Algorithmische Bioinformatik

Suche nach mehreren Mustern:
Keyword-Trees / Aho-Corasick Algorithmus

Ulf Leser

Wissensmanagement in der Bioinformatik

Ziele

- Verständnis des Aho-Corasick-Algorithmus in allen Einzelschritten
- Verhältnis zum KMP-Algorithmus
- Ein weiterer Trick zum Trade-Off Zeit/Platz

Inhalt dieser Vorlesung

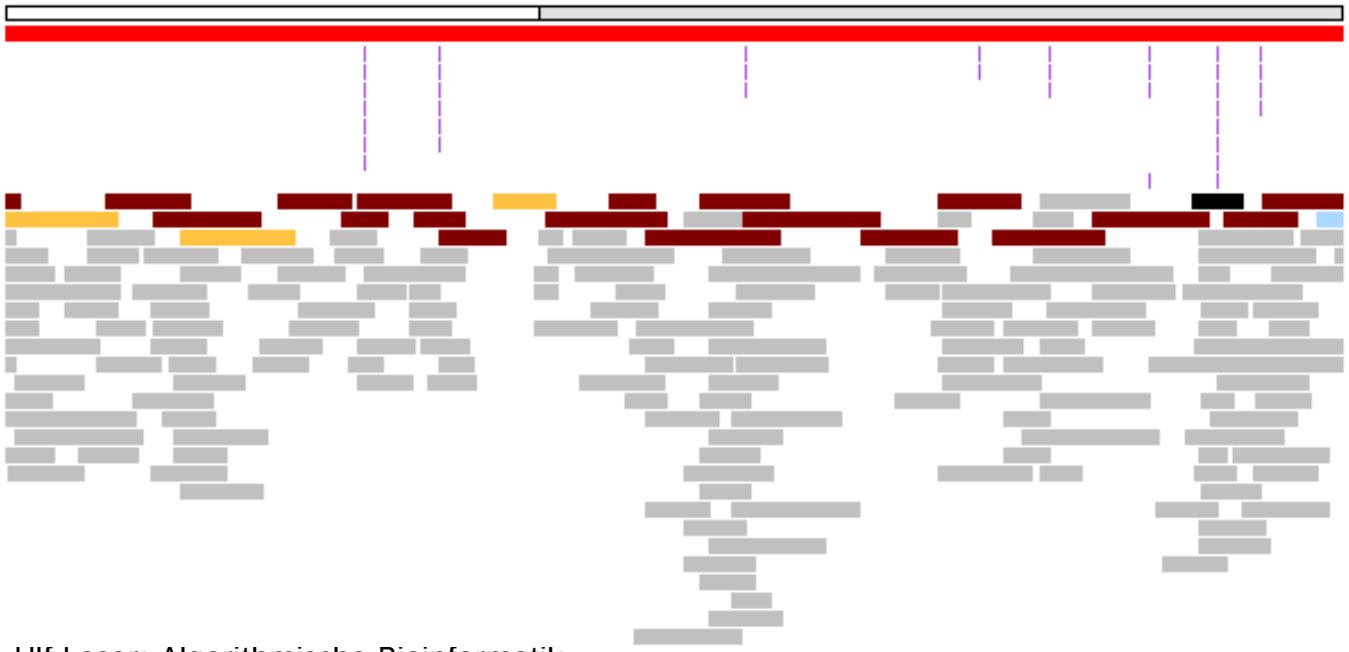
- Suche nach mehreren Mustern
- Keyword-Trees
  - Definition
  - Failure Links zur Suche
  - Konstruktion von Failure-Links in linearer Zeit
  - Output-Links
  - Ein cleverer Trick
- Suche mit Wildcards

Suche nach mehreren Mustern

- Bisher: Suche eines Pattern P in einem String T
- Jetzt: Gleichzeitige Suche nach mehreren Pattern $P_1, \ldots, P_z$
- Motivation
  - Suchmaschinen: „xyz AND abc“
    - Achtung: Das macht man in der Praxis nicht mit Keyword-Trees
  - Lokalisierung einer neuen Sequenz auf einer STS/EST Karte
  - Suche nach typischen Primern / Vektoren in einer DNA
  - Suche nach Schnittstellen verschiedener Restriktionsenzyme
  - Suche nach Bindungsstellen von Transkriptionsfaktoren
  - ...

Mapping und Sequenzierung

- Mapping-basierter Ansatz zur Genomsequenzierung
  - Zerlegung in Bruchstücke (Reads)
  - Bestimmung aller Überlappungen
  - Bestimmung des Minimum Tiling Paths
  - Sequenzierung nur dieser Reads



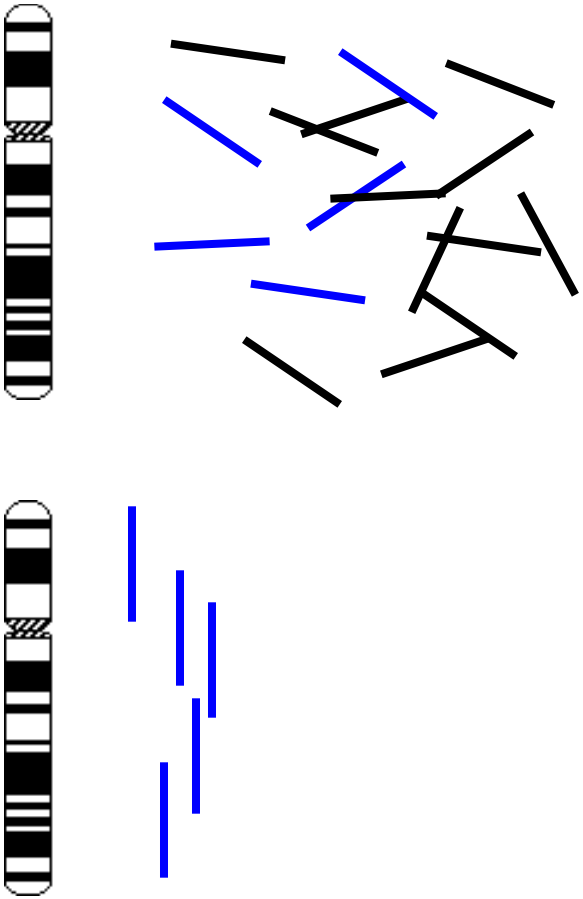
BAC Map, Chr.X 1-3 MB; http://www.sanger.ac.uk

DNA Marker

- Eindeutig im Genom lokalisierbare „Punkte“
- Können polymorph sein
  - Z.B. definierter Anfang und definiertes Ende, aber variables, bei verschiedenen Individuen unterschiedlich langes, Zwischenstück
- STS: Sequence-tagged Sites
  - Genomweit eindeutige Sequenzen
  - Länge 300-500 Basenpaare
- EST: Expressed Sequence Tags
  - Genomweit eindeutige kodierende Sequenzen
  - Länge 300-500 Basenpaare

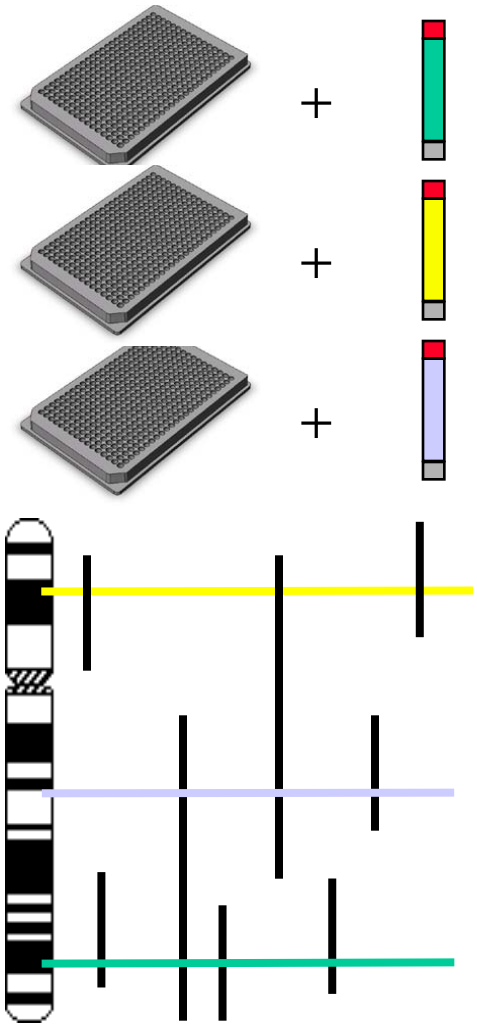
Kartierungstechniken

- Wie findet man Überlappungen?
  - Zwei Bruchstücke überlappen sich
    - Wenn Sequenzen bekannt: Alignierung
    - Experimentell: Hybridisierung
  - Zwei Bruchstücke enthalten den gleichen Marker
    - Wenn Sequenzen bekannt: Electronic PCR
    - Experimentell: PCR

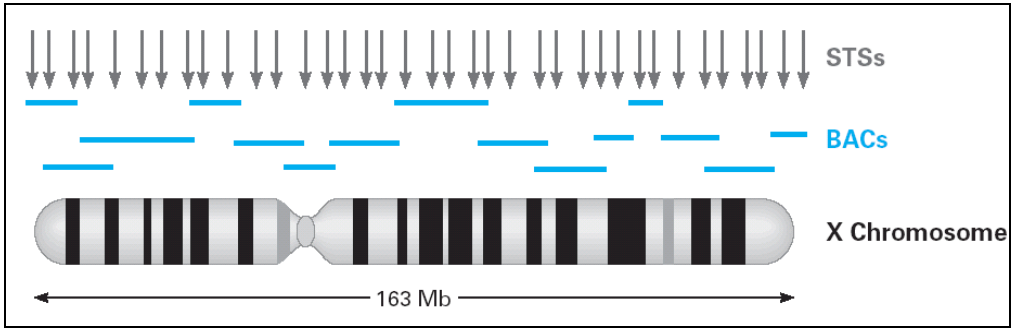


STS Content Mapping

- Gegeben: Clone, STS mit Position in Genom
- Welcher Clone enthält welche STS?
  - Experimentell: PCR mit jedem STS
  - „Anchoring“ der Clone
  - Bei dichter Abdeckung des Genoms kann man die wahrscheinlichste Reihenfolge der Clone berechnen
    - Vorsicht: Fehler passieren immer
  - STS Sequenz muss nicht bekannt sein



Electronic PCR: STS Kartierung



- Electronic PCR
  - Gegeben eine Menge von STS in einer Karte (Position und Sequenz) und eine Sequenz (Clone)
  - Elektronisch: Welche STS liegen in welcher Clonesequenez?
  - Gleichzeitige Suche von Tausenden (Chromosom bekannt) oder Millionen von STS
- „z“ (P_1,…P_z) ist ein wichtiger Faktor für Komplexität

Inhalt dieser Vorlesung

- Motivation: Suche nach mehreren Mustern
- Keyword-Trees
  - Definition
  - Failure Links zur Suche
  - Konstruktion von Failure-Links in linearer Zeit
  - Output-Links
  - Ein cleverer Trick
- Suche mit Wildcards

Erster Versuch

- Sei $P=\{P_1,P_2,\ldots,P_z\}$, $n=|P_1|+|P_2|+\ldots+|P_z|$
  - Vorsicht: Dies ist ein anderes „n“ als bisher
- KMP braucht $O(m+|P_i|)$ für Suche mit einem Pattern P_i
  - $O(|P_i|)$ Preprocessing, $O(m)$ Suche
- Naive Erweiterung auf z Pattern
  - Preprocessing für jedes Pattern
    - $O(|P_1|+|P_2|+\ldots+|P_z|) = O(n)$
  - Sequentielle Suche aller Pattern in T: $O(z*m)$
  - Zusammen: $O(n + z*m)$

Grundidee

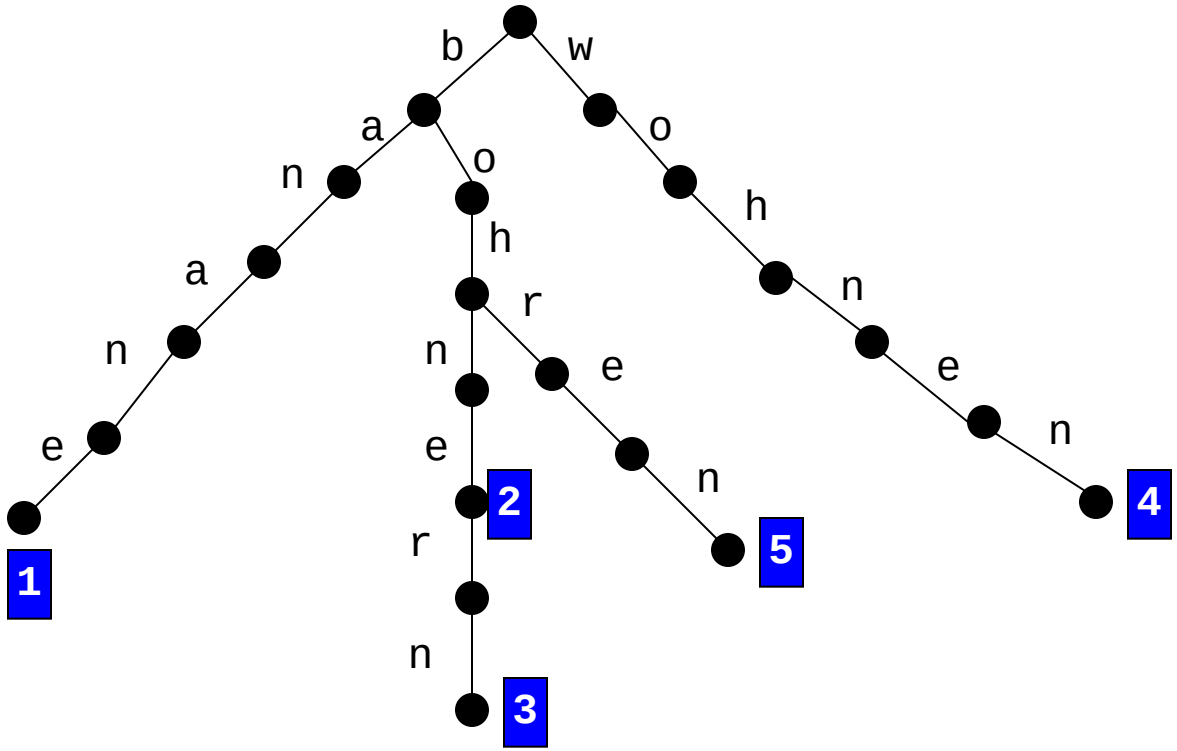
- Wir werden das auf O(m+n+k) verbessern
  - Bedeutung von k später
- Grundidee
  - Pattern aus P haben i.d.R. Zeichen / Substrings gemeinsam
  - Die wir von links nach rechts suchen, interessieren uns insbesondere gemeinsame Präfixe
  - Gesucht: Datenstruktur zur Repräsentation gemeinsamer Präfixe
  - Mit dieser Datenstruktur gleichzeitig nach allen Pattern suchen
    - Performanz soll nur von „Größe“ der Datenstruktur abhängen
    - Hängt indirekt von der Zahl/Länge der Pattern ab

Keyword Trees (Prefix Trees, Tries)

- Definition
  - *P eine Menge von Pattern. Ein Keyword Tree für P ist ein Baum mit:*
    - *Jede Kante ist mit genau einem Zeichen beschriftet (Label)*
    - *Wenn ein Knoten mehrere Kinder hat, so sind die Kanten zu diesen Kindern mit unterschiedlichen Zeichen beschriftet*
    - *Das Label eines Knoten k, label(k), ist die Konkatenation der Kantenlabel auf dem Pfad von der Wurzel zu k*
    - *P_i entspricht einem Knoten k gdw label(k)=P_i*
      - *Wir markieren k mit i, wenn k dem Pattern P_i entspricht*
      - *Dies bezeichnen wir mit mark(k)=i*
      - *Bei nicht-markierten Knoten ist mark(k)=UNDEF*
    - *Jedes Blatt entspricht mindestens einem P_i*
      - *Also: Jedes Blatt k ist markiert, mark(k)≠UNDEF*
- Folgerungen
  - Jeder Pfad ab der Wurzel ist eindeutig
  - Baum ist minimal für P; es gibt keine „überflüssigen“ Knoten/Kanten

Beispiel

P={banane, bohne, bohnern, wohnen, bohren}



Konstruktion

- Komplexität der Konstruktion?
  - Wir zählen das Hinzufügen von Knoten/Kanten
- Konstruktion in O(n)
  - Beginne mit P_1
    - Baumkonstruktion dauert $O(|P_1|)$
  - Betrachte P_2. Laufe im Baum das Präfix von P_2 ab
    - … bis Mismatch an Position l auftritt. Dann eine neue Abzweigung mit Beschriftung $P_2[l]$ einfügen und mit $P_2[l+1..]$ verlängern
    - … bis P_2 komplett aufgetreten ist. Dann „2“ an Endknoten schreiben
    - Braucht zusammen $O(|P_2|)$
  - Induktion über P_3 – P_z zeigt die Behauptung
- Beachte: Alle Pfade sind eindeutig

Naive Verwendung

- Naive Verwendung eines Keyword Tree
  - Baue Keyword Tree K für Patternmenge P in O(n)
  - Iteriere über Positionen i in T
    - Laufe Präfix von T[i..] in K ab
    - Wenn markierter Knoten passiert wird, melde das Pattern
    - Ist für Position $l \geq i$ kein Pfad in K vorhanden: Starte erneut ab Position i+1 in T und Wurzel von K
  - Gesamtkomplexität: $O(n+m*n_{max})$, mit $n_{max}=\max(|P_i|)=depth(K)$
    - Vielleicht was gewonnen (wenn $n_{max}<z$)
- Vorgehen entspricht dem naiven Suchalgorithmus
  - Wir nutzen nicht aus, was wir zwischen i und l gerade gesehen haben (und wo das noch mal in K vorkommt)
  - Gesucht: KMP für Bäume

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"*
  - sp_i erlaubt, kein Zeichen von T zweimal positiv zu matchen
- Übertragung: Aho-Corasick Algorithmus (AC)
  - Statt sp_i im Pattern bauen wir Failure Links im Keyword Tree
  - Failure Links zeigen von einem Knoten X im Baum auf einen Knoten Y, dessen Label das längste echte Suffix des Labels von X ist
  - Wenn es einen Mismatch nach X gibt, springen wir zu passenden Suffixen und matchen dort weiter

Failure Links

- Definition

  Sei K ein Keyword Tree für P und k ein Knoten von K:

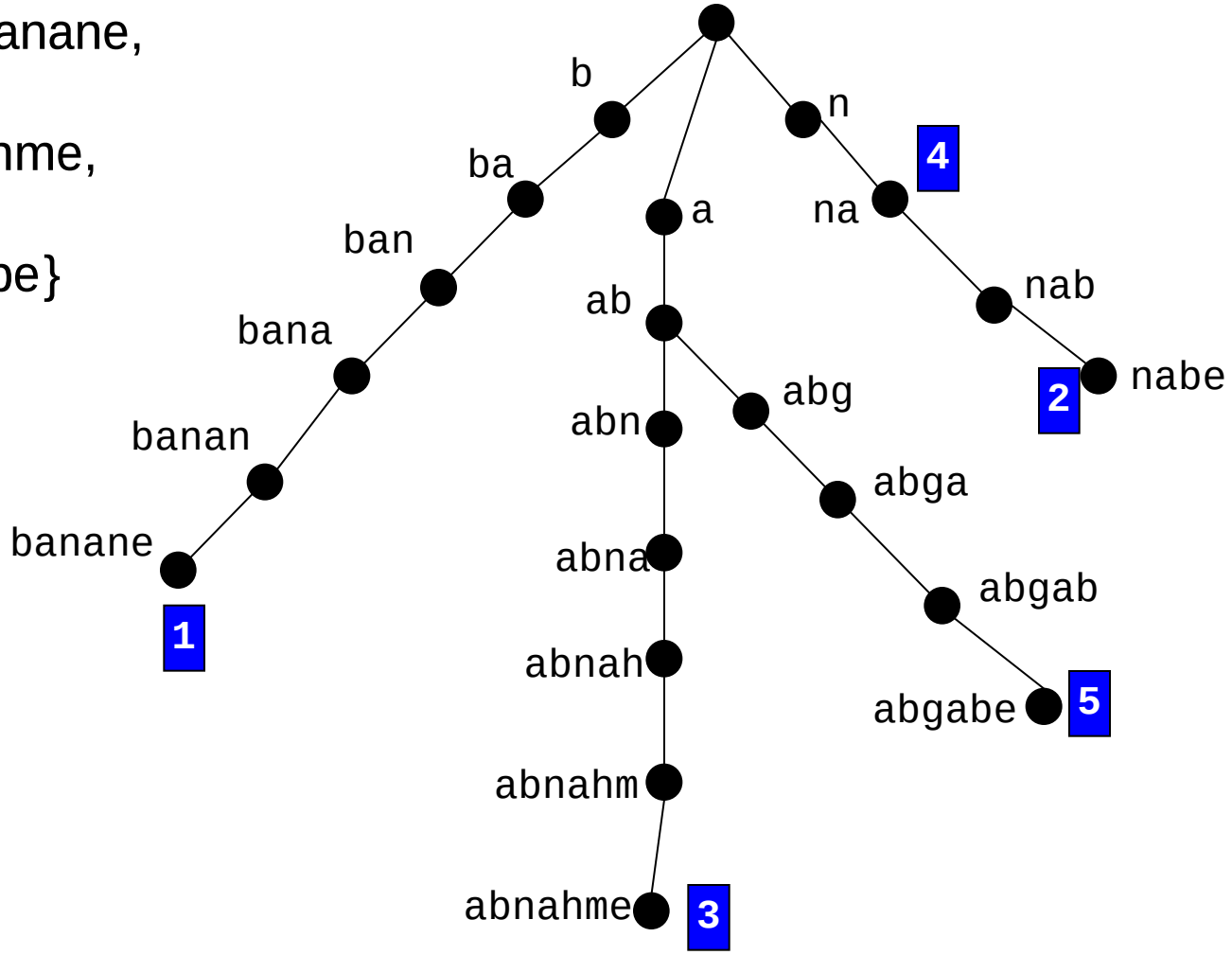
  - *length(k) ist die Länge des längsten echten Suffix von label(k), das auch Präfix mindestens eines Patterns aus P ist*
    - *Gibt es kein solches Suffix, dann sei length(k)=0*
  - *fl(k) ist der Knoten für den gilt:*

    $$label(fl(k)) = label(k)[\ |label(k)|-length(k)+1\ ..\ |label(k)|\ ]$$

    - *Für Knoten k mit length(k)=0 gilt: fl(k)=root*
  - *Die Verbindung (k, fl(k)) heißt Failure Link von k*
- Beachte
  - label(fl(k)) ist genau das „längste echte Suffix“ von label(k)
  - fl(k) ist per Konstruktion von K eindeutig

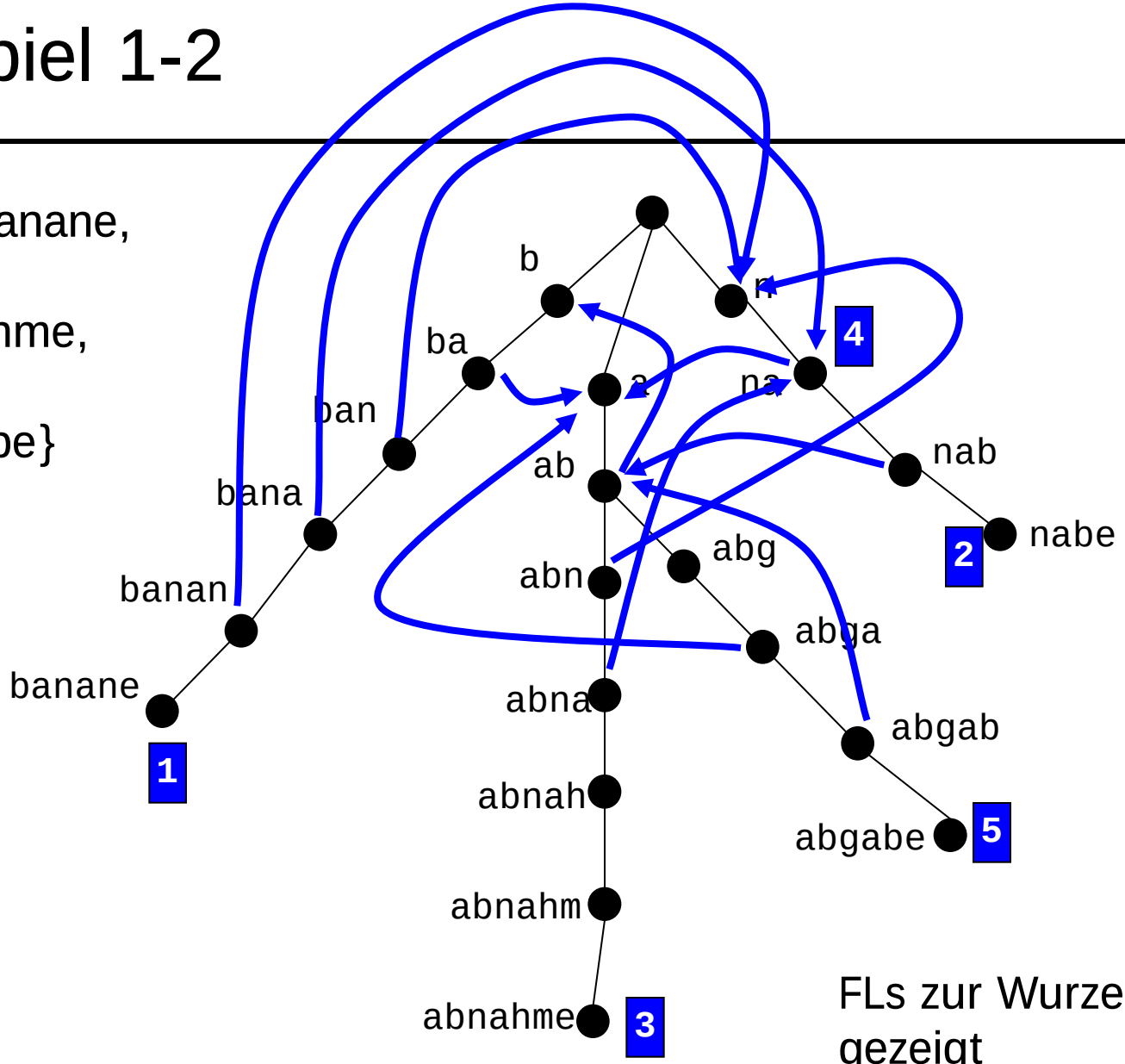
Beispiel 1-1 - Tafel

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



Beispiel 1-2

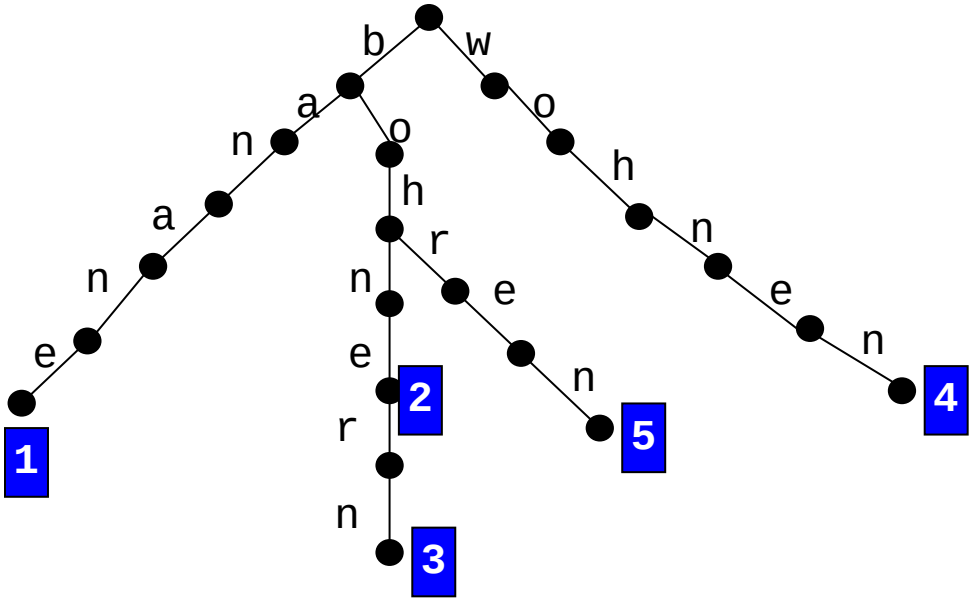
P={banane,
nabe,
abnahme,
na,
abgabe}



FLs zur Wurzel nicht gezeigt

Beispiel 2

P={banane,
bohne,
bohnern,
wohnen,
bohren}



- Alle Failure Links zeigen auf Root
  - Startbuchstaben b und w kommen in keinem Pattern an einer Position $\neq 1$ vor
  - Kein echtes Suffix kann einem Präfix entsprechen

Suchen mit Failure Links

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

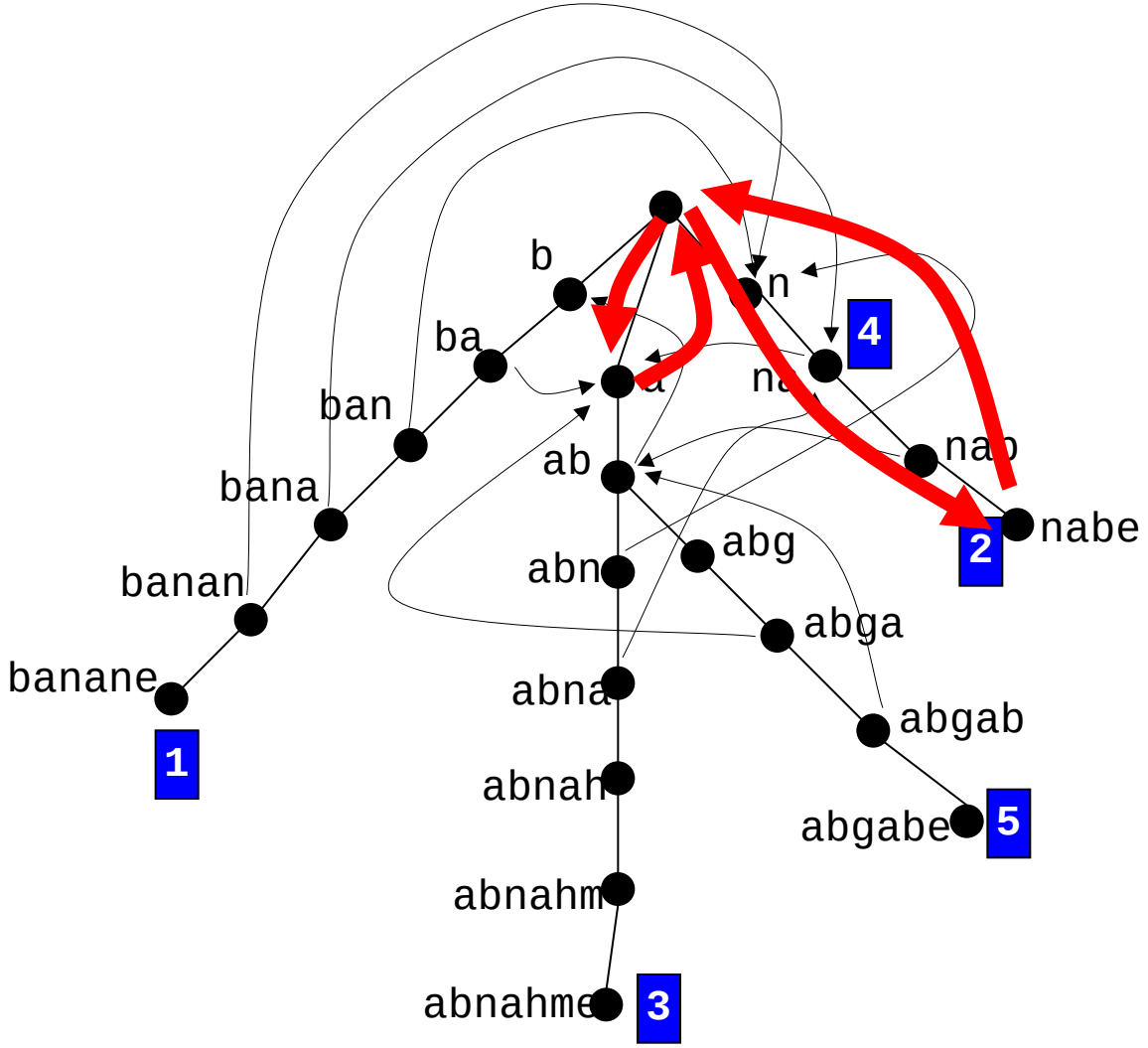
Suchen mit Failure Links

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

Beispiel

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

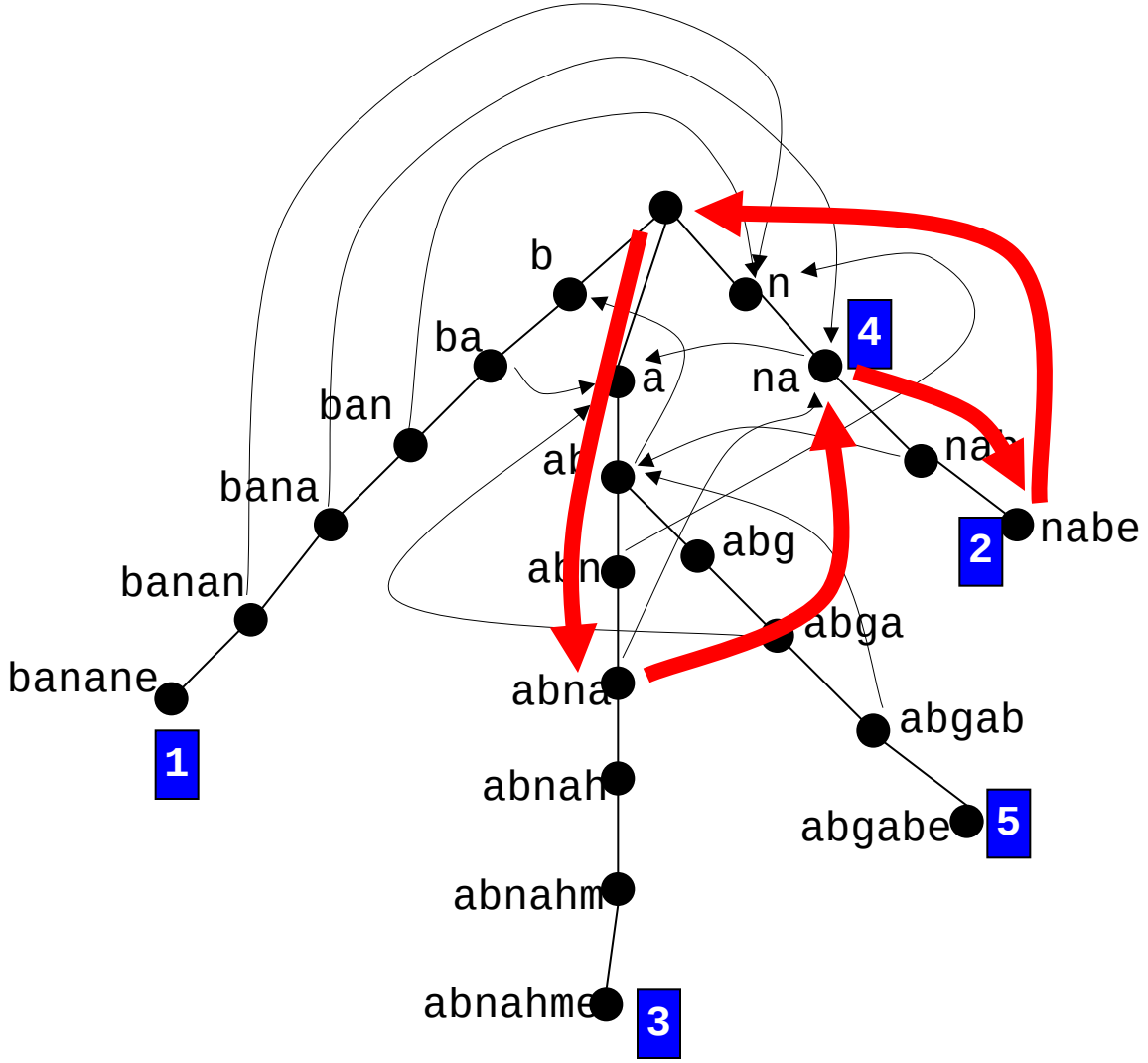
T = radnaben



Beispiel 2

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

T = abnabeln



Algorithmus

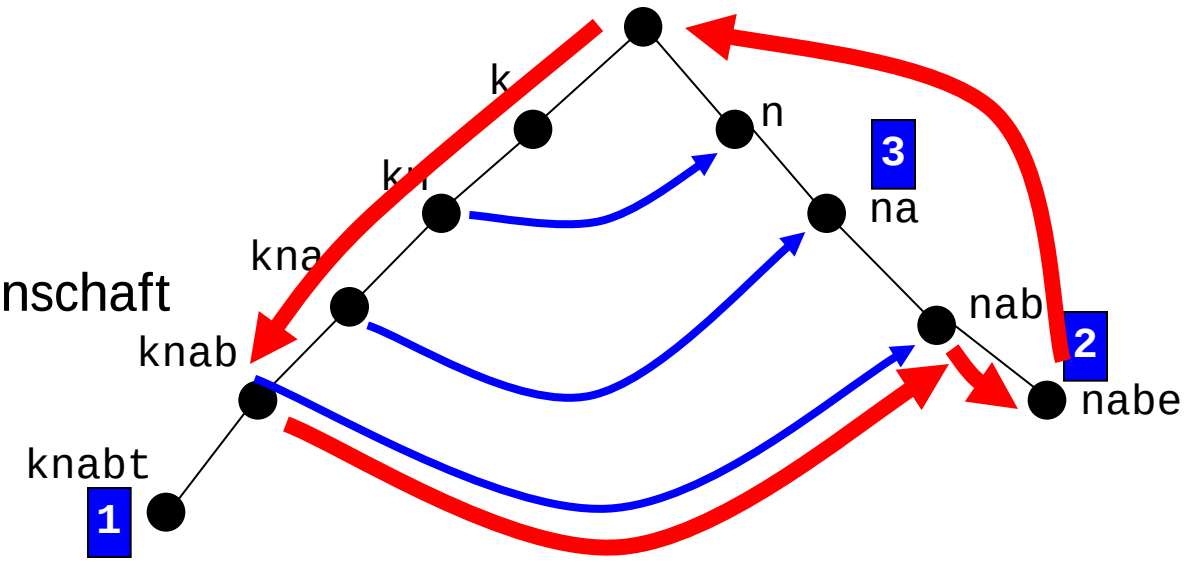
```
i := 1;                          // Next comparison in T
l := 1;                          // Start of pattern in T
k := root(K);                    // Current node in keyword tree
while (i<|T|)
        while exists edge (k,k‘) with label T[i]
                if mark(k‘)≠NULL then
                        report mark(k’) with start l;
                end if;
                k := k‘;         // Down the tree
                i := i+1;        // Check next character
        end while;
        if k=root(K) then        // Immediate mismatch: move on in T
                i := i+1;
                l := l+1;
        else
                k := fl(k);      // Follow the failure link
                l := i-len(k);
        end if;
end;
```

- l nur notwendig, wenn Startposition der Matches verlangt
- Komplexität O(m) (Beweis ähnlich wie bei KMP)

Alles klar?

P={knabt,
nabe,
na}

T = 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!

Inhalt dieser Vorlesung

- Suche nach mehreren Mustern
- Keyword-Trees
  - Definition
  - Failure Links zur Suche
  - Konstruktion von Failure-Links in linearer Zeit
  - Output-Links
  - Ein cleverer Trick
- Suche mit Wildcards

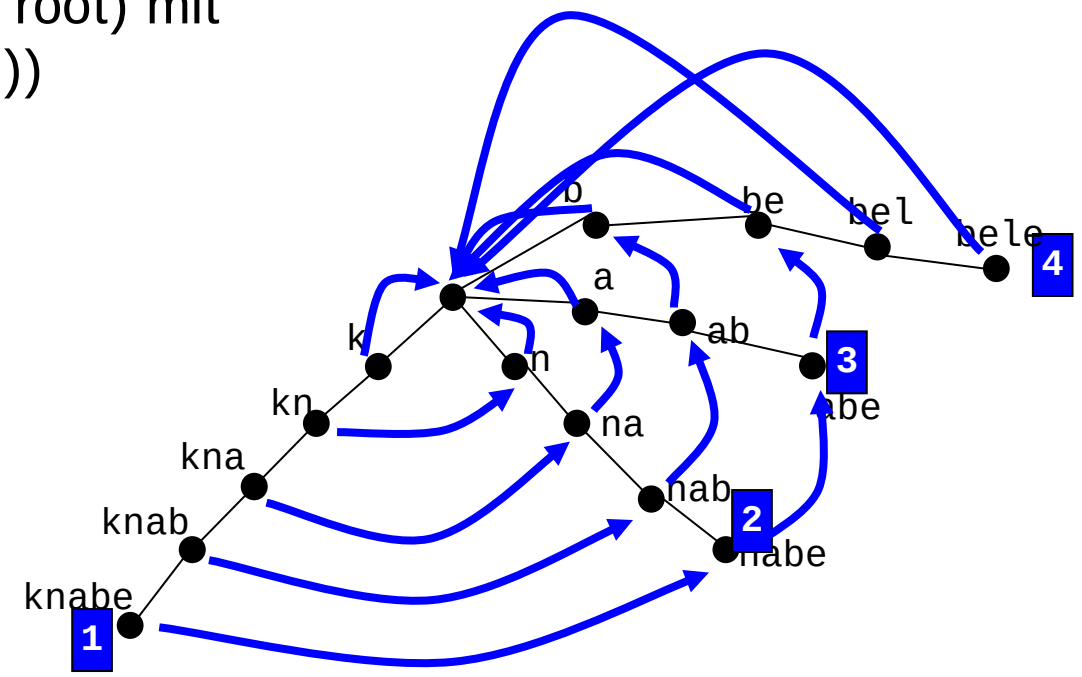
Konstruktion der Failure Links

- Bisher
  - Mit Failure Links ist die Suchphase O(m)
    - Wir ignorieren erstmal das „Übersehen“-Problem
  - Konstruktion des Keyword Trees ist O(n)
- 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) per Breitensuche
    - Beachte: Failure Links zeigen immer zu echten Suffixen
    - D.h., $\forall$k: depth(k) > depth(fl(k))

Beobachtung

- Für Knoten k: Alle Präfixe, die identisch zu einem echten Suffix von label(k) sind, erreichen wir durch Failure Links, und zwar in absteigender Reihenfolge ihrer Länge
  - Das längste Präfix findet man mit fl(k)
  - Alle weiteren (bis root) mit fl(fl(k)), fl(fl(fl(...)))

P={knabe,
nabe,
abe,
bele}



Algorithmusidee

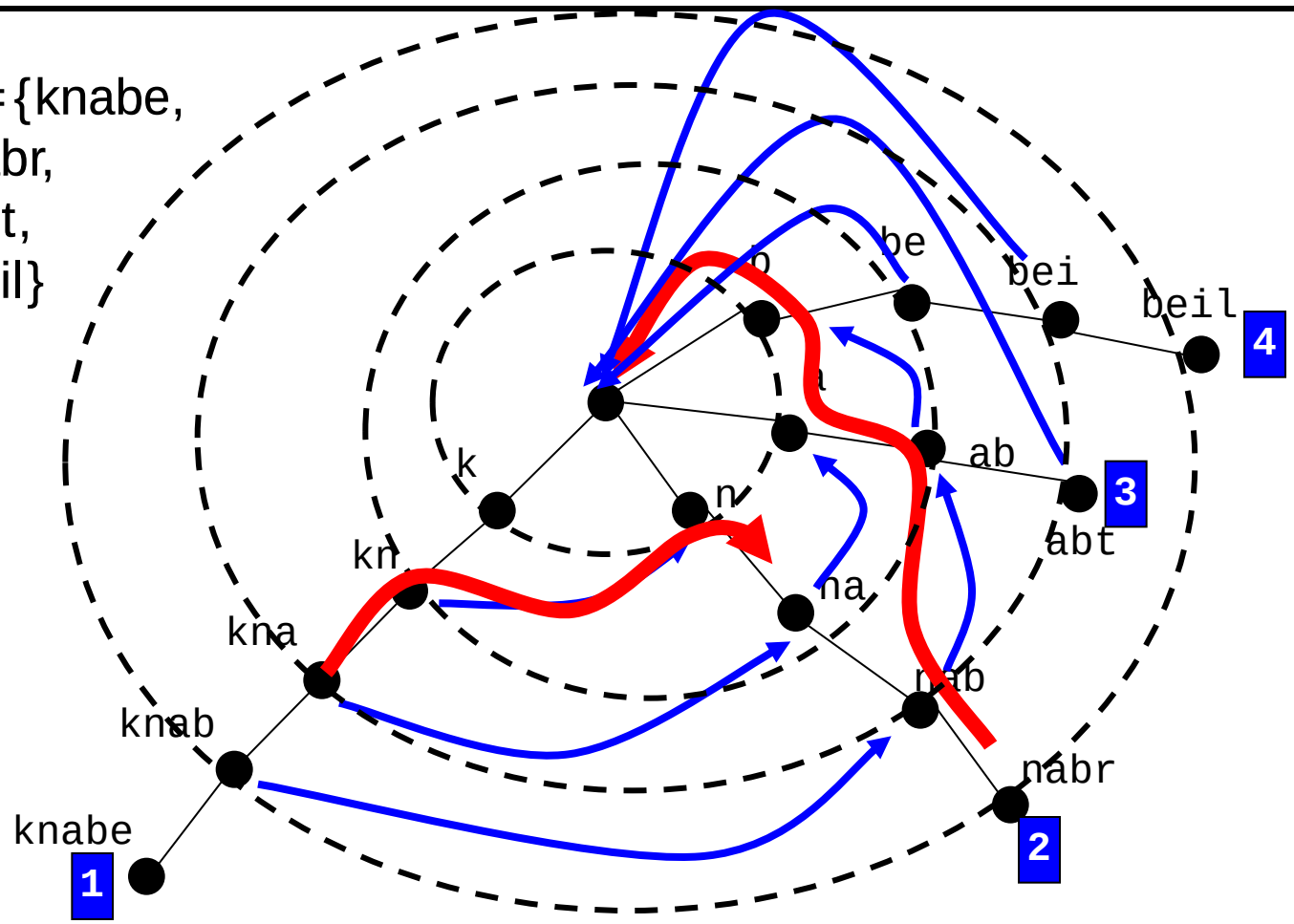
- Induktionsanfang: $\forall$k mit depth(k)=1: fl(k):=root(K)
- Induktionsschritt von i-1 zu i (i: Tiefe der Front im Baum)
  - Seien alle Failure Links von Knoten l mit depth(l)<i bekannt
  - $\forall k \in K$ mit depth(k)=i
    - Sei k‘ der Vater von k und x das Label der Kante (k‘,k)
    - Jedes Suffix von label(k‘) wird durch x verlängert zu einem Suffix von label(k)
    - Alle Präfixe, die identisch zu einem Suffix von 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 wir 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 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 dem Failure Link von k‘ aus zu fl(k‘)=v
      - Wenn es eine Kante (v,v‘) mit Label x gibt: fl(k)=v‘
      - Wenn nicht
        - » Wenn v=root(K), dann fl(k)=root; stop
        - » Sonst: folge Kante fl(v)=v‘‘, suche eine ausgehende Kante mit x ...
        - » Wiederhole rekursiv

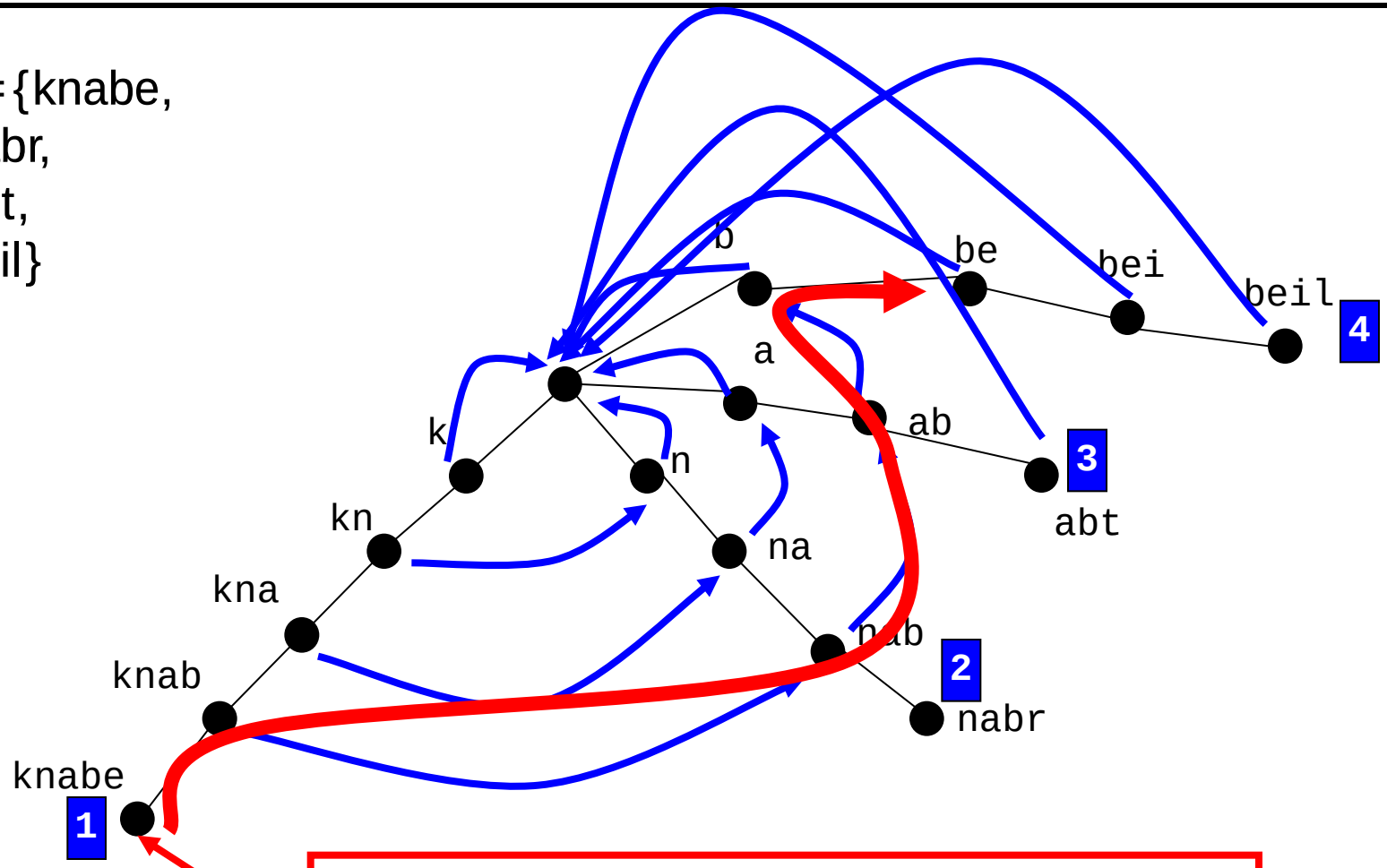
Example

P={knabe,
nabr,
abt,
beil}



Beispiel

P={knabe,
nabr,
abt,
beil}



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

Algorithmus

```
// We search failure link for k, depth(k)>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(k,k’)=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?

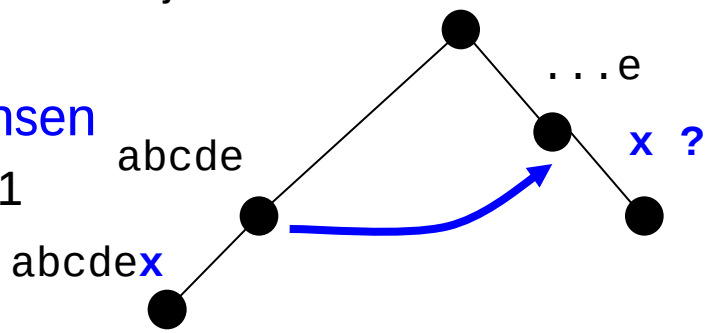
Beweisidee

```
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 es insgesamt aber nur O(n) Sprünge gibt
  - Betrachte den Pfad eines Pattern P_i
  - Wir zeigen, dass für alle Knoten auf diesem Pfad (also für dieses Pattern) insgesamt nur $O(|P_i|)$ Sprünge erfolgen
  - Damit wären es insgesamt nur O(n) Sprünge
    - Beachte: Wir überschätzen die tatsächliche Zahl, denn gleiche Präfixe sind in K nur einmal repräsentiert

Beweis

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



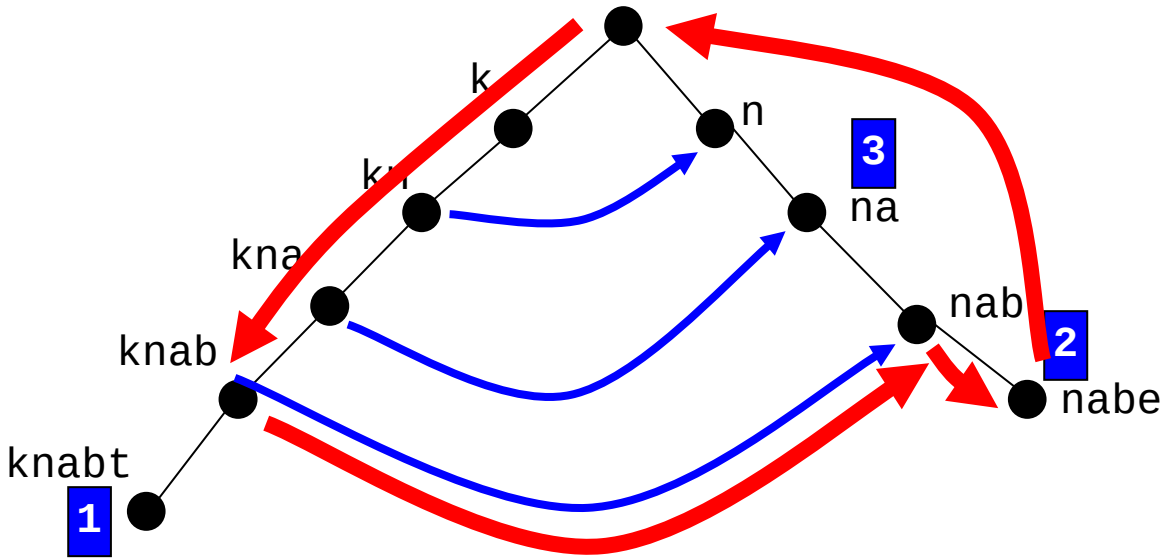
    - $length(v_j)$ wird also höchstens t mal um 1 größer über alle Knoten v_j
  - $length(v_j)$ schrumpft aber bei jedem Sprung in der WHILE Schleife um mindestens 1
  - Zusammen
    - Mit jedem Sprung >1 weniger, nie kleiner 0, insgesamt nur t-mal +1
    - Also kann es maximal t Sprünge geben

Inhalt dieser Vorlesung

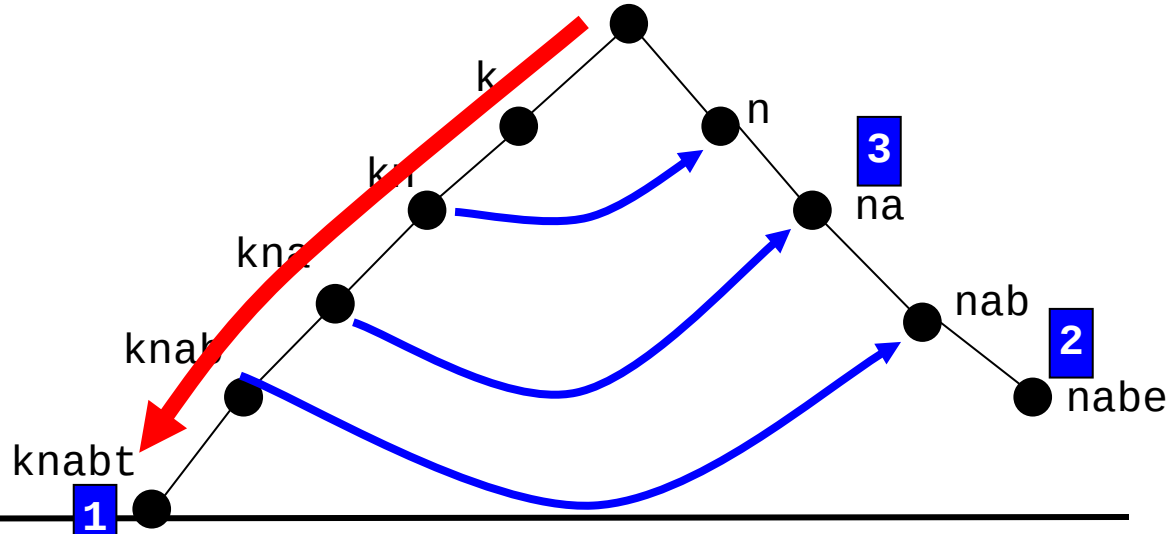
- Suche nach mehreren Mustern
- Keyword-Trees
  - Definition
  - Failure Links zur Suche
  - Konstruktion von Failure-Links in linearer Zeit
  - Output-Links
  - Ein cleverer Trick
- Suche mit Wildcards

Wann kann es schiefgehen?

T = knabenschaft



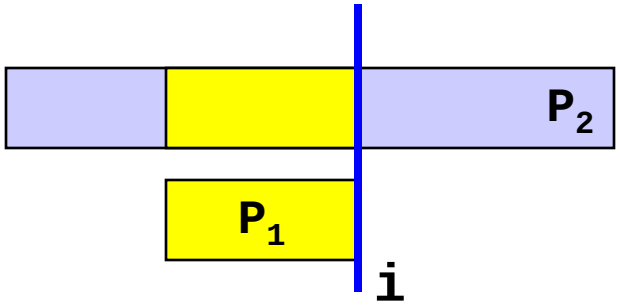
T = knabtern

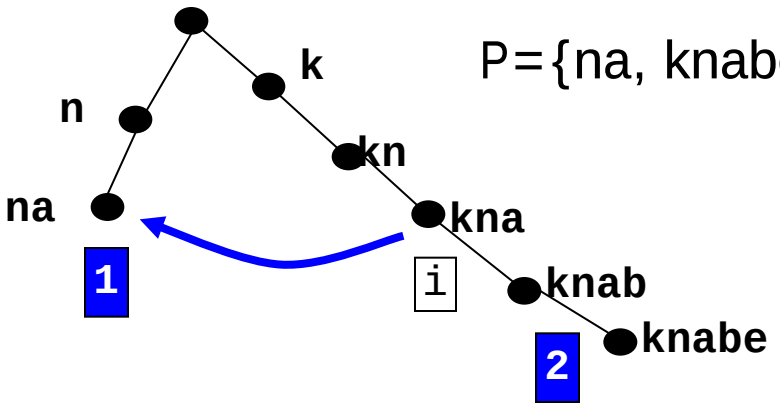


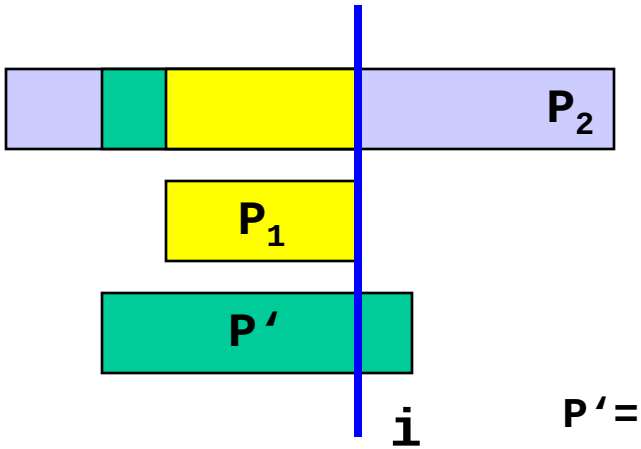
Spezialfall

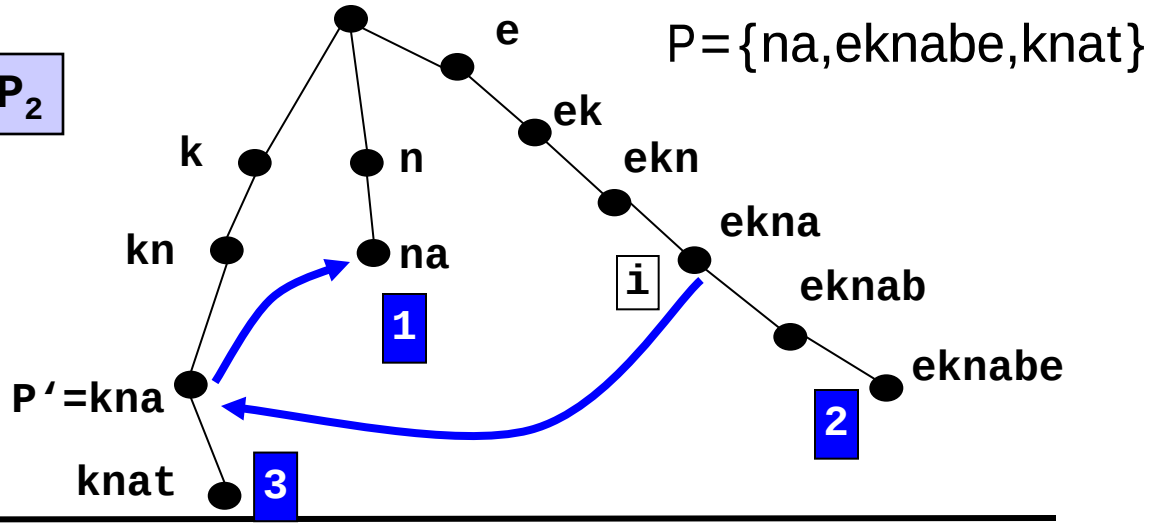
- Wir können solche Pattern übersehen, die Infix aber nicht Präfix eines anderen Pattern sind
- Sei P_1 Infix aber nicht Präfix von P_2
- Dann ist P_1 echtes Suffix von $P_2[1..i]$ für ein $i>|P_1|$
- Annahme: P_1 sei das längste echte Suffix von $P_2[1..i]$
  - Dann gilt $fl(P_2[i])=P_1$
  - Damit ist noch nichts gewonnen: Bei einer Suche folgen wir den meisten failure links ja nicht
- Sonst gibt es P‘, das längstes echtes Suffix von $P_2[1..i]$ ist
  - 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


P2
P1
i


P={na, knabe}
k
n
kn
na
kna
1
i
knab
knabe
2


P2
P1
P'
i


P={na,eknabe,knat}
e
ek
ekn
ekna
eknab
eknabe
k
n
kn
na
1
i
P'=kna
knat
3
2

Folgerung

- Wenn wir 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 Suchweg machen

Output Links

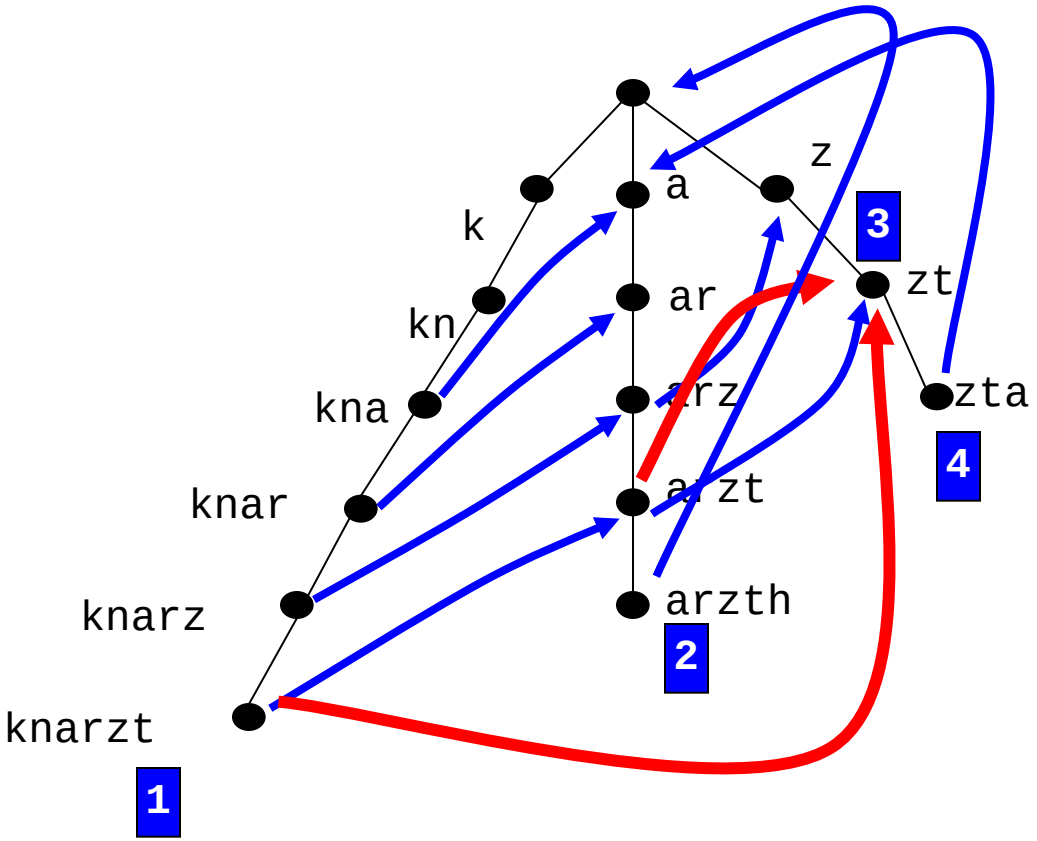
- Definition
  Der Output Link k'=out(k) von Knoten k zeigt zum ersten markierten Knoten in der Kette der Failure Links von k.
- Bemerkungen
  - Nicht alle Knoten haben Output Links
  - Output Links deuten immer auf kürzere Pattern
  - Wir können die Output Links beim Breadth-First-Traversieren des Baumes 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(k,k')=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;
```

Breadth-First Konstruktion von Output Links

P={knarzt,
arzth,
zt,
zta}



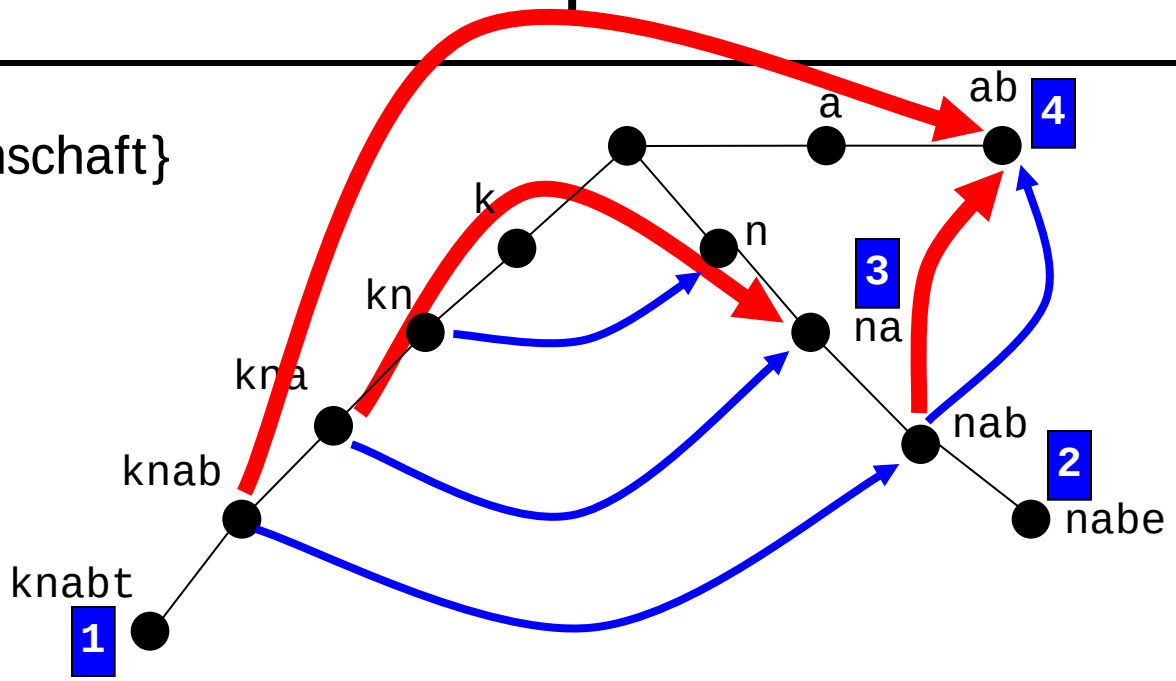
Suchphase mit Output Links

- 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

Beispiel

T={knabenschaft}

P={knabt,
nabe,
na,
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 Suchalgorithmus

```
j := 1;                    // Next comparison in T
k := root(K);              // Root node of keyword tree
while (j<|T|)
       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      // Mismatch: move on in T
              j := j+1;
       else
              k := fl(k);    // Follow the failure link
       end if;
end;
```

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)
  - Berechnung Failure Links O(n)
    - Dabei auch Berechnung der Output Links
  - Suche mit Failure/Output Links O(m+k)
- Zusammen O(n+m+k)

Inhalt dieser Vorlesung

- Suche nach mehreren Mustern
- Keyword-Trees
  - Definition
  - Failure Links zur Suche
  - Konstruktion von Failure-Links in linearer Zeit
  - Output-Links
  - Ein cleverer Trick
    - Idee: Martin Stigge und Alexandra Rostin, WS07/08
- Suche mit Wildcards

Tafel

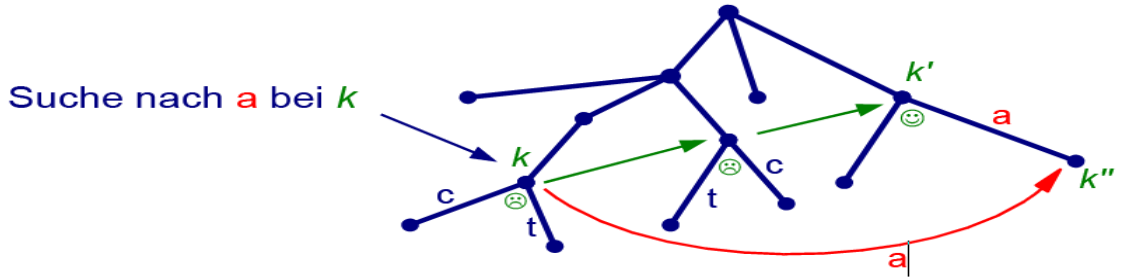
- Pattern: {EDNAR, DNAG, NAD, AX}
- Template: EDNAXEDNAXEDNAX...

Aho-Corasick Revisited

gruppe2_aufgabe2.pdf - Adobe Acrobat Professional

Erweiterung des Aho-Corasick Algorithmus

- Mismatch von x an Knoten k:
  - Failure Links ablaufen,
  - bis endlich ein Knoten k' das x an einer Kante (k', k") hat
- Das ist ja immer der selbe, also vorher klar!



- Also: Neue Kante von k zu k"

Algorithmische Bioinformatik, Aufgabenblatt 2 – Gruppe 2 10

Aho-Corasick Revisited 2

gruppe2_aufgabe2.pdf - Adobe Acrobat Professional

Erweiterung des Aho-Corasick Algorithmus

- Folgen:

Ergebnis: Beschleunigung um Faktor 4

  "nach oben" gehen,
  d.h. kein Baum mehr
  - Root zeigt mit den fehlenden Kanten auf sich selbst (das repräsentiert gerade die "finalen Mismatches")
  - ⇒ Bei der Suche: matches = |T|, mismatches = 0 (⇒FL obsolet!)
- Konstruktion der fehlenden Kanten:
  - Per Breitensuche als neue Phase nach FL-Konstruktion
  - Immer nur 1 FL pro Knoten folgen ⇒ O(n)

Algorithmische Bioinformatik, Aufgabenblatt 2 – Gruppe 2 11

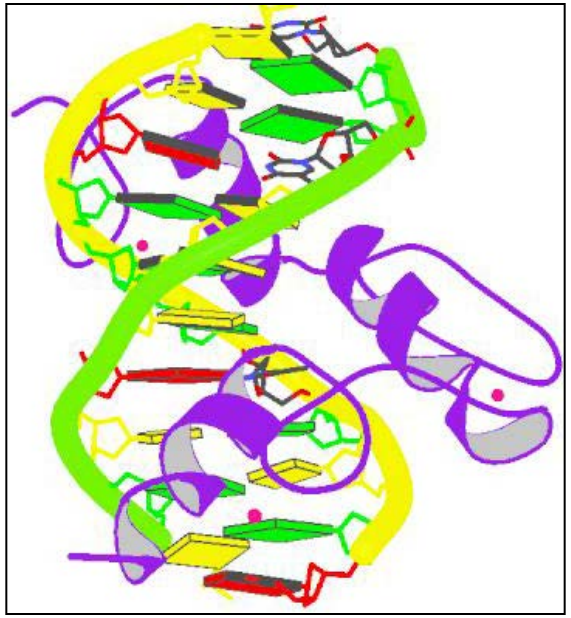
Inhalt dieser Vorlesung

- Suche nach mehreren Mustern
- Keyword-Trees
  - Definition
  - Failure Links zur Suche
  - Konstruktion von Failure-Links in linearer Zeit
  - Output-Links
  - Ein cleverer Trick
- Suche mit Wildcards

Suche mit Wildcards

- Nur ein Pattern P, aber P darf Wildcards enthalten
  - „*“ steht für exakt ein beliebiges Zeichen

- Beispiel
  - Zinc Finger Domain
  - C**C**********H**H
  - Typisches Motiv für DNA/RNA-bindende Proteine
  - Interpro IPR007087, PDB 1A1F



Clevere Verwendung von Aho-Corasick

- Initialisiere Integer-Array C=[0,0,0,…,0], mit |C|= |T|
- Sei P'={P_1,…,P_s} die Multimenge aller maximalen Substrings in P ohne Wildcards und l_1,…,l_s ihre Startpositionen
- Berechne Keyword Tree für P' und suche mit AC in T
  - 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, 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*A}
   12345678
T={TABTABDADAZA}
   123456789012
```

P_1=AB, l_1=1
P_2=DA, l_2=5
P_3=A, l_3=8

$$z=j-l_i+1$$

```
C=[000000000000]
```

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

^ Treffer

Komplexität

- O(n+m+k)
  - Gesamtlänge aller Patternfragmente ist maximal n=|P|
    - Eigentlich n-t, wenn t die Anzahl der Wildcards im Pattern ist
  - Array wird in konstanter Zeit während der AC Suche aktualisiert
  - Immer wenn dabei C[z]=s, gib z aus

Grenzen des linearen Stringmatching

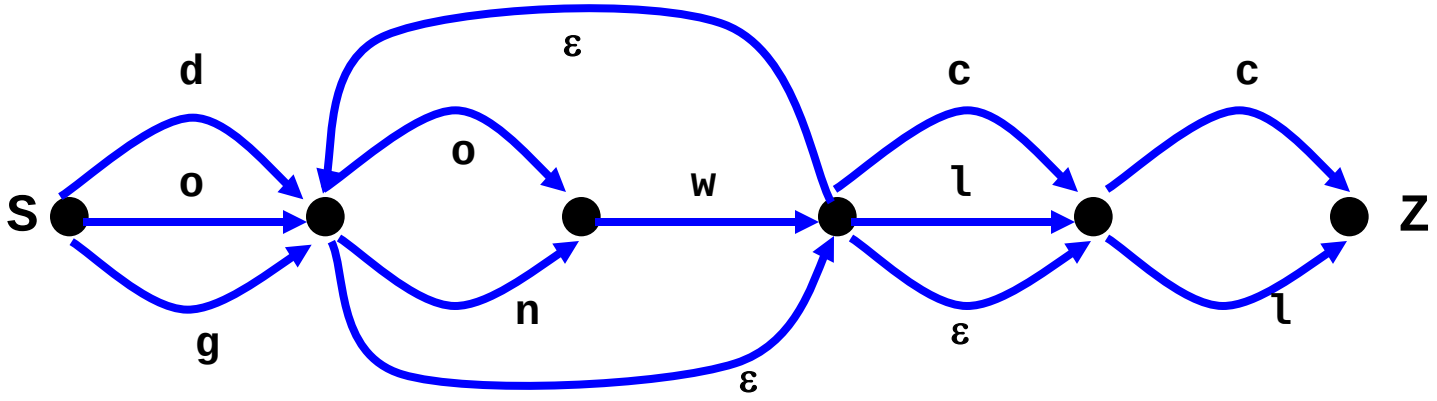
- Wir können in linearer Zeit finden ...
  - alle Vorkommen eines Pattern P in einem Template T
  - alle Vorkommen einer Menge von Pattern in einem Template T
  - alle Vorkommen eines Pattern P mit Wildcard in einem Template T
  - alle Vorkommen eines Pattern P mit maximal k Mismatches in T
    - Zeigen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

Suche mit regulären Ausdrücken

- Reguläre Ausdrücke
  - „ε“ Leeres Zeichen
  - „|“ „oder“
  - „()“ Gruppierung
  - „*“ Kleene‘sche Hülle
  - (Es fehlen „+“, „.“, „[]“, zählen, …)
  - Rekursive Definition wie üblich
- Beispiel aus der Bioinformatik
  - PROSITE: Datenbank für Motive in Proteinsequenzen
  - Proteindomänen sind funktionale Einheiten in Proteinsequenzen
  - Beschrieben durch reguläre Ausdrücke
    - Aber tw. andere Operatoren als die oben genannten

Problem

- Gegeben: Regulärer Ausdruck R, Template T
- Gesucht: Das erste Vorkommen von R in T
- Äquivalenz: nichtdeterministischer endlicher Automaten (NEA)
  - Konstruktion: Siehe Literatur
  - 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 und G(R) der dazugehörige NEA mit Startzustand S und Endzustand Z*
  - *Ein Substring T‘ von T matched R, wenn T‘ durch einen Pfad in G(R) von S nach Z ausgesprochen wird*
- Bemerkungen
  - 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 das erste T‘ in T zu finden, das R matcht
  - Erweiterung auf alle T‘: Literatur

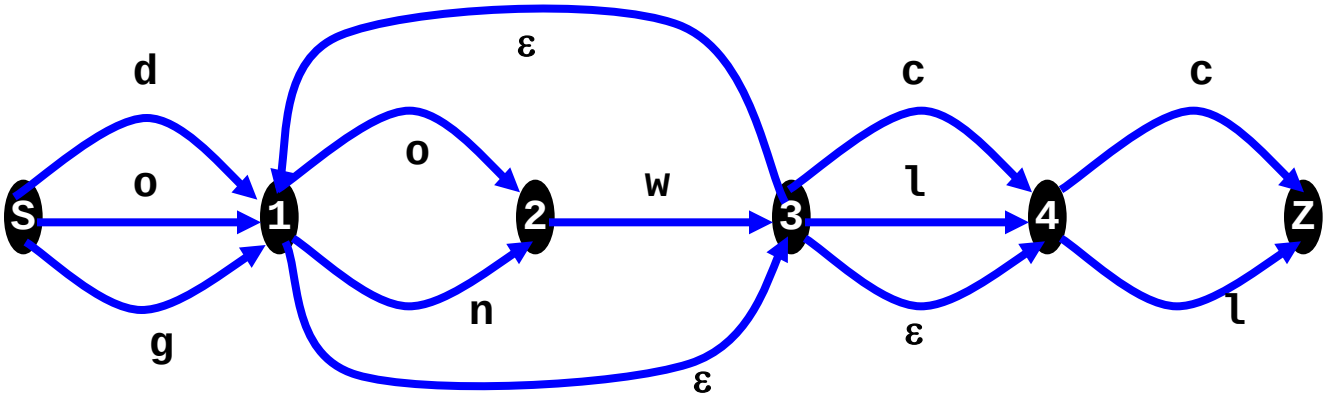
Algorithmusidee

- Wir beginnen mit dem leichteren Problem: Matched ein Präfix von T mit R?
- Idee
  - Betrachte G(R) und T von links nach rechts
  - Zähle in aufsteigender Länge alle Pfade in G(R) auf, 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 beliebig vielen ε-Kanten erreicht werden können. Außerdem sei $S \in N(0)$
  - Induktionsschritt
    - 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 beliebig viele (oder keine) 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)
  - Sei e die Gesamtanzahl von ε-Kanten in G(R)
  - Match für T[i] finden geht in O(1) (oder O(|Σ|))
  - Danach können höchstens e ε-Kanten folgen
    - Wichtig ist nur, welche Zustände wir erreichen können – wie, ist egal
    - Schranke e gilt für alle Zustände aus N(i-1), die wir weiter verfolgen
    - Zyklen können abgebrochen werden (braucht etwas Speicher)
  - Also ist dieser Schritt O(e)
- Wir berechnen |T|=m Mengen N(1) ... N(m): O(m*e)
- Beobachtung
  - Ein reg. Ausdruck mit n=|R| Symbolen hat maximal O(n) ε-Kanten
  - Sonst kann er minimiert werden
- Zusammen: O(m*n)

Bemerkung

- Jeden NEA kann man in einen DEA konvertieren
- Ein DEA erlaubt Matchen in linearer Zeit
  - Wenn wir die Failure Links dazu berechnen
- Aber
  - Linear in der Zahl der Zustände
  - Der durch Konvertierung erzeugte DEA hat aber im Worst-Case exponentiell viele Zustände
- In der Praxis hat er die aber nicht – RegExp matching ist meistens „quasi-linear“

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
  - Hier passiert anscheinend ein Komplexitätssprung: Länge des Matches ist nicht mehr durch Länge des Patterns vorgegeben

Selbsttest

- Was ist ein Failure Link und wozu ist er da?
- Wo sind Failure Links im KMP Algorithmus „versteckt“?
- Konstruieren Sie zum Pattern P=„...“ ein zweites Pattern P‘ und ein Template so, dass ein Vorkommen von P‘ bei der Suche ohne Output-Links übersehen werden würde
- Der AC-Algorithmus ist O(m+n+k). Was ist eine obere Schranke für k?