

Bioinformatik

Suche nach mehreren Mustern



Ulf Leser
Wissensmanagement in der
Bioinformatik



Grundidee Knuth-Morris-Pratt

- Erinnerung an den naiven Algorithmus

ctgagatcgcgta

gagatc
gagatc
gagatc
gagatc
gatatc
gatatc
gatatc

abcxabcgedsc

abcxabcde

Wie weit kann man jetzt sicher springen ?

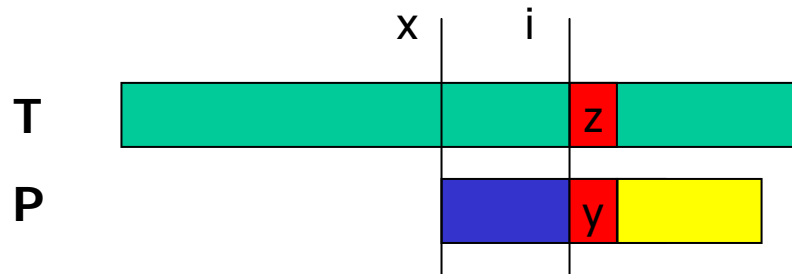
- T beginnt mit abcxabc
- Das zweite „a“ in P kommt an Position 5

Also: 4 Zeichen (mind.) schieben

- Idee: Ausnutzen von
 - Wissen über die gematchten Zeichen benutzen
 - Dazu: Vorprozessierung von P

Preprocessing

- Was wollen wir wissen?
 - Bei einem Mismatch an Position i (in P) gilt:
 $T[x .. x+i-1] = P[1 .. i-1]$



- Wir suchen nach einem möglichst großen Shift
- Wir vergleichen das (schon gematchte) Präfix von P mit dem Rest
- Kommt der Anfang von P noch mal in $P[1 .. i-1]$ vor?
- Wenn ja – schiebe bis dahin
 - mehrmaliges Vorkommen später
- Wenn nein – schiebe um $i-1$
 - den Teil von T kennen wir noch nicht

Definitionen

- Definition

- Sei sp_i die *Länge des längsten echten Suffix von $P[1..i]$, das mit einem Präfix von P matched*
- Sei sp_i' die *Länge des längsten echten Suffix von $P[1..i]$, das mit einem Präfix von P matched und für das gilt: $P(i+1) \neq P(sp_i+1)$; sonst 0*

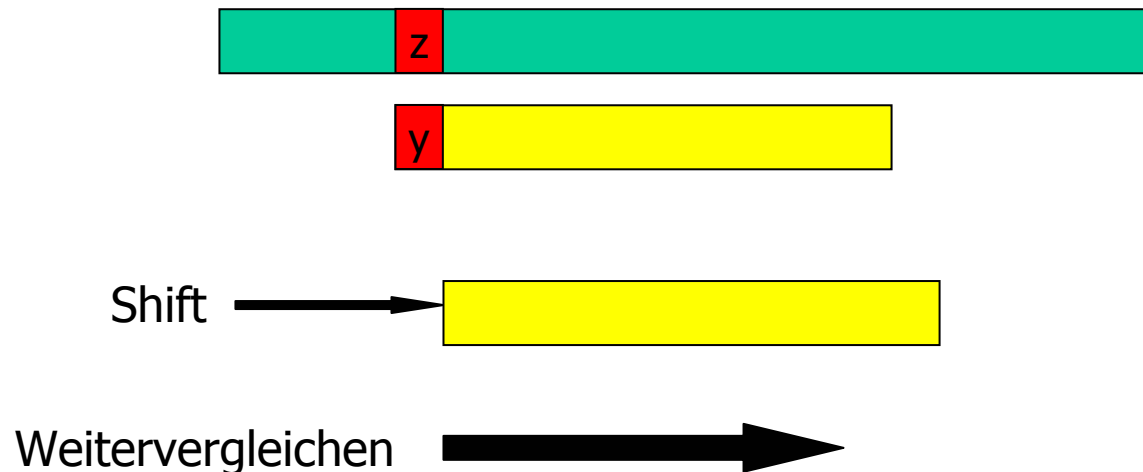
- Beispiel

P: **abcaeabcabd**
sp_i: 00010123420
 abca
 abcaeab
 abcaeabc
 abcaeabca

P: **abcaeabcabd**
sp_i: 00010123420
sp_i' : 00010**000**420
 abcaeabc
 abcaeabcab

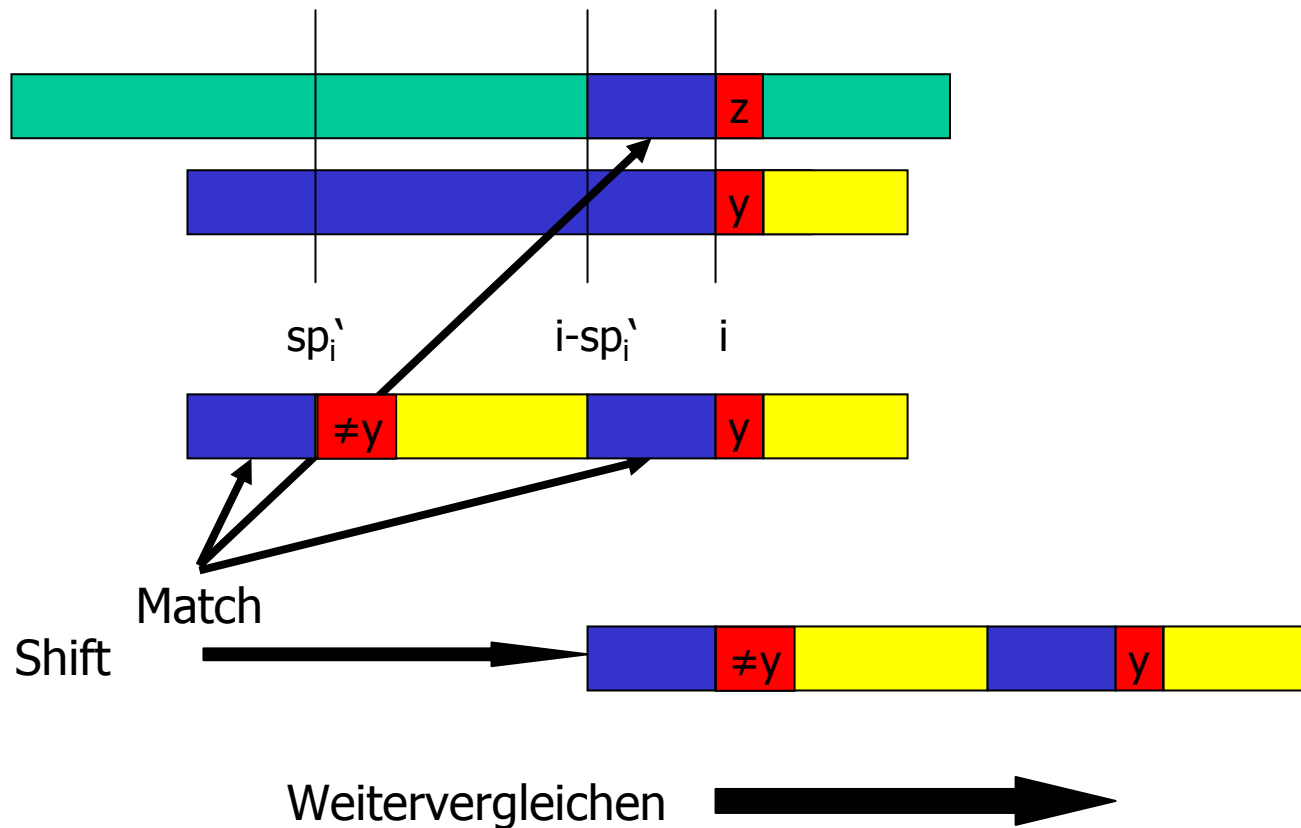
Shift Regel 1

- Wenn $P(1)$ und $T(k)$ **sofort mismatchen**
 - Schiebe P um 1 Positionen nach rechts
 - Vergleiche weiter ab $P(1)$



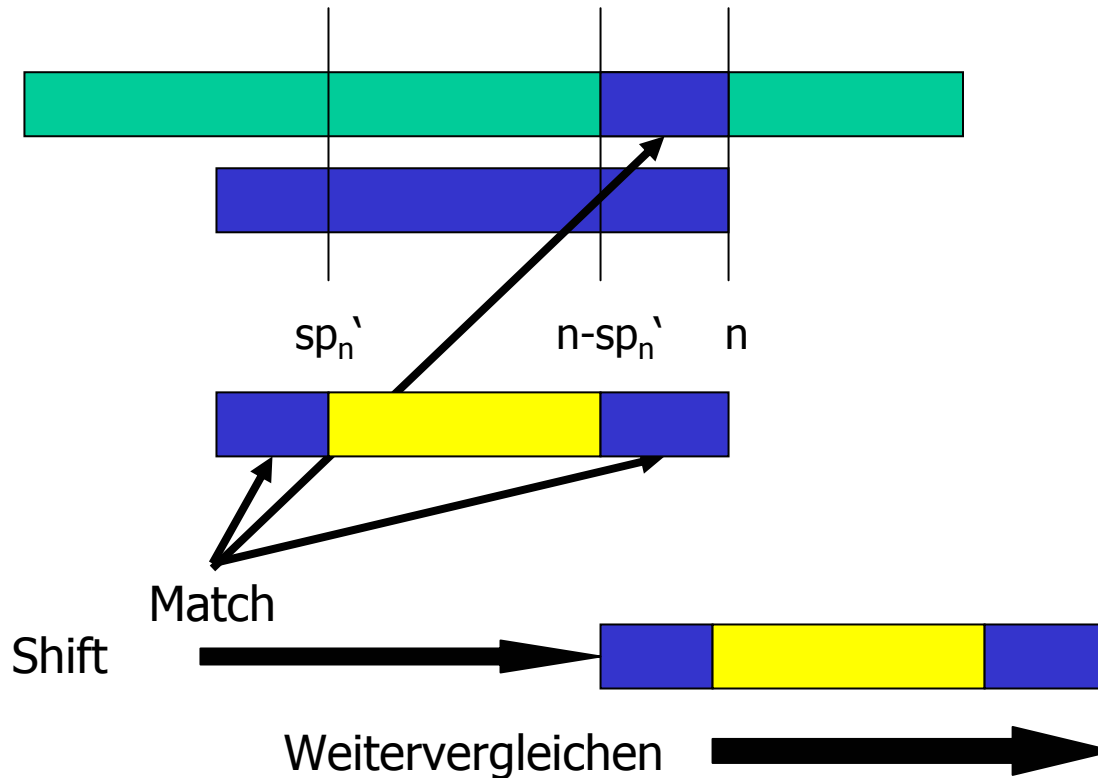
Shift Regel 2

- Wenn bei $i+1$ in P der **erste Mismatch** vorkommt
 - Schiebe P um $i - sp_i'$ Positionen nach rechts
 - Vergleiche weiter ab $P(sp_i' + 1)$



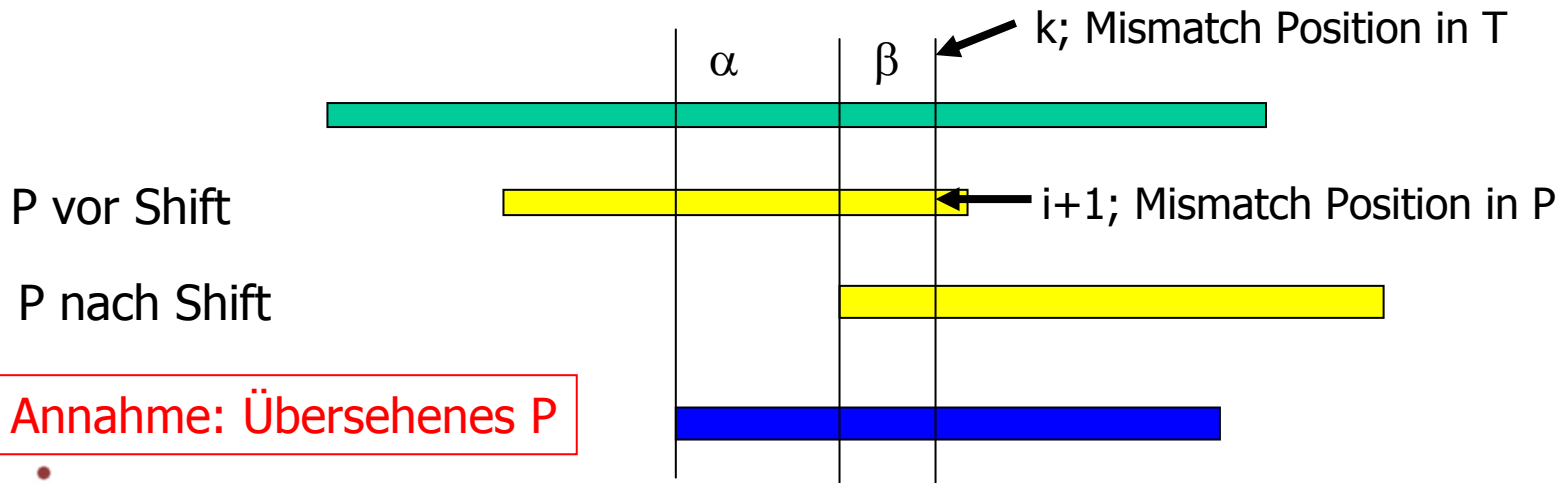
Shift Regel 3

- Wenn ein **kompletter Match** gefunden wird
 - Schiebe P um $n - sp_n'$ Positionen nach rechts
 - Vergleiche weiter ab $P(sp_n' + 1)$



Korrektheit der Shift-Regel

- Schiebt man nicht eventuell zu weit?
- Theorem
 - *Die Shift-Regel verschiebt nie soweit, dass ein Vorkommen von P in T übersehen wird*
- Beweis
 - Wenn das anders wäre, hätten wir:



Preprocessing

- Wir führen das Preprocessing auf Z-Boxen zurück
- Erinnerung (Z-Box)
 - Sei $i > 1$. Dann ist Z_i die Länge des *größten Substrings* x in P mit
 - $x = P[i..i+|x|-1]$ (x startet an Position i in P)
 - $P[i..i+|x|-1] = P[1..|x|-1]$ (x ist auch Präfix von P)
 - x heißt *Z-Box* von P an Position i mit Länge $Z_i(P)$



Kompletter KMP Algorithmus

```
compute spi`;  
c := 1;           // Next comparison in T  
p := 1;           // Next comparison in P  
while c+(n-p)<=m do  
    while P(p)=T(c) and p<=n do  
        p++;  
        c++;  
    end while;  
    if p=n+1 then  
        print c-n;    // Match  
    else if p=1 then  
        c++;           // First comp. fails - move 1 pos  
    end if;  
    // Comparison in T will continue at position c  
    // Comparison in P will continue after shift  
    p:=spp-1` +1;    // p is mismatch pos. in P  
end while;
```

Vergleich

	Z-Box	Boyer-Moore (Apostolico-Giancarlo)	Knuth-Morris-Pratt
Komp. Preproc. Komp. Suche Komp. Gesamt	$O(m+n)$ $O(m)$ $O(m+n)$	$O(n)$ $O(m)$ $O(m+n)$	$O(n)$ $O(m)$ $O(m+n)$
Größe Alphabet	<ul style="list-style-type: none"> • Praktisch unabhängig von Alphabetgröße 	<ul style="list-style-type: none"> • Je größer, desto besser – BCR führt zu großen Sprüngen • BCR greift nicht bei kleinen Alphabeten 	<ul style="list-style-type: none"> • Praktisch unabhängig von Alphabetgröße
Bemerkung	<ul style="list-style-type: none"> • Avg und Worst Case Komplexität gleich - es wird jedes Zeichen mind. einmal verglichen 	<ul style="list-style-type: none"> • WC der einfachen Variante bei Wiederholungen (z.B: a^n in a^m) • Best Case ist $O(m/n)$ [Suche a^n in b^m] 	<ul style="list-style-type: none"> • Erweiterbar auf mehrere Pattern

Inhalt dieser Vorlesung

- Suche nach mehreren Mustern
 - Motivation
 - Keyword-Trees
 - Failure Links
 - Suche mit Failure Links

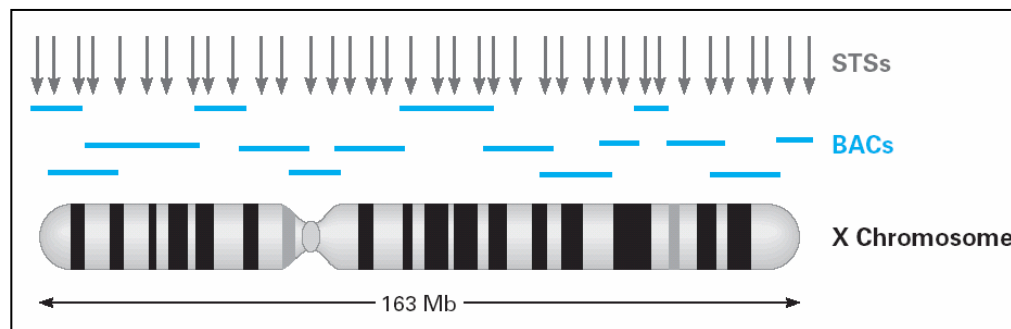
Suche nach mehreren Mustern

- 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)

Physikalische Karten - Übersicht

- Zwei Arten von Objekten
 - Clone: YAC, PAC, BAC, Cosmids, ... mit **Ausdehnung**
 - Marker: Loci, STS, EST, ... „ohne“ **Ausdehnung**
- Kartierungsverfahren
 - STS Content Mapping



Restriktionsenzyme

- Proteine, die DNA an **spezifischen Stellen** schneiden
 - Notwendig in Zelle als „Müllabfuhr“
 - Viele bekannt (kommerziell)
 - Partieller Verdau möglich
- Beispiel: Enzym EcoRI schneidet GAATTC

...GAATTC...

...G | AATTC...

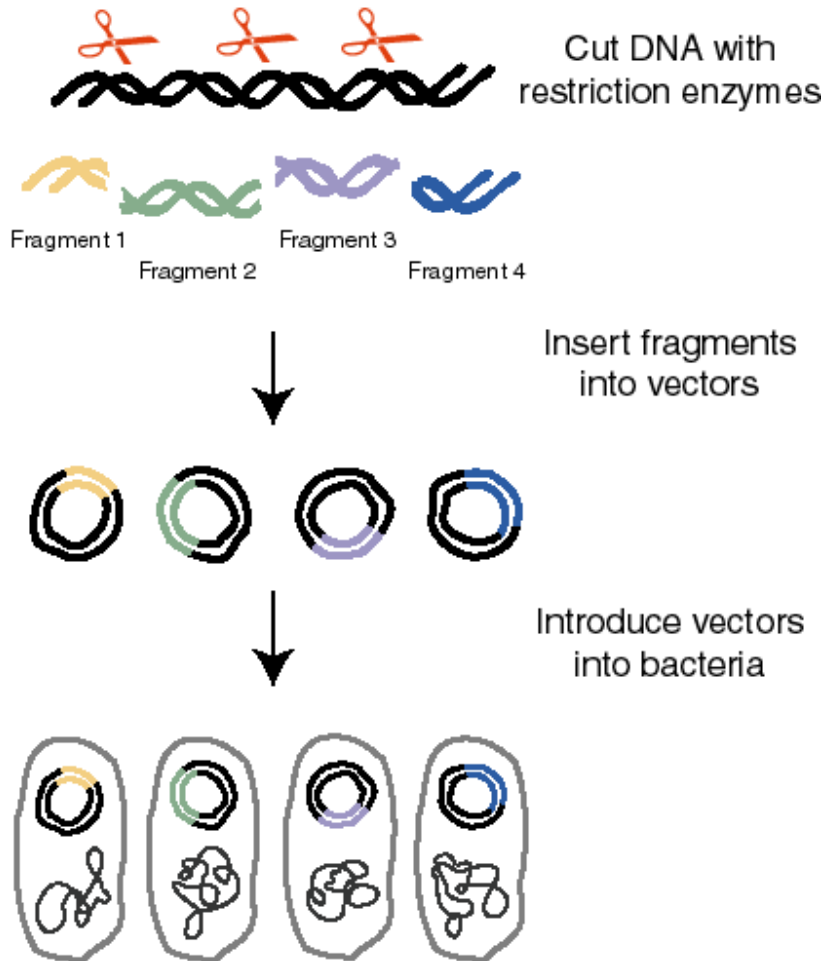
...CCAGAATTCTCG...

...CCAG | AATTCTCG...

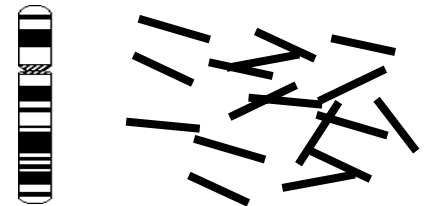
...GGTCTTAAGAGC...

...GGTCTTAA | GAGC...

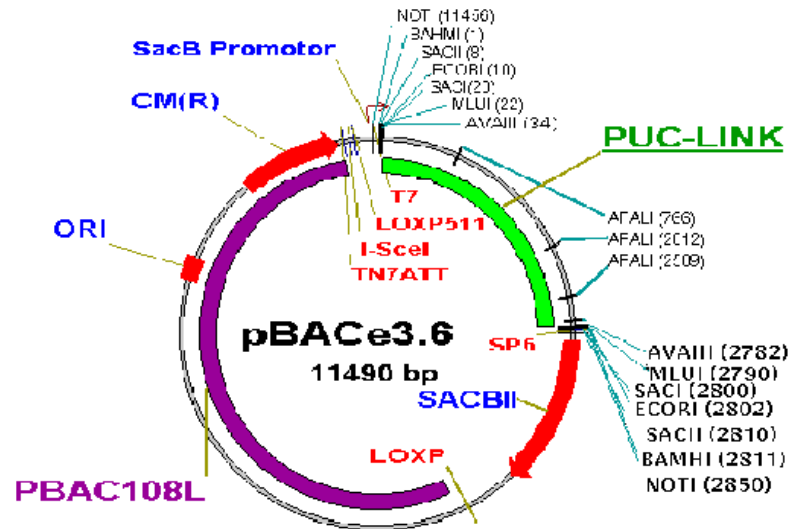
Clonieren: Erstellen der „Bruchstücke“



- Schneiden mit Restriktionsenzymen
 - Bruchstücke unterschiedlicher Länge
- Auftrennen nach Länge
 - Gelelektrophorese
- Clonierung in Bakterien
 - Vervielfältigung
- Ergebnis:



Clonierung



- Isoliertes Bruchstück wird in einem „Vektor“ eingebracht
 - DNA replication initiation site
 - Antibiotika-Resistenzen
 - Schnittstellen zur späteren Extraktion
 - ...
- Dieser wird von Bakterie als Chromosom betrachtet

Clonebibliotheken

- Mapping erfordert viele Versuche
- Identität der Clone muss gleich bleiben
- Bibliotheken
 - Mengen von Bakterien mit Clonen
 - Lassen sich einfach vervielfältigen
 - Internationale Verwendung
 - Identifikation: Koordinaten
 - ICRFy900F18, CEPH44G, 128_H_44
- Zur Weiterverarbeitung: Übertragung auf „Filter“
 - Extraktion der „Nutzlast“ aus Bakterien
 - Fixierung auf Material (Nylon, Glas, Silikon)



Clonebibliotheken kann man kaufen

The Sanger Institute : Information | Resource Center/Primary Database | Loading...

Library Information Table

Please mail dmb@sanger.ac.uk with any comments about this page.

Library Type	Code	Library	Library Code	Antibiotic	Vector	Cloning Site	Genomic Digest	Excise insert with	Amplify insert with
Cosmid	c	LLOXNC01 "U" Lawrence Livermore flow-sorted X-chromosome	cU	Kanamycin 30ug/ml	Lawrist 16 (sequence)	BamHI	Mbol Partial	Sfil	-
Cosmid	c	LL22NC03 "N" Lawrence Livermore flow-sorted X-chromosome	cN	Kanamycin 30ug/ml	Lawrist 16 (sequence)	BamHI	Mbol Partial	Sfil	-
Cosmid	c	SC6cA Sanger Institute flow-sorted chromosome 6	cA	Kanamycin 30ug/ml	Lawrist 16 (sequence)	BamHI	Mbol Partial	Sfil	-
Cosmid	c	LA04NC01 Los Alamos flow-sorted chromosome 4	cL	Kanamycin 30ug/ml	sCos1(sequence)	BamHI	?	?	-
Cosmid	c	SC7cD Sanger Institute flow-sorted chromosome 7	cD	Kanamycin 30ug/ml	Lawrist 16 (sequence)	BamHI	Mbol Partial	Sfil	-
		SCXcC							

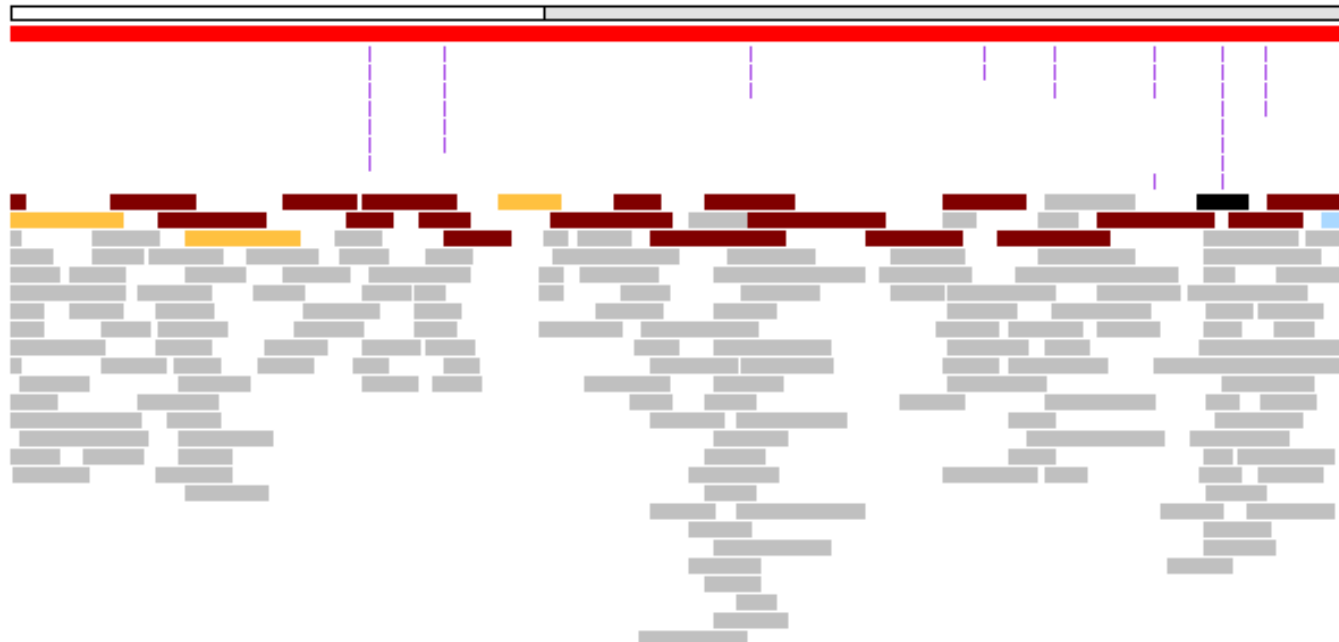
Transferring data from www.rzpd.de...

DNA Marker

- Eindeutig im Genom lokalisierbare „Punkte“
- Können polymorph sein
 - Z.B. definierter Anfang und definiertes Ende, aber variables Zwischenstück
- STS
 - „Sequence-tagged Sites“
 - Eindeutige Sequenzen
 - Länge 300-500 Basenpaare
- EST
 - „Expressed Sequence Tags“
 - Spezialfall eines STS; muss aus kodierendem Abschnitt (Gen) kommen

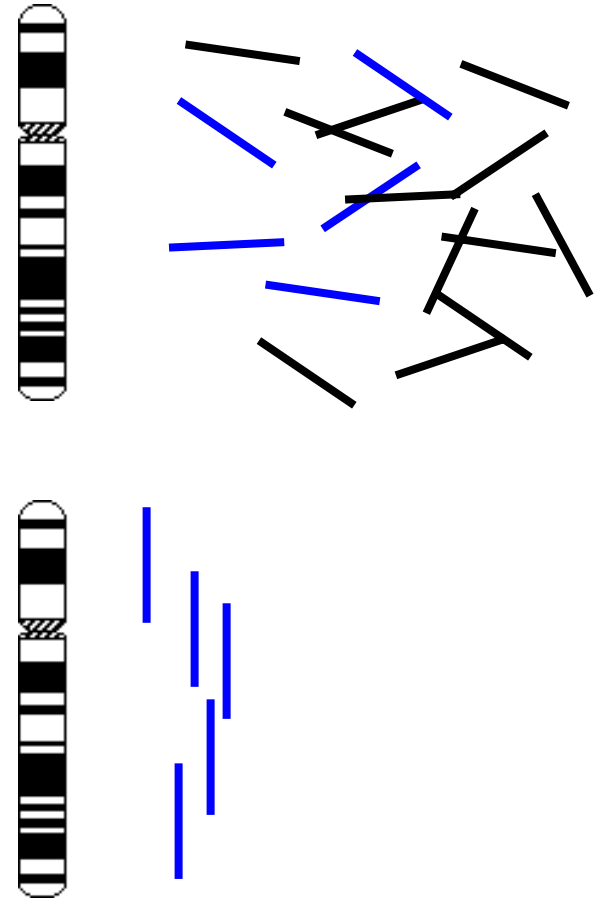
Mapping und Sequenzierung

- Zerlegung in Bruchstücke (Clonierung)
- Berechnung aller Überlappungen
- Bestimmung der wahrscheinlichsten Gesamtsequenz
- Variante: Bestimmung des **Minimum Tiling Paths**



Kartierungstechniken

- Ziel ist der Minimum Tiling Path
- Was kann man über Clone/Marker herausbekommen ?
 - Clone enthalten Marker
 - PCR



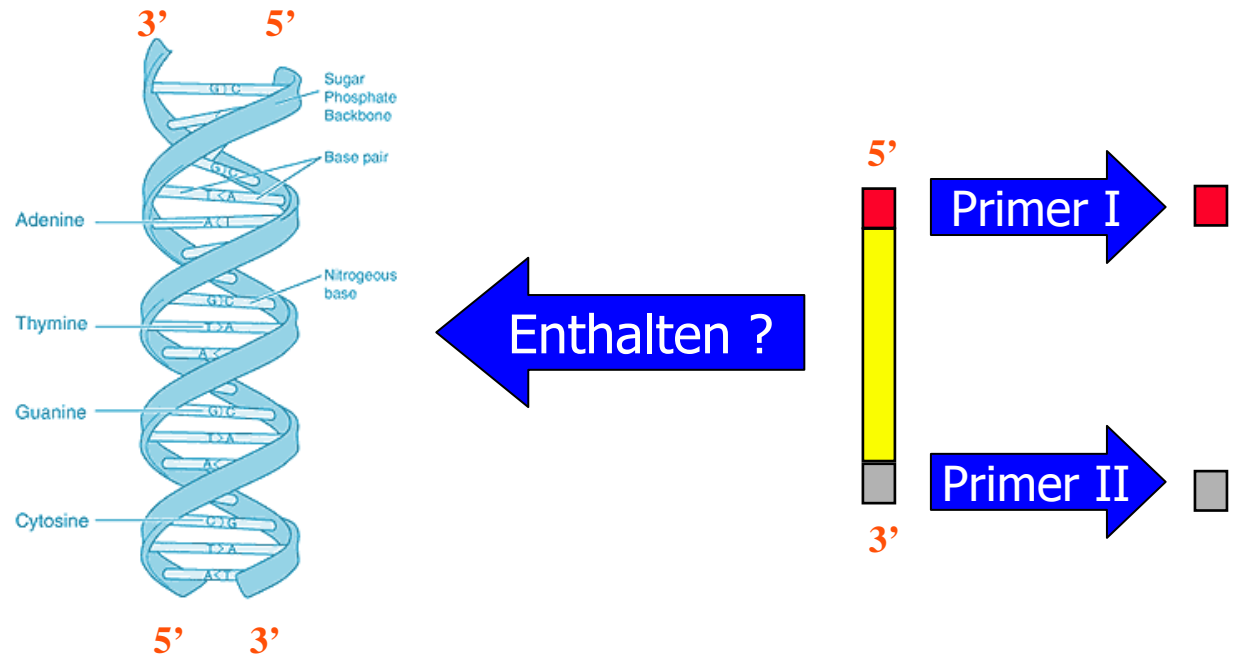
PCR (Sketch)

- **Polymerase Chain Reaction**

- Methode zur Vervielfältigung von DNA Bruchstücken
- Zusammen mit Gelelektrophorese hochempfindliches Nachweisverfahren
- Nachzuweisende Sequenz muss bekannt sein
- Vielfältige Verwendung
 - „Genetischer Fingerabdruck“
 - Forensik
 - Vaterschaftstests

PCR Illustration

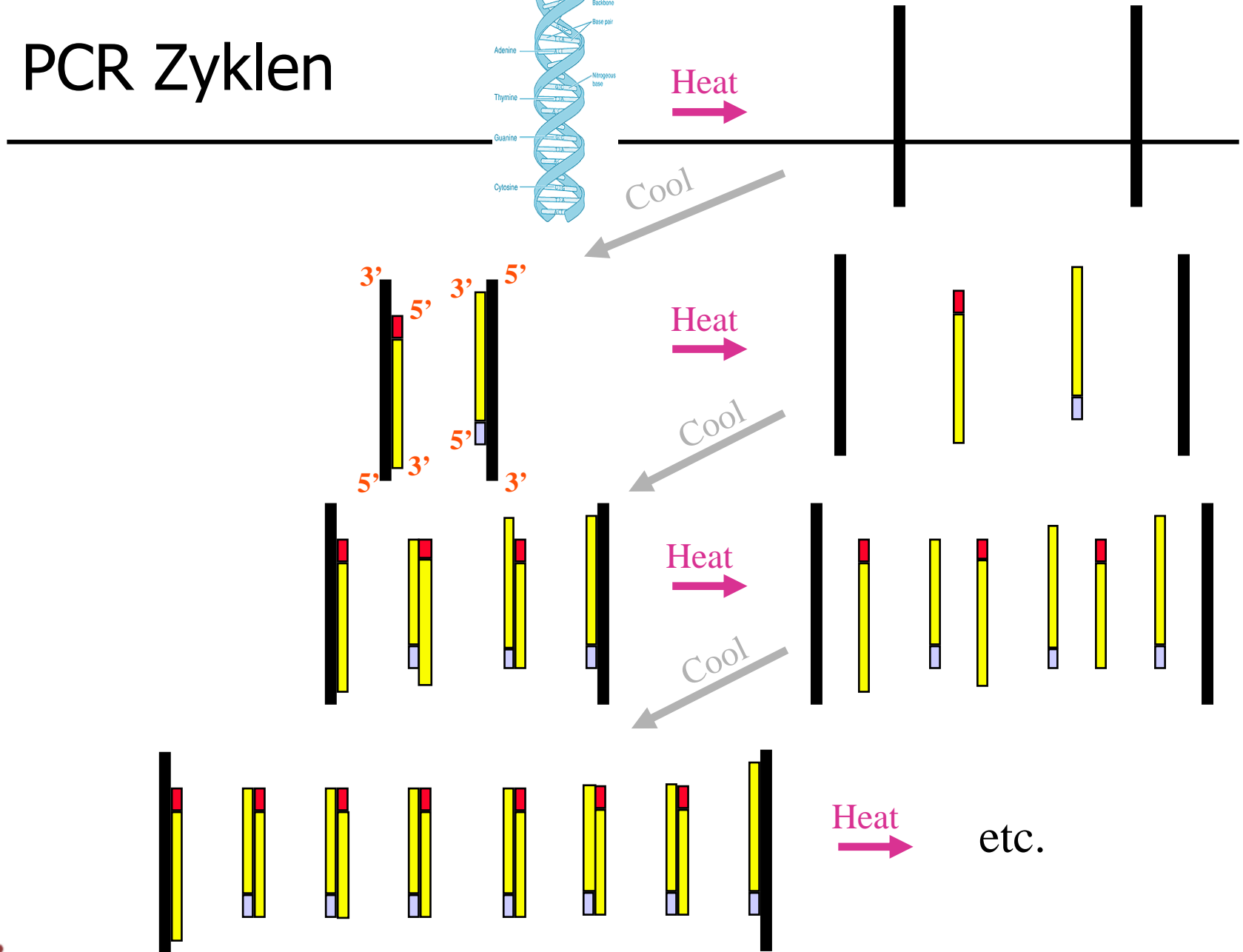
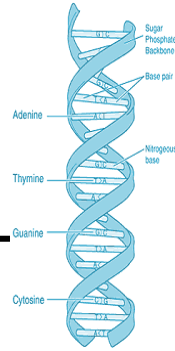
- Gegeben:
 - Doppelsträngige DNA D
 - Probe S mit bekannter Sequenz
- Frage:
 - $S \in D$?



PCR Idee

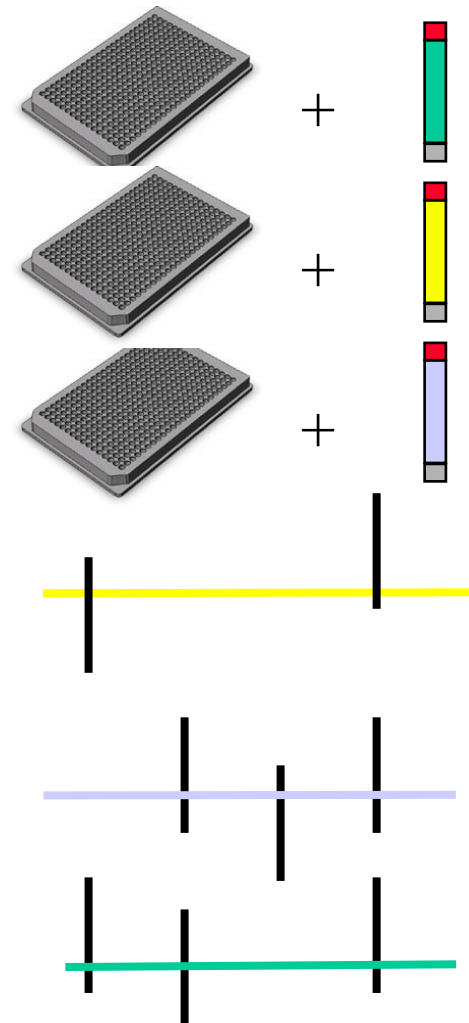
- Zutaten
 - Doppelsträngige DNA kann durch Erhitzen in einzelsträngige aufgebrochen werden
 - Polymerase ist ein Enzym, dass einzelsträngige DNA verdoppelt
 - Muss an ein Stück doppelsträngige DNA binden
 - Verlängert diese immer in eine Richtung, am 3' Ende
- PCR Lösung
 - Zu untersuchende DNA
 - Primer
 - Polymerase
 - Nukleinsäuren

PCR Zyklen

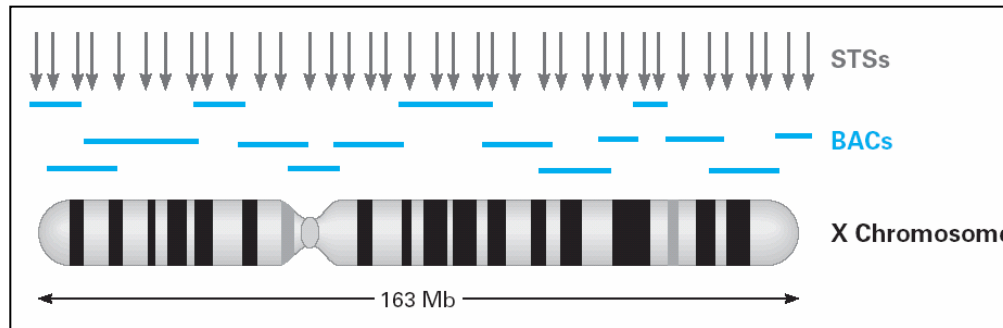


STS Content Mapping

- Gegeben
 - Menge von Clonen
 - Menge von STS
- Gesucht
 - Welcher der Clone enthält welche STS ?
- Verfahren
 - PCR mit jedem STS
- Ergebnis
 - „Anchoring“ der Clone
 - Daraus kann man die Reihenfolge der Clone berechnen



Electronic PCR: STS Kartierung

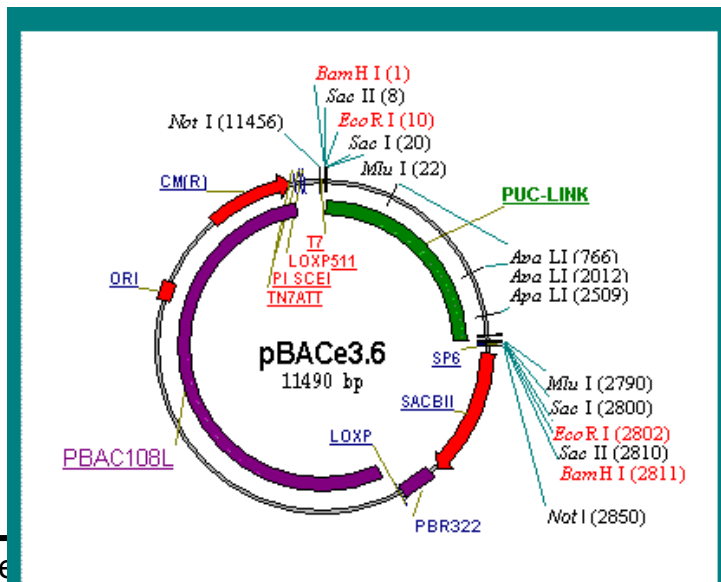


- Problem

- Gegeben eine STS Karte und ein Stück Sequenz (Clone)
- Sequenz der STS sei bekannt (Sequenzierung)
- Frage: Wo liegt der Clone auf der Karte?
 - Welche STS sind in der Sequenz enthalten?
 - Gleichzeitige Suche von **Tausenden** (Chromosom bekannt) oder **Millionen** von STS auf der Clonesequenz
 - „Z“ (P_1, \dots, P_z) ist ein wichtiger Faktor für Komplexität

Beispiel 2: Sequenzmaskierung

- Sequenzen durchlaufen viele Schritte
 - Vervielfältigung durch PCR
 - Clonierung und Vervielfältigung in Wirtszelle
 - Extraktion
- Häufig auftretende Probleme: **Verunreinigungen**
 - PCR Primer
 - Spezieller „Vektoren“ für und von Wirtszelle



Quelle: <http://www.genome.washington.edu>

Sequenzmaskierung 2

- Sequenzmaskierung
 - Falsche Teilsequenzen sind schlimm: **Sequenzassemblierung misslingt** oder ist falsch
 - Standard: Durchsuchen einer frisch erstellten Sequenz nach allen bekannten potentiellen Verunreinigungen
 - Wichtiger Schritt vor Assemblierung
- Es gibt spezielle **Bibliotheken typischer Verunreinigungen**
 - Abhängig von Wirtsorganismus und Clonierung / Sequenzierungsverfahren
- Problem
 - Gegeben
 - Menge von typischen Verunreinigungsvektoren
 - Neue Sequenz
 - Durchsuchen Sequenz nach allen Verunreinigungen („Vektoren“)

-
- Suche nach mehreren Pattern P_1, \dots, P_z in einem String T

Erster Versuch

- Sei $P = \{P_1, P_2, \dots, P_z\}$, $n = |P_1| + |P_2| + \dots + |P_z|$
 - 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 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: $O(n + z * m)$
- Geht das nicht schneller?

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
 - Finden einer **Datenstruktur zur Abbildung der Gemeinsamkeiten**
 - Suche durch T mit dieser Datenstruktur sucht **gleichzeitig** nach allen Pattern
 - Egal, wie viele es sind

Keyword Trees

- Definition

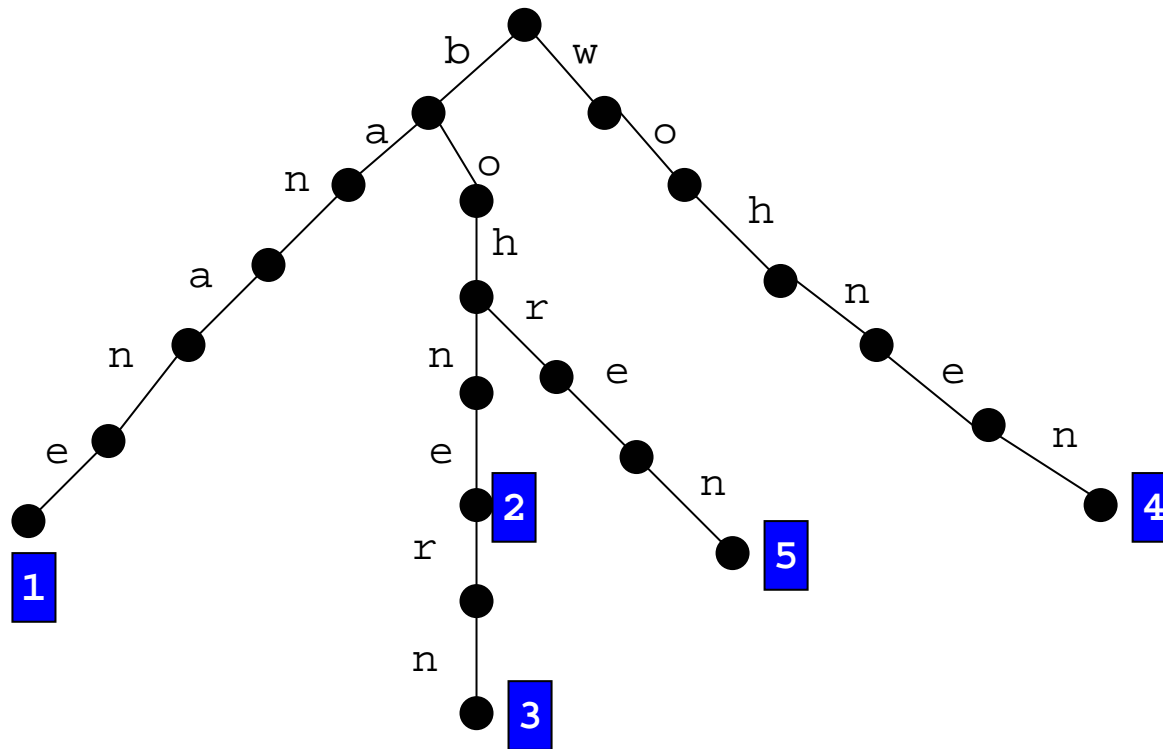
- Sei P eine Menge von Pattern. Ein *Keyword Tree* für P ist ein Baum mit den folgenden Eigenschaften
 - Jede Kante ist mit *genau einem Zeichen* beschriftet (Label)
 - Wenn ein Knoten mehrere Kinder hat, so sind die Kanten zu den Kindern mit *unterschiedlichen Zeichen* beschriftet
 - Das Label eines Knoten k , $label(k)$, ist die Konkatination der Label 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 $mark(k)=i$
 - Für nicht-markierte Knoten k ist $mark(k)=UNDEF$
 - *Jedes Blatt entspricht mindestens einem P_i*
 - Also: Jedes Blatt k ist markiert, $mark(k) \neq UNDEF$

- Folgerungen

- Jeder Pfad ist eindeutig
- Der Baum ist minimal; es gibt keine „überflüssigen“ Knoten

Beispiel

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



Konstruktion

- **Konstruktion möglich in $O(n)$**

- Beginne mit P_1 (das 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 $O(|P_2|)$

- Induktion über $P_3 - P_z$ zeigt die Behauptung

- **Beachte**

- Alle Pfade sind eindeutig – Präfix ablaufen ist also linear

Naive Verwendung

- Naive Verwendung eines Keyword Tree
 - Gegeben Menge von Pattern P und Template T
 - Baue Keyword Tree K für P in $O(n)$
 - Iteriere über Positionen i in T
 - Laufe Präfix von $T[i..]$ in K ab
 - Wenn markierter Knoten passiert wird, melde die Pattern
 - Ist für Position $l > i$ kein Pfad in K vorhanden: Starte erneut ab Position $i+1$ in T und Wurzel von K
 - Komplexität: $O(n+m*n_{\max})$, mit $n_{\max} = \max(|P_i|)$
 - Im Average-Case schon was gewonnen, aber nicht genug
- Pendant zum naiven Suchalgorithmus
- Gesucht: das **Pendant zum KMP**

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

Weiteres Vorgehen

- Definition von Failure Links
- Suchphase
- Ende für heute

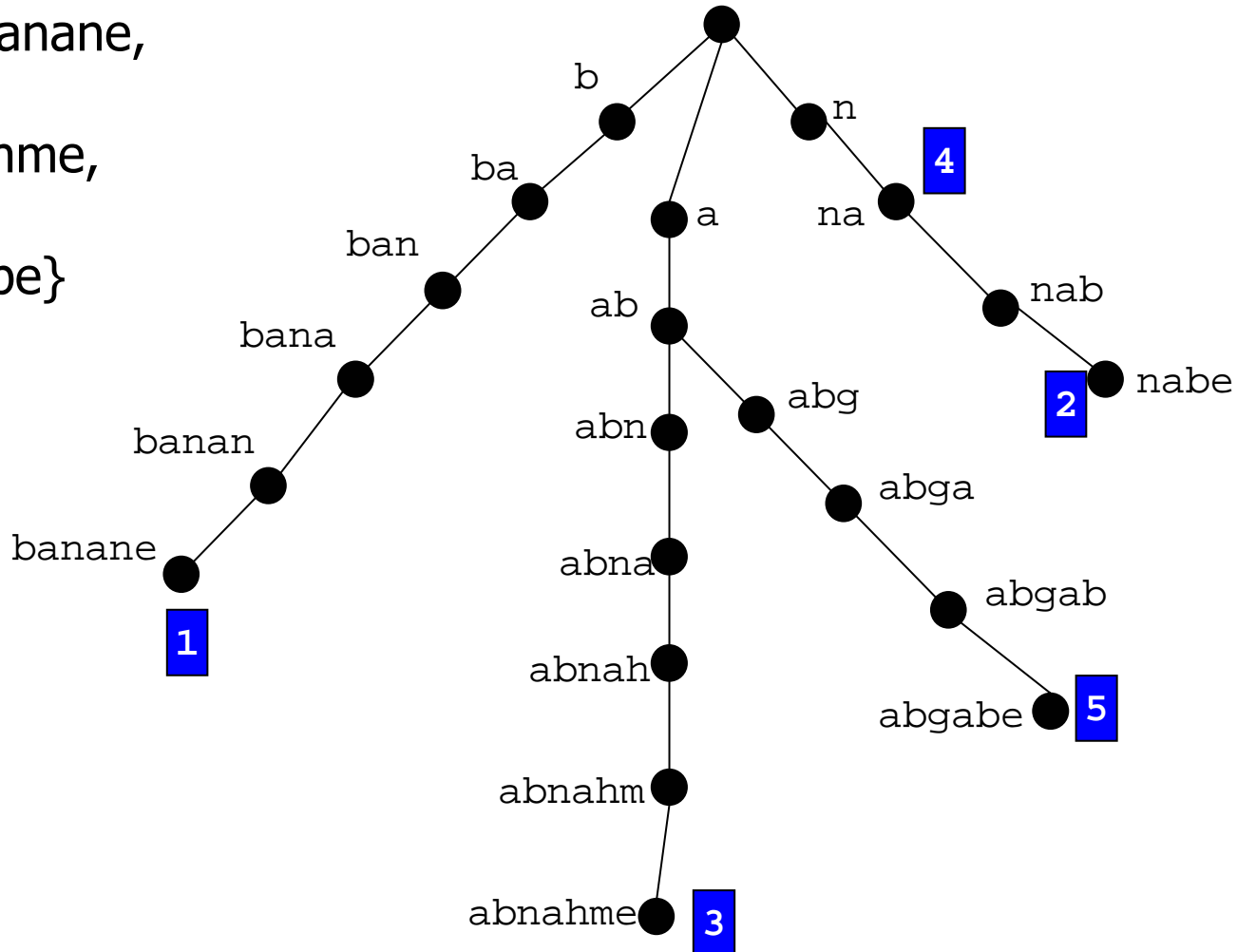
- Nächste Stunde
 - Lineare Berechnung der Failure Links (Preprocessing)
 - Gesamtalgorithmus

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

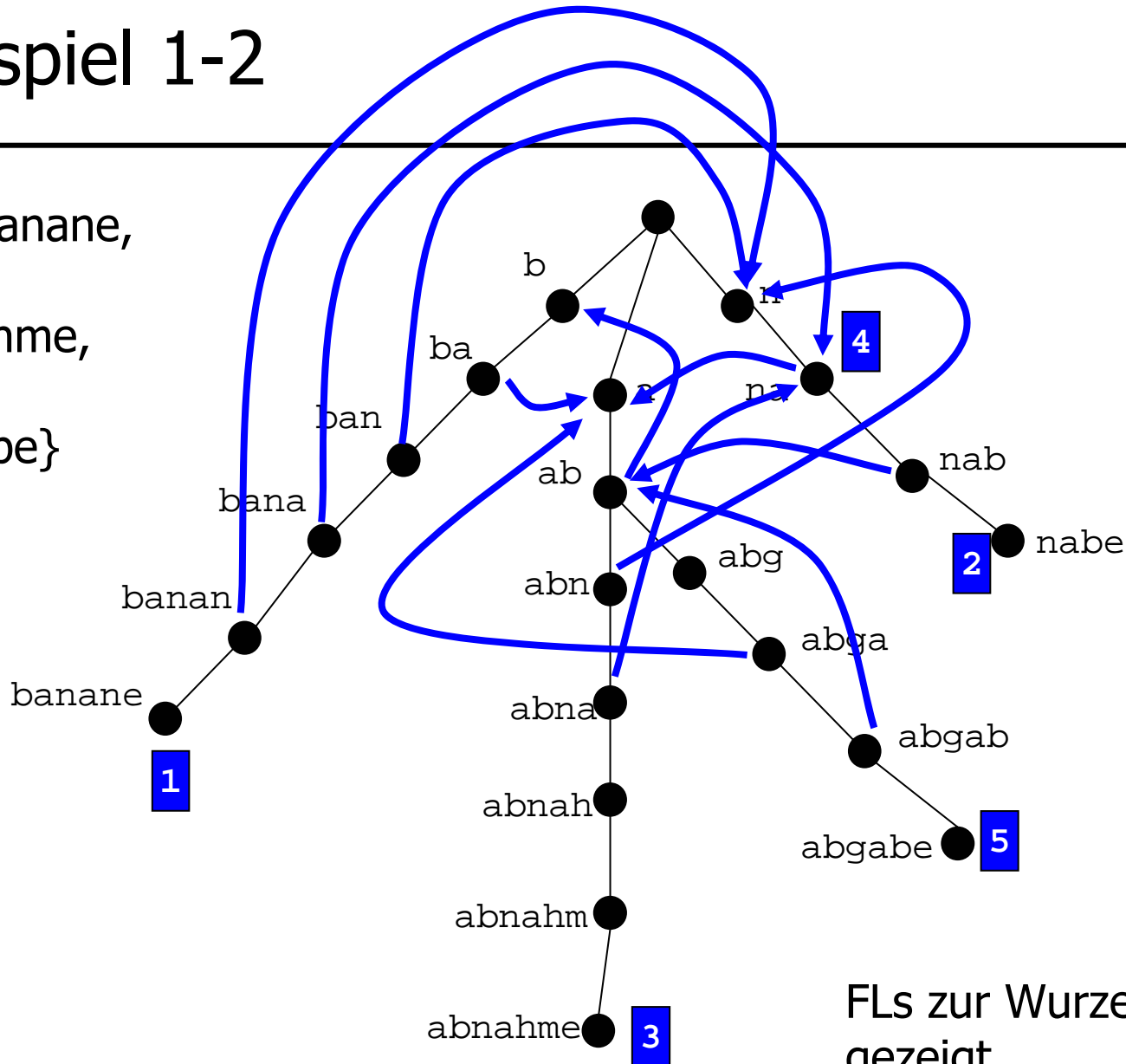
Beispiel 1-1

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



Beispiel 1-2

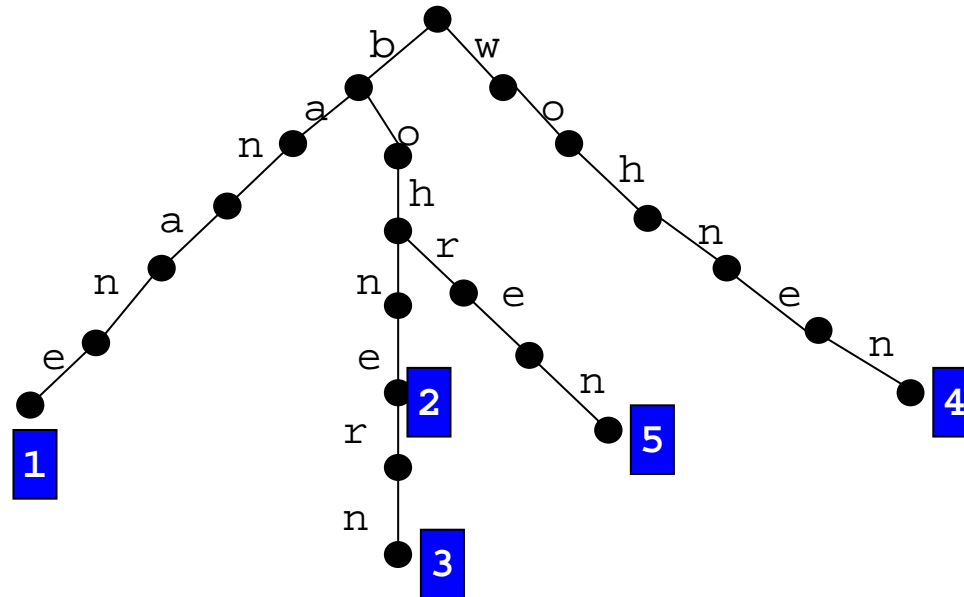
$P = \{\text{banane, nabe, abnahme, na, abgabe}\}$



FLs zur Wurzel nicht gezeigt

Beispiel 2

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



- 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, T
- Wir suchen ab Position j in T
- Matche Substring $T[j..]$ in K
 - Bei Match gehe weiter den Baum hinab
 - $j=j+1$
 - Wenn 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 zu Knoten $fl(k)$
 - $label(fl(k))$ haben wir gerade in T gesehen
 - Bzw. $fl(k)$ geht zur Wurzel
 - Matche weiter in T ab $j+x$ und in K ab $fl(k)$

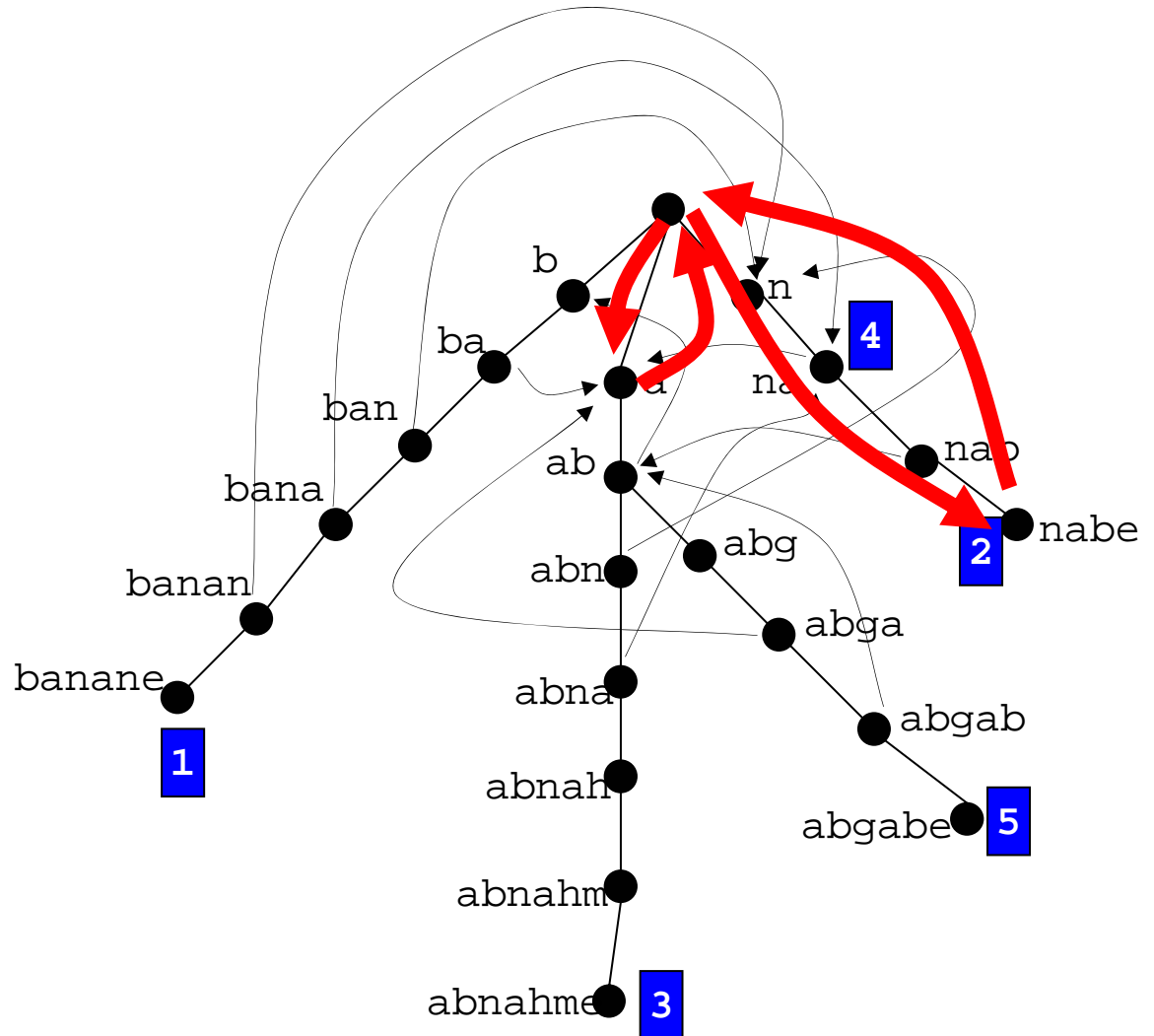
Suchen mit Failure Links

- 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 Mismatches für $T(j+x)$
 - 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 $j+x$ und in K ab $fl(k)$

Beispiel

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

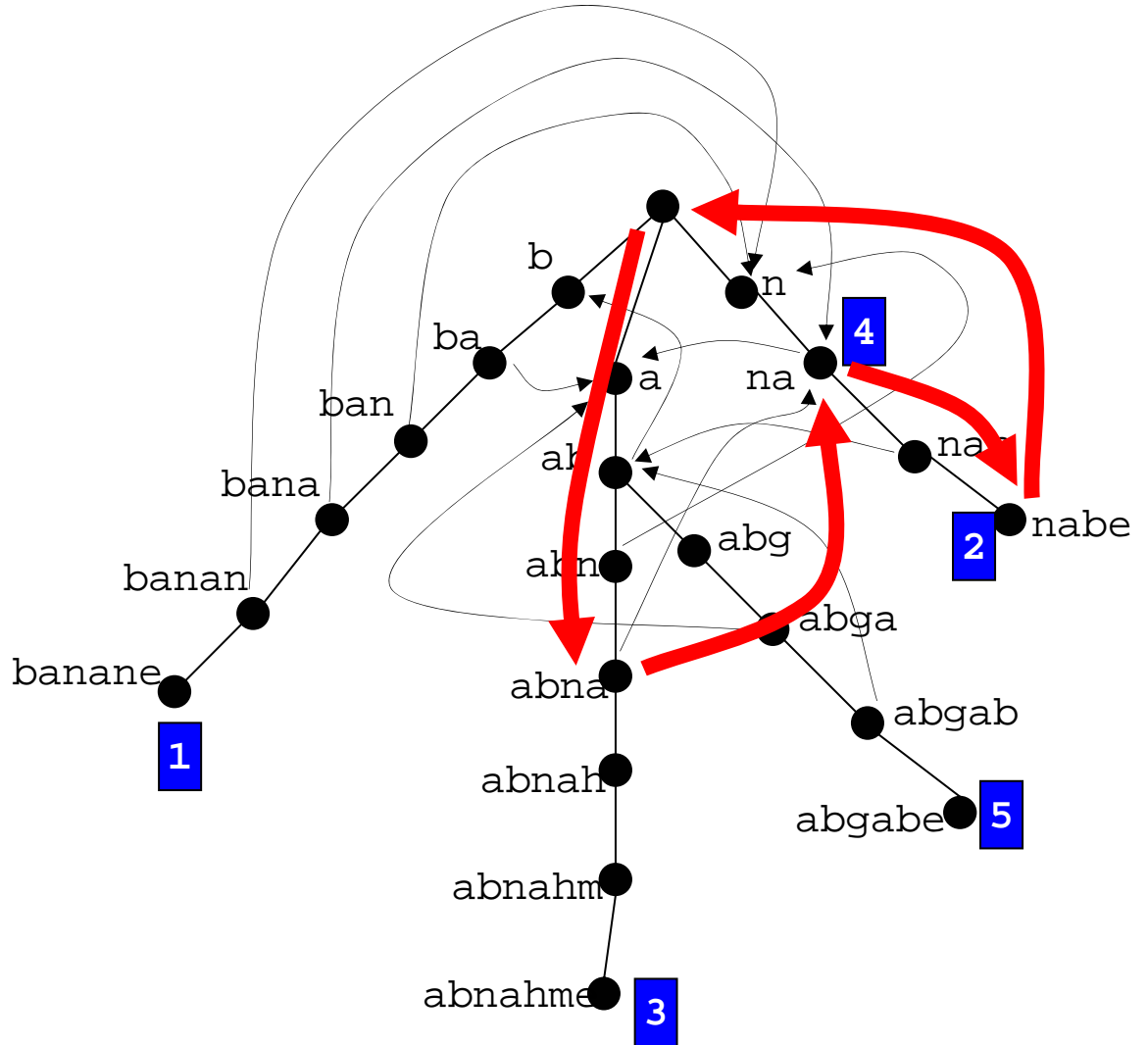
$T = \text{radnaben}$



Beispiel 2

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

$T = \text{abnabeln}$



Algorithmus

```
j := 1;           // Next comparison in T
l := 1;           // Start of pattern in T
k := root(K);     // Current node in keyword tree
repeat
    while exists edge (k,k') with label T(j)
        if mark(k')≠NULL then
            report mark(k') with start l;
        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;
        l := l+1;
    else
        k := fl(k); // Follow the failure link
        l := j-len(k);
    end if;
end repeat;
```

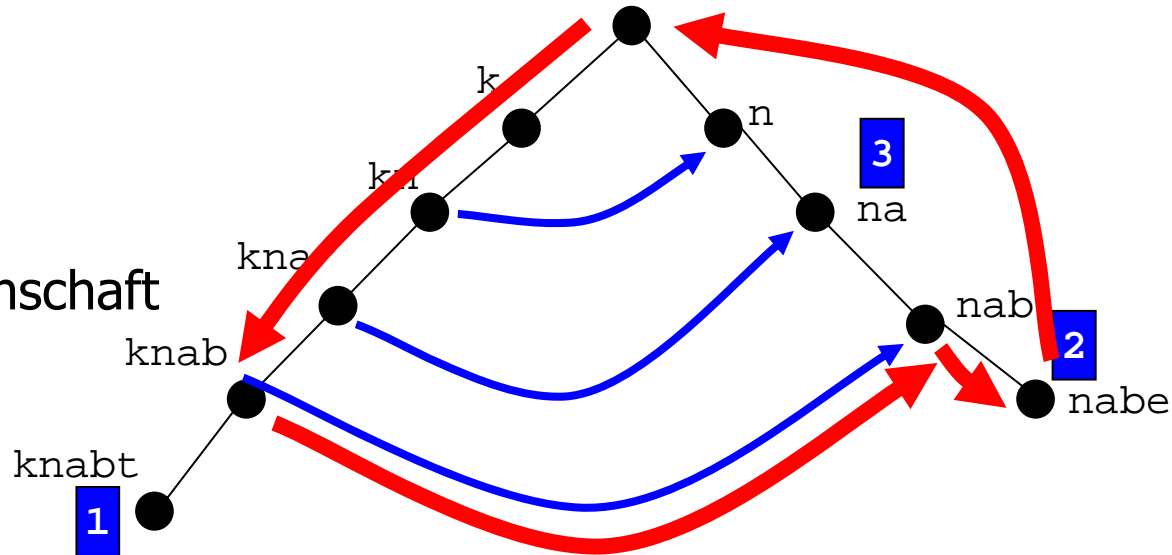
- I nur notwendig, wenn Startposition der Matches verlangt
- Komplexität $O(m)$ (Beweis ähnlich wie bei KMP)



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

