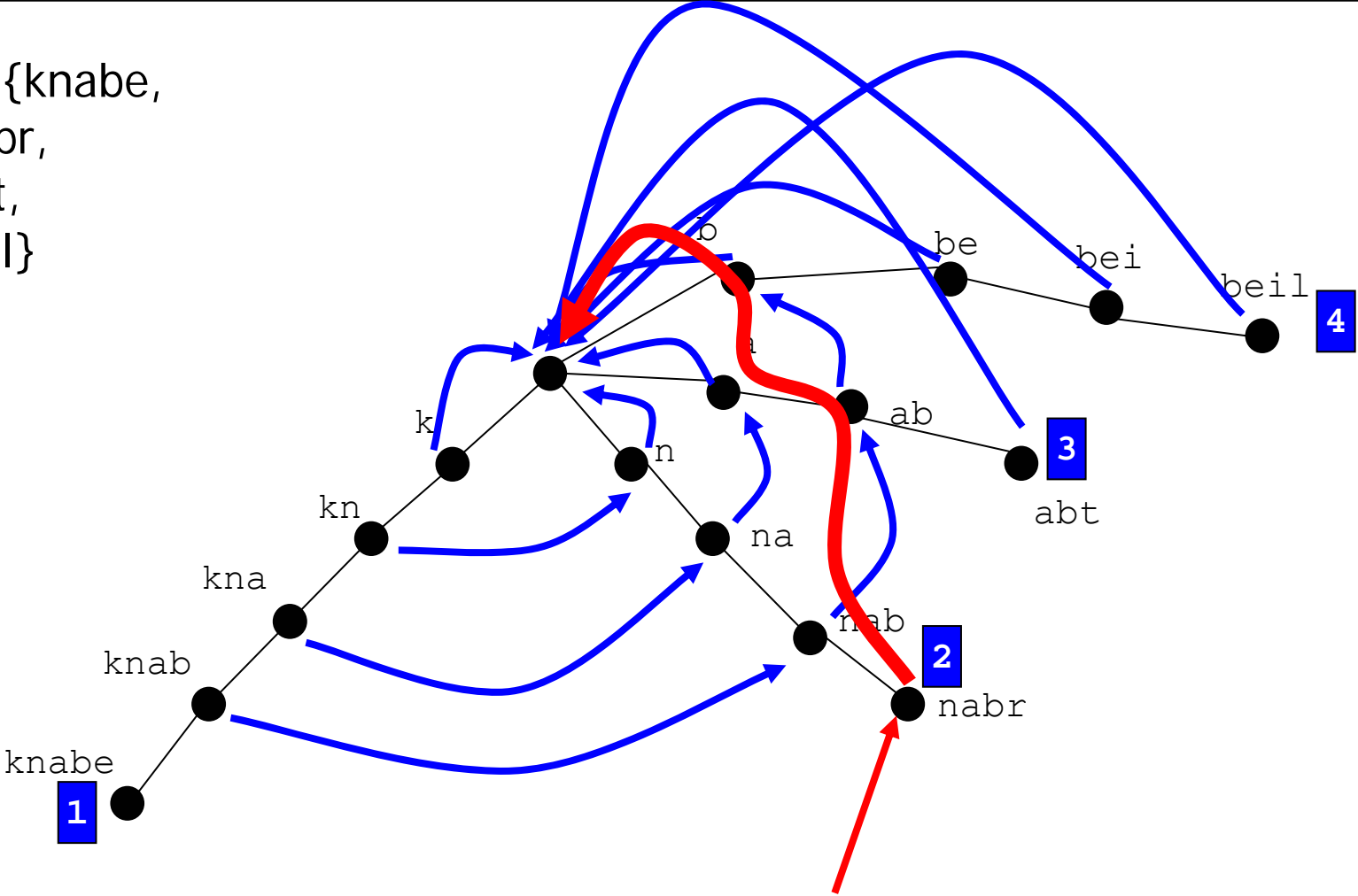
$P = \{ \text{knabe,} \\ \text{nabr,} \\ \text{abt,} \\ \text{beil} \}$



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

Beispiel

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



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: $O(n)$

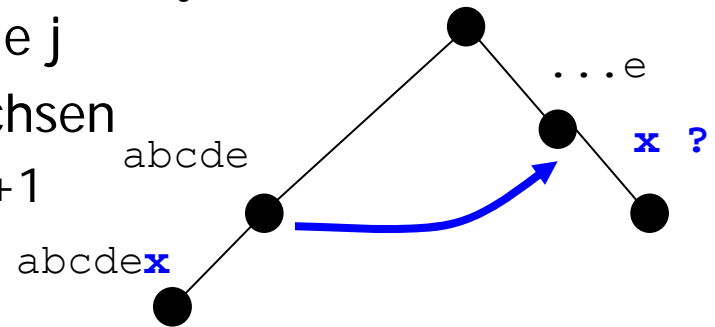
Beweisidee

```
...
while (v≠root(K)) and (not exists edge (v,v') with label(v,v')=x)
    v = fl(v);           // Follow failure link
end while;
...
```

- Es zählt nur die WHILE Schleife, alles andere ist konstant
- Die wird $O(n)$ mal gestartet (für jeden Knoten in K)
- Wir zeigen, dass dabei **insgesamt höchstens $O(n)$ Sprünge** durchgeführt werden
- Dazu betrachten wir den Pfad eines einzelnen P_i und zeigen, dass auf dem Pfad nur $O(|P_i|)$ Sprünge erfolgen
 - Zusammen ist das dann $O(n)$
 - Übrigens: Wir überschätzen die tatsächliche Zahl, denn gleiche Präfixe sind kompakt repräsentiert

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: $\text{length}(v_j) \leq \text{length}(v_{j-1}) + 1$



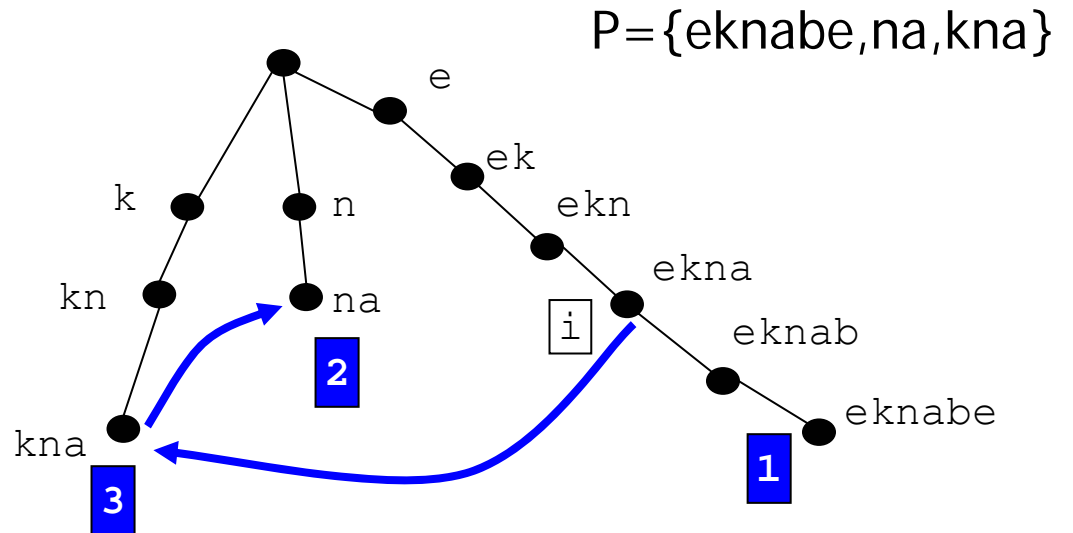
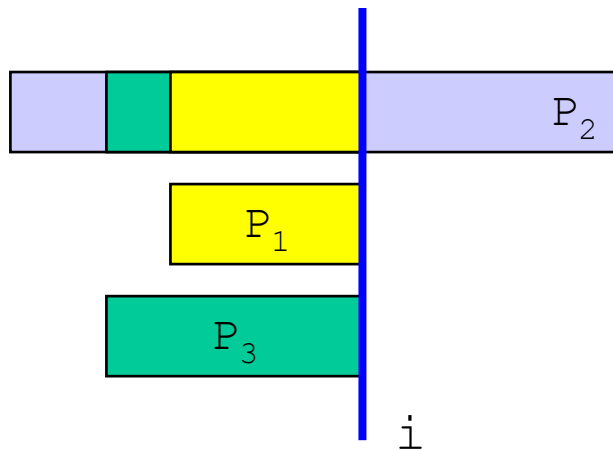
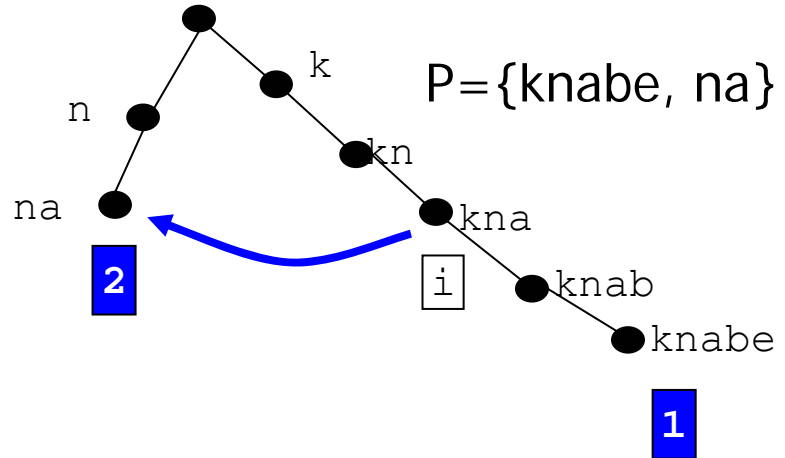
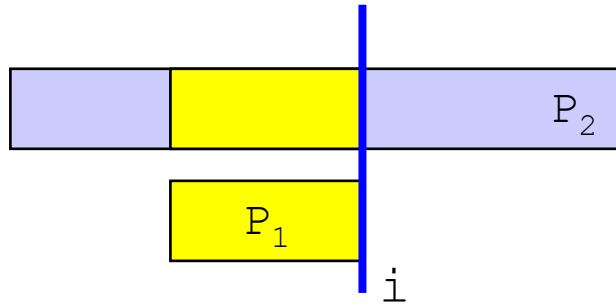
- $\text{length}(v_j)$ wird also höchstens t mal um 1 größer über alle Knoten v_j
- $\text{length}(v_j)$ schrumpft auch mit steigendem j
 - Bei jedem Sprung in der WHILE Schleife um mindestens 1
- Zusammen
 - Mit jedem Sprung 1 weniger, nie kleiner 0, insgesamt nur t mal 1 mehr
 - Also kann es maximal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

-
- Jetzt können wir Failure Links in linearer Zeit berechnen
 - Es fehlt noch die Behandlung der übersehenen Matches

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 von 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
 - $fl(P_2(i)) = P_1$
 - Sonst 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



Folgerung

- Wenn wir also von einem Knoten k ausgehen
 - Und den Pfad der Failure Links verfolgen
 - Und dabei auf einen markierten Knoten k' treffen
 - Dann ist das **Pattern, das k' entspricht, in T enthalten**
- Auch die Umkehrung gilt
 - **Alle enthaltenen Pattern werden so gefunden (Präfix eines Suffix)**
- Problem: Wir müssen das für jeden Knoten auf unserem Weg machen
 - Nicht nur bei markierten Knoten
 - Ohne dabei unsere Komplexität schlechter werden zu lassen
- Wie geht das geschickt?
 - **Preprocessing und abkürzen**

Output Links

- Definition

- Der *Output Link* eines Knoten k , $out(k)$, ist der Knoten k' für den gilt
 - k' ist markiert
 - k' ist der erste markierte Knoten, der in der Kette der Failure Links von k aus erreicht wird

- Bemerkungen

- Nicht alle Knoten haben Output Links
- Output Links deuten immer auf kürzere Pattern
- Wir können die Output Links beim Breadth-First Traversieren des Baumes in konstanter Zeit mitbestimmen

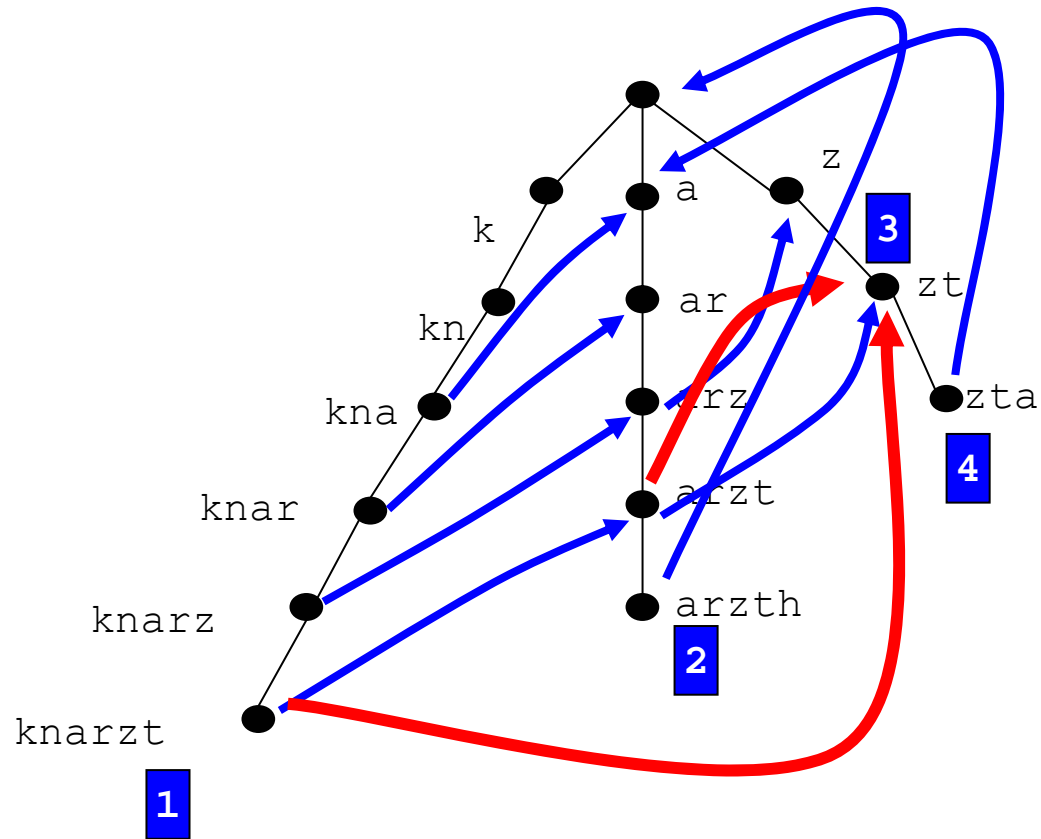
Failure Links und Output Links

```
// 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';
        if mark(v')≠NULL then out(k)=v'; else out(k)=NULL;
    else
        fl(k) = root(K);
        out(k) = NULL;
    else
        fl(k) = v';          // Continuation of prefix with x
        If mark(v') ≠ NULL then
            out(k) := v';    // Obviously the closest marked node
        else
            out(k) = out(v');
        end if;
    end if;
end if;
```

Keine Änderung der Komplexität – nur **konstante Operationen**

Breadth-First Konstruktion von Output Links

$P = \{ \text{k narzt,} \\ \text{arzth,} \\ \text{zt,} \\ \text{zta} \}$



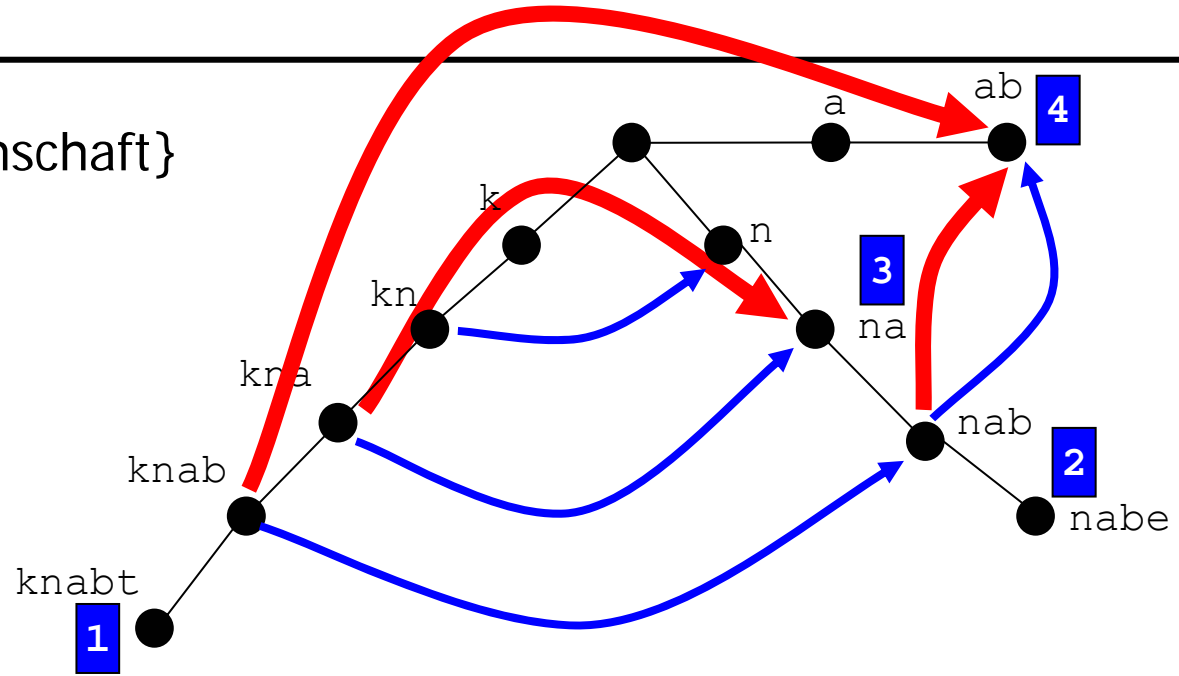
Suchphase mit Output Links

- 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

Beispiel

$T = \{\text{knabenschaft}\}$

$P = \{\text{knabt}, \text{nabe}, \text{na}, \text{ab}\}$



1. Algorithmus matcht KNA ...
 - Output Link folgen ergibt Treffer mit P_3
2. ... matcht weiter bis KNAB
 - Output Link folgen ergibt Treffer mit P_4
3. „b“ ist der letzte Match – Failure Link zu NAB
4. Erweiterung zu NABE – Treffer mit P_2



Kompletter Algorithmus

```
j := 1;           // Next comparison in T
k := root(K);    // Root node of keyword tree
repeat
  while exists edge (k,k') with label(k,k')=T(j)
    if mark(k') ≠ NULL then
      report mark(k');
    end if;
    z = out(k');
    while (z ≠ NULL)           // Check output links
      report mark(z);         // Found a match
      z = out(z);             // Recursion
    end if;
    k := k';                  // Down the tree
    j := j+1;                 // Check next character
  end while;
  if k=root(K) then          // Immediate mismatch: move on in T
    j := j+1;
  else
    k := fl(k);              // Follow the failure link
  end if;
end repeat;
```

Komplexität

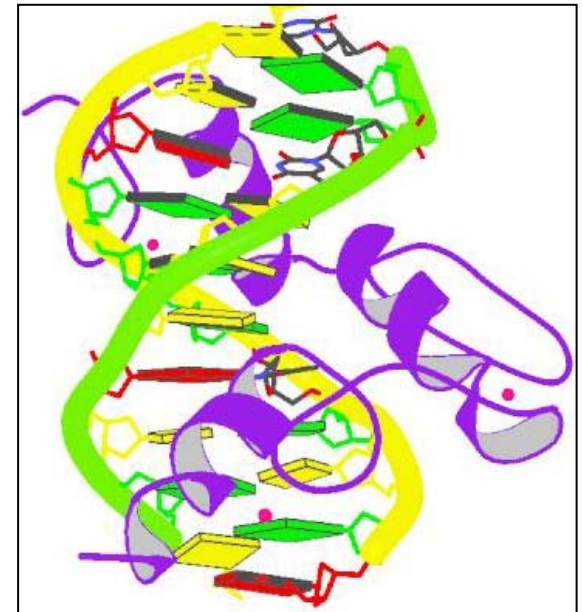
- Komplexität der Suchphase
 - Sei k die Gesamtzahl an Matches von Pattern aus P in T
 - Die innere WHILE Schleife wird maximal k -mal passiert
 - Also: $O(m+k)$
- Gesamtkomplexität
 - Berechnung Keyword Tree für P $O(n)$ (trivial)
 - Berechnung Failure Links $O(n)$ (BF)
 - Dabei auch Berechnung der Output Links
 - Suche mit Failure/Output Links $O(m+k)$
- Zusammen $O(n+m+k)$

Variationen des Themas

- Suche mit Wildcards *
 - Clevere Anwendung von Aho-Corasick
- Suche mit regulären Ausdrücken
 - Wiederholung aus PI-3 (?)

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, l_1,\dots,l_s 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



Beispiel

$P = \{AB^*DA^*\}$

12345678

$T = \{TABTABDADAZA\}$

123456789012

$P_1 = AB, \quad l_1 = 1$

$P_2 = DA, \quad l_2 = 5$

$P_3 = A, \quad l_3 = 8$

$C = [000000000000]$

P_1	an	$j=2,$	$z=2$	\Rightarrow	010000000000
P_1	an	$j=5,$	$z=5$	\Rightarrow	010010000000
P_2	an	$j=7,$	$z=3$	\Rightarrow	011010000000
P_2	an	$j=9,$	$z=5$	\Rightarrow	011020000000
P_3	an	$j=2,$	$z=-5$	\Rightarrow	011020000000
P_3	an	$j=5,$	$z=-2$	\Rightarrow	011020000000
P_3	an	$j=8,$	$z=1$	\Rightarrow	111020000000
P_3	an	$j=10,$	$z=3$	\Rightarrow	112020000000
P_3	an	$j=12,$	$z=5$	\Rightarrow	112030000000

^ **Treffer**

Komplexität

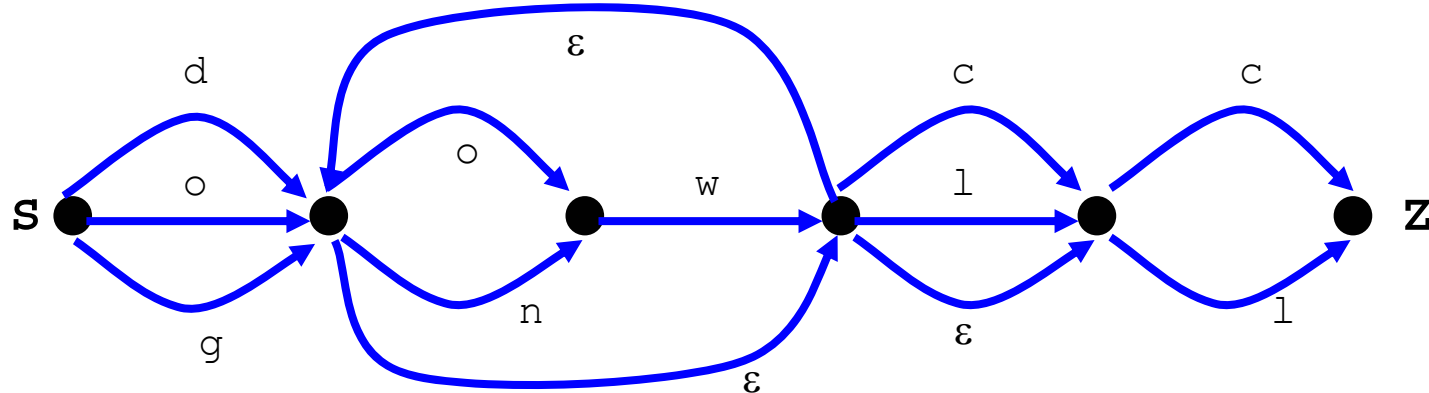
- $O((n-s) + m + k)$
 - Gesamtlänge aller Pattern ist $n-s$
 - Array wird in konstanter Zeit während der AC Suche aktualisiert
 - Immer wenn dabei $C[z]=s$, gib z aus.

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 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: $(d|o|g)((n|o)w)^*(c|l|\varepsilon)(c|l)$
 - Matcht z.B. dnwnwowc, ol, gowll, ...



Definitionen

- Definition
 - Sei P ein regulärer Ausdruck. Dann ist $G(P)$ der dazugehörige endliche Automat
 - Ein Substring T' von T *matcht mit P* , wenn T' durch einen Pfad in $G(P)$ von S nach Z spezifiziert wird
- Komplexität der Konstruktion von $G(P)$ ist linear
- Die Menge aller Strings, die von $G(P)$ akzeptiert werden, bilden die Sprache zu P
- Gesucht: Algorithmus, um alle T' in T zu finden, die mit P matchen

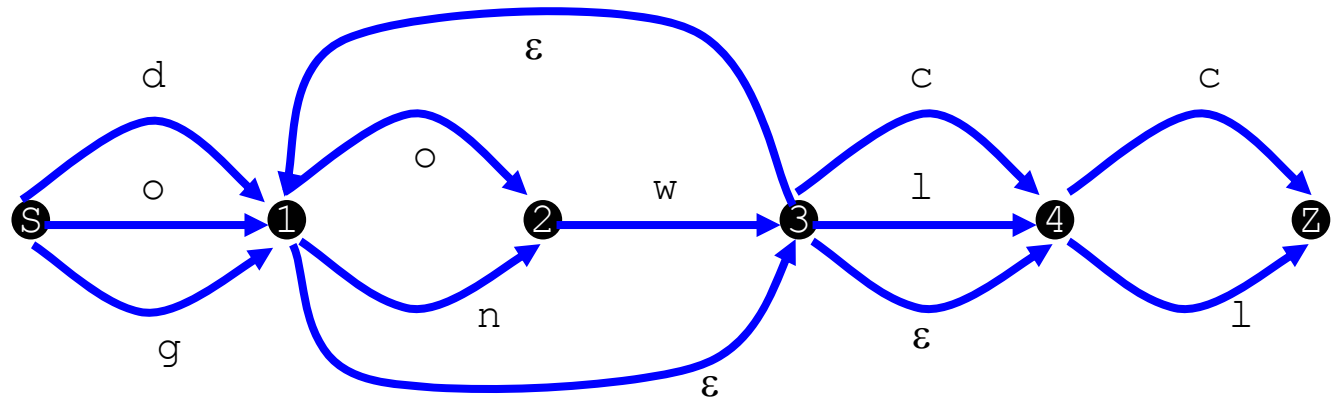
Algorithmusidee

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

Beispiel



- Pattern: $(d|o|g)((n|o)w)^*(c|l|\epsilon)(c|l)$

- 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 $G(p)$ ist
 - N ist eine Menge, keine Multimenge!
 - Also ist dieser Schritt $O(e)$
- Wir berechnen m Mengen $N(1) \dots N(m)$
- Also $O(m \cdot e)$
- Beobachtung
 - Ein regulärer Ausdruck mit n Symbolen hat höchstens n ε -Kanten
 - Sonst kann er minimiert werden
- Zusammen: **$O(m \cdot n)$**

Zusammenfassung

- Aho-Corasick Algorithmus ist Erweiterung des KMP auf simultane Suche nach mehreren Pattern
- Erstaunlich gute Komplexität – Ausnutzung der Gemeinsamkeiten in einem Keyword Tree
- Einfacher Trick führt zu linearem Suchen mit Wildcards
- Suche mit regulären Ausdrücken ist aber quadratisch